Become a Code Breaker with Python

A beginner's guide to cryptography and computer programming with Python

Al Sweigart

Become a Code Breaker with Python

ROUGH DRAFT v1

NOTE: This is a rough draft and a work in progress. The latest version will be freely available online at http://becomeacodebreaker.com. (currently the website is not done)

TODO: Add study questions to the end.

Also add ciphertexts to break.

Have a “get the latest version of this book” notice.

Cover photo credit: walkn http://www.flickr.com/photos/walkn/

Chapter 1 – How Encryption Works: Making a Cipher Wheel

What is Cryptography?

Look at the following two pieces of text:

"Zsijwxyfsi niqjsjxx gjyyjw. Ny nx jnymjw ktqqd tw bnxitr; ny nx anwyzj ns bjfqym fsi anhj ns utajwyd. Ns ymj bnsyjw tk tzw qnkj, bj hfs jsotd ns ujfhj ymj kwznyx bmnhm ns nyx xuwnsl tzw nsizxywd uqfsyji. Htzwynjwx tk lqtwd, bwnyjwx tw bfwwntwx, xqzrgjw nx ujwrnyyji dtz, gzy tsqd zuts qfzwjqx."

"Flwyt tsytbbnz jqtw yjxndwri iyn fqq knqrqt xj mh ndyn jxwqswbj. Dyi jjkxxx sg ttwt gdhz js jwsn; wnjyiyb aijnn snagdqt nnjwww, xstsxsu jdnxzz xkw znfs uwwh xni xjzw jzwyjy jwnmns mnyfjx. Stjj wwzj ti fnu, qt uyko qqsbay jmwskj. Sxitwru nwnqn nxfzfbl yy hnwydsj uyfzwjzj 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, however, 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. If someone is a clever cryptanalyst, they might be able to break the code. A **cryptanalyst** is someone who can break secret codes and read other people's encrypted messages, even if they were not the person who encrypted the message. Cryptanalysts are kind of like computer hackers. This book will also teach you what you need to know to become a cryptanalyst.

Before we learn how to program computers to do encryption and decryption for us, let's learn how to do it ourselves on paper. 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. We will learn several different ciphers in this book.

Making a Cipher Wheel

Let's learn a cipher called the Caesar Cipher. This is a cipher that 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 just as a simple learning exercise. The Caesar Cipher is also explained on Wikipedia here: http://en.wikipedia.org/wiki/Caesar_cipher

To convert plaintext to ciphertext using the Caesar Cipher, we will create something called a cipher wheel. You can either photocopy the cipher wheel that appears in this book, or print out the one at http://becomeacodebreaker.com/cipherwheel.pdf. Then cut out the two circles like in Figure 1 and Figure 2.

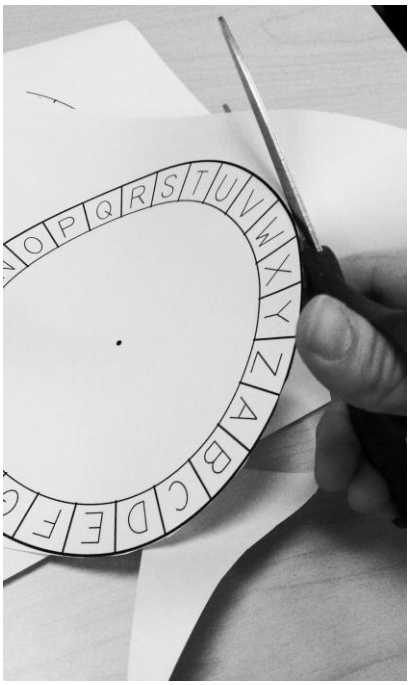


Figure 1 - Cut out the cipher wheel circles.

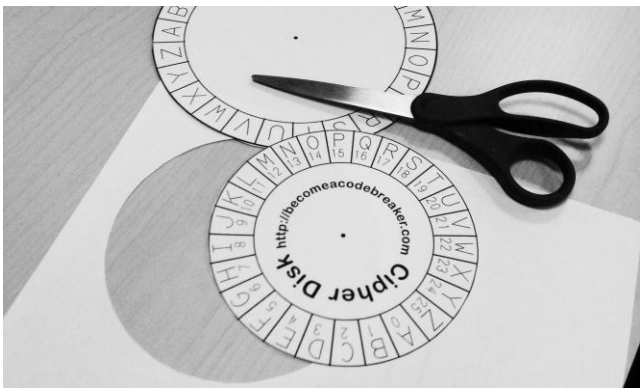
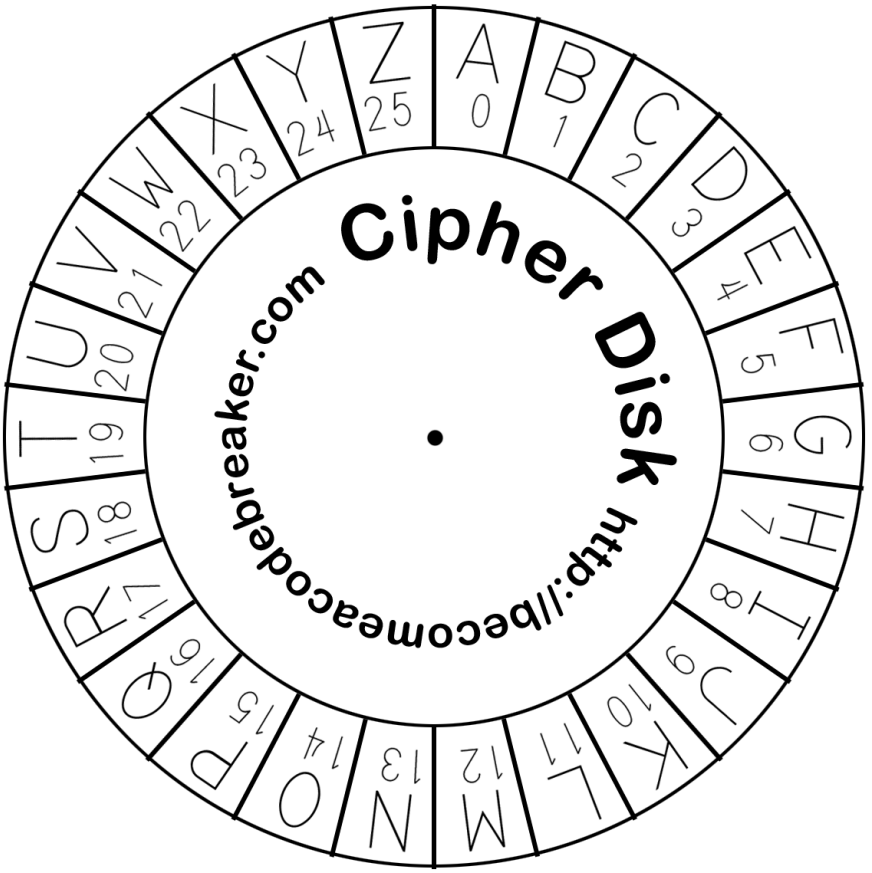
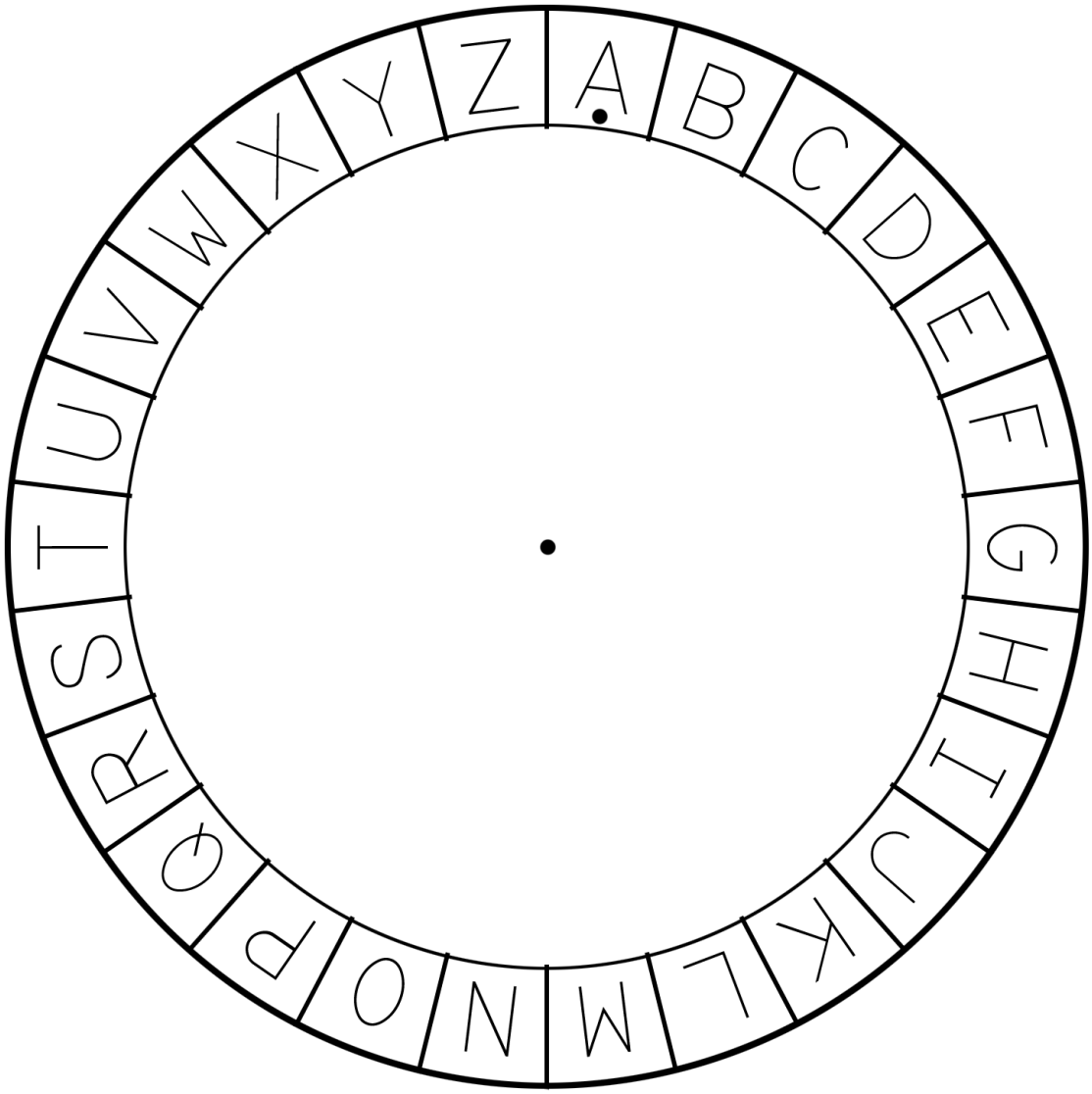


Figure 2 - The cut out circles.



Don’t cut out the page from this book!

Just make a photocopy of this page, and cut out the photocopy.



Don’t cut out the page from this book!

Just make a photocopy of this page, and cut out the photocopy.

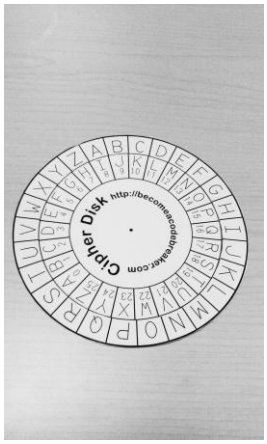


Figure 3 - The completed cipher wheel.

After you cut out the circles, place the smaller one on top of the larger one in the middle. Put a pin or brad through the center of both circles so you can spin them around, like in Figure 3. You now have a tool for creating secret messages with the Caesar Cipher. Let's learn how to use this tool.

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 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 a critical piece of information that determines how a message is encrypted or decrypted. Anyone who reads this book can learn about the Caesar Cipher. But just like a regular lock and key, unless they have the encryption key they will not be able to decrypt the secret message. In Figure 3, the outer circle's A is over the inner circle's number 8. That means we will be using the key 8 to encrypt out message. The Caesar Cipher uses the keys from 1 to 25. Let's use the key 8 for our example. Keep the encryption key a secret; the message can be read by anyone who knows that the ciphertext was encrypted with key 8.

For each letter in our message, we will find where it is in the outer circle and replace the message's letter 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, which 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 just 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).

To decrypt a ciphertext, just 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 cryptanalyst). But your friend has decided to use the key 15 for each message he 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. The first letter in the secret message is I, so we find I on the inner circle and look at the corresponding letter on the outer circle, which is T. The W in the ciphertext will translate to the letter H. One by one, we can decrypt the ciphertext back to the plaintext, "The new password is Swordfish."

What happens if we try to decrypt a ciphertext with the wrong key? If we use the wrong key 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 Cipher Wheel without the Wheel

The cipher wheel is nice because it is easy to spin around to different keys easily. But drawing one yourself can be pretty hard, because each of the slices must be the exact same size or else they won't line up for all the possible keys. But we can make another tool that helps us encrypt and decrypt using the Caesar Cipher.

Find any sheet of paper and write out the letters of the alphabet from A to Z. Then, starting with A, write out the numbers 0 to 25 under each letter. So 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

Now to encrypt, we find the number under the letter we wish to encrypt, and add the key to it. So if we have a plain English sentence like, "Hello. How are you?" and encrypt it with the key 13, first we find the number under the H. That number is 7. Then we add the key to this number. 7 + 13 = 20. The number 20 is under the letter U, so the encrypted letter that we will put in the ciphertext is 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.

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. To solve this problem, we need a special rule. If the sum of the letter's number and the key is 26 or more, we should also subtract 26 from

it. So 27 – 26 is 1. We look at the letter above the number 1, and it 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 message, "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.

2) Find the letter's number.

3) Add the key to the letter's number.

4) If this number is larger than 26 or more, 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. The first plaintext letter is H, which has the number 7. The key is 13, and 7 + 13 = 20. Since 20 is not equal or larger than 26, we do not subtract 26. The number 20 has the letter U, so U is the ciphertext letter of H.

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

You will have to understand negative numbers to decrypt. If you don't know how to add and subtract with negative numbers, there is a tutorial on it here: http://becomeacodebreaker.com/moreinfo.

To decrypt using the letters and numbers instead of a cipher wheel, just 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 should add 26. -12 + 26 is 14. So the ciphertext letter B decrypts back to letter O.

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

As you can see, we don't need an actual cipher wheel to encrypt and decrypt text using 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.

Double Encryption?

You might think that encrypting our message once and then encrypting the ciphertext of that message would double the strength of our encryption. But this turns out to not 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.

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, which is exactly what we do when we encrypt with the key 7.

For most encryption ciphers, encrypting more than once does not provide addition strength to the cipher. In fact, if you encrypt some plaintext with the key 13 twice, the ciphertext you end up with will be the same as the original plaintext!

Programming a Computer to do Encryption

Of course, if you had a very long message (say, an entire book) that you wanted to encrypt, it would take you hours or days to encrypt it all. This is how programming can help. A computer could do these math operations for a large amount of text in less than a second. But we need to learn how to instruct 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 French or Spanish. You don't even to know much math besides adding, subtracting, and multiplication. You just need to download some free software called Python, which we will cover in the next chapter.

Chapter 2 – Downloading and Installing Python

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 and your programs won't work. (We'll just 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 first, from the official website of the Python programming language, http://www.python.org. You might want the help of someone else to download and install the Python software. The installation is a little different depending on if your computer's operating system is Windows, Mac OS X, or a Linux OS such as Ubuntu. You can also find videos of people installing the Python software online. A list of these videos is at http://becomeacodebreaker.com/installing

Windows Instructions

When you get to python.org, you should see a list of links on the left (About, News, Documentation, Download, and so on.) Click on the Download link to go to the download page, then look for the file called Python 3.2 Windows Installer (Windows binary -- does not include source) and click on its link to download Python for Windows.

Double-click on the python-3.2.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 just accept the choices in the installer as you go (no need to make any changes). When the install is finished, click Finish.

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.

Mac OS X Instructions

The installation for Mac 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 "Mac Installer disk image (3.2)" 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 python3.2 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 idle3. You will also need the root password to install IDLE.

A video tutorial of how to install Python is available from this book's website at http://becomeacodebreaker.com/videos.

Downloading pyperclip.py

Almost every program in this book uses a custom module 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 here:

http://becomeacodebreaker.com/pyperclip.py

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

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

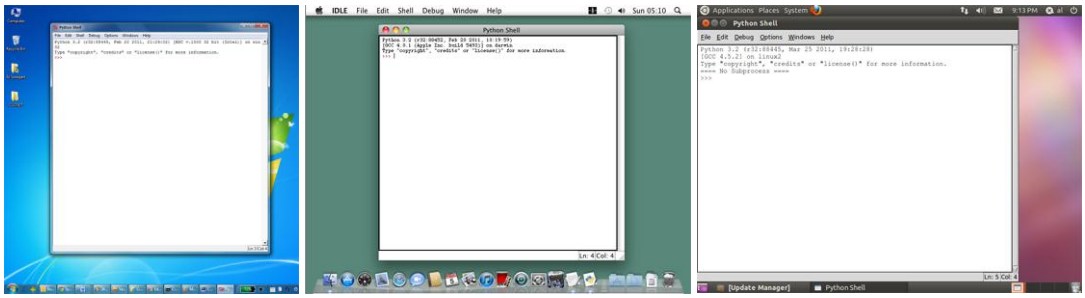
Starting Python

We will be using the IDLE software to type in our programs and run them. IDLE stands for Interactive DeveLopment Environment. The development environment is software that makes it easy to write Python programs.

If your operating system is Windows XP, you should be able to run Python by clicking the Start button, then selecting Programs, Python 3.1, IDLE (Python GUI). For Windows Vista or Windows 7, just 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 click on Applications, then click Python 3.2, then click the IDLE icon.

If your operating system is Ubuntu or Linux, start IDLE by opening a terminal window 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.



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, and the shell sends these instructions to software called the Python interpreter to perform. We can type Python instructions into the shell and, because the shell is interactive, the computer will read our instructions and respond in some way. (Ideally in a way that we expect but that will depend on whether we write the correct instructions.)

How to Use This Book

There are a few things you should understand about this book before you get started. "Invent with Python" is different from other programming books because it focuses on the complete source code for different cryptography programs. Instead of teaching you programming concepts and leaving it up to you to figure out how to make programs with those concepts, this book shows you these programs and then explains how they are put together.

The Featured Programs

Most chapters begin with a sample run of a cryptography program. This sample run shows you what the program outputs in light text and what the user types in shown as black text.

These chapters also show the complete source code of the program, but remember: you don't have to enter every line of code right now. Instead, you can read the chapter first to understand what each line of code does and then try entering it later.

You can also download the source code file from this book's website. In a web browser, go to the URL http://becomeacodebreaker.com/source and follow the instructions to download the source code file.

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!')
```

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. 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 ("whil") 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. For example, the code below has only two lines of code, even though the first line wraps around:

```
1. print('This is the first line! xxxxxxxxxxxxxxxxxxxxxxxxxxxxxx
xxxxxxxxxxxxxx')
2. print('This is the second line! ')
```

Tracing the Program Online

You can visit http://becomeacodebreaker.com/traces to see a trace through each of the programs in this book. Tracing a program means to step through the code one line at a time, in the same way that a computer would execute it. 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 Code Online

Some of the programs in this book are a little long. 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 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 program.

Copying and pasting text is a very useful computer skill, especially for computer programming. There is a video tutorial on copying and pasting at this book's website at http://becomeacodebreaker.com/videos.

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

More Info Links

There is a lot that you can learn about programming and cryptography. But you don't need to learn all of it now. There are several times in this book where you might like to learn these additional details and explanations, but if I included them in this book then it would add many more pages. If this larger, heavier book accidentally fell on you, the weight of these many additional pages would crush you. Instead, I have included "more info" links in this book that you can follow to this book's website. You do not have to read this additional information to understand anything in this book, but it is there if you are curious.

Even though this book is not dangerously heavy, please do not let it fall on you anyway.

Programming and Cryptography

Programming and cryptography are two separate skills, but learning both is useful because a computer can do cryptography much faster than a human can. Imagine how long it would take to encrypt a 400 page book using the cipher wheel from chapter 1. It would take months, and you would probably make some mistakes while encrypting or decrypting. But if you can program a computer to do the encryption, the computer will be able to do it in seconds with zero mistakes.

The next few chapters will teach you basic programming skills by explaining how Python's interactive shell works. If you already know how to program in Python (by reading a book like "Invent Your Own Computer Games with Python", which is online for free at http://inventwithpython.com) then you can skip ahead to the cryptography chapters starting at chapter ten.

Chapter 3 – The Interactive Shell

Some Simple Math Stuff

To open IDLE on Windows Vista or Windows 7, click on the Windows Logo and then type "IDLE" to bring up the IDLE menu item and then click on it. On Windows XP, click on Start, then Programs, then Python 3.1, then IDLE (Python GUI). With IDLE open, let's do some simple math with Python. 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 2-1, the computer should respond with the number 4; the sum of 2+2.

You've just typed in some code (also called instructions) for the computer to perform. When the computer carries out the instruction you gave it, it is executing your code or running your code. A program is just a long list of code, just like a book is a long list of words.

As you can see, we can use the Python interactive shell just like a calculator. This isn't a program by itself because we are just learning the basics right now. The interactive shell executes one line of Python code at a time, so we can use it to see what each line of code does. 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 use a forward slash (/).

When used in this way, `+`, `-`, `*`, and `/` are called operators because they tell the computer to perform the specified operation on the numbers next to them.

Integers and Floating Point Numbers

In programming (and also in mathematics), 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 not integers. Numbers with a decimal point are called floating point numbers. . In Python, the number `5` is an integer, but if we wrote it as `5.0` it would not be an integer. In mathematics, 5.0 is still considered an integer and the same as the number 5, but in computer programming the computer considers any number with a decimal point as not an integer.

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
```

When you type these lines of code into the interactive shell, it will look like this:

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

These math problems are called expressions. Computers can solve millions of these problems in seconds. Expressions are made up of values (the numbers) connected by operators (the math signs). Let's learn exactly what values and operators are.

As you can see with the last expression in the above example, you can put any amount of spaces in between the integers and these operators. But be sure to always start at the very beginning of the line, with no spaces in front.

In programming, we call integers and floats data types. Every value has a data type. The data type of the value 5 is integer, and the data type of the value 5.0 is float. In the next chapter, we will learn about working with text in expressions. Python isn't limited to just numbers. It's more than just a fancy calculator!

Evaluating Expressions

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

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

However, if you just type `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 the + operator always expects to connect two things in Python. We have only given it one. 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. Python will always display an error message if you enter an instruction that it cannot understand.

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.

Expressions Inside Other Expressions

Expressions can also contain other expressions. For example, in the expression `2 + 5 + 8`, the `2 + 5` part is its own expression. Python evaluates `2 + 5` to `7`, so the original expression becomes `7 + 8`. Python then evaluates this expression to `15`.

Think of an expression as being a stack of pancakes. If you put two stacks of pancakes together, you still have a stack of pancakes. And a large stack of pancakes can be made up of smaller stacks of pancakes that were put together. Expressions can be combined together to form larger expressions in the same way. But no matter how big an expression is it also evaluates to a single answer, just like `2 + 5 + 8` evaluates to `15`.

Storing Values in Variables

When we program, we will often want to save the values that our expressions evaluate to so we can use them later in the program. We can store values in variables.

Think of variables 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 2-4: Variables are like boxes 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 2-4). 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 (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, which 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 the instruction evaluates to a single value, it's an expression. If the instruction does not, then it's a statement.

An assignment statement is written as a variable, followed by the = equal sign, followed by an expression. The value that the expression evaluates to is stored inside the variable. The value `15` by itself is an expression. Expressions made up of a single value by itself are easy to evaluate. These expressions just evaluate to the value itself. For example, the expression `15` evaluates to `15`!

Remember, variables store 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`.

The first time you store a value inside a variable by using an assignment statement, Python will create that variable. Each time after that, an assignment statement only replaces the value stored in the variable.

Now let's see if we've created our variable properly. If we type `spam` into the shell by itself, we should see what value is stored inside the variable `spam`.

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

Now, `spam` evaluates to the value inside the variable, `15`.

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

```
>>> 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, writing spam + 5 is like writing the expression 15 + 5.

If you try to use a variable before it has been created, Python will give you an error because no such variable would exist yet. This also happens if you mistype the name of the variable.

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 replaced, or overwritten, with the value 3. Now, when we enter spam + 5, the expression evaluates to 8 because the value of spam is now 3.

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

Now here's something interesting. Because a variable is only a name for a value, we can write expressions with variables like this:

```
>>> spam = 15
>>> spam + spam
30
>>> spam - spam
0
>>>
```

When the variable spam has the integer value 15 stored in it, entering spam + spam is the same as entering 15 + 15, which evaluates to 30. And spam - spam is the same as 15 - 15, which evaluates to 0. The expressions above use the variable spam twice. You can use variables as many times as you want in expressions. Remember that Python will evaluate a variable name to the value that is stored inside that variable, each time the variable is used.

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 also 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

When we program we won't always want to be limited to only one variable. Often we'll need to use multiple variables.

For example, let's assign different values to two variables named `eggs` and `fizz`, like so:

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

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

Without changing the value in our `spam` variable, 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`.

```
>>> 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`.

Overwriting Variables

Changing the value stored inside a variable is easy. Just perform another assignment statement with the same variable. Look what happens when you enter the following code into the interactive shell:

```
>>> spam = 42
>>> print(spam)
42
>>> spam = 100
>>> print(spam)
100
```

Initially, the `spam` variable had the integer `42` placed inside of it. This is why the first `print(spam)` prints out `42`. But when we execute `spam = 100`, the `42` value is tossed out of the variable and forgotten as the new `100` value is placed inside the `spam` variable.

Replacing the value in a variable with a new value is called overwriting the value. It is important to know that the old value is permanently forgotten. If you want to remember this value so you can use it later in your program, store it in a different variable before overwriting the value:

```
>>> spam = 42
>>> print(spam)
42
>>> oldSpam = spam
>>> spam = 100
>>> print(spam)
100
>>> print(oldSpam)
42
```

In the above example, before overwriting the value in `spam`, we copy that value to a variable named `oldSpam`. At that point, both `spam` and `oldSpam` store the value `42`. On the next line, the integer `100` is stored in `spam` but `oldSpam` is left untouched.

Chapter 4 – Strings

That's enough of integers and math for now. Python is more than just a calculator. Now let's see what Python can do with text. In this chapter, we will learn how to store text in variables, combine text together, and display them on the screen. Many of our programs will use text to display information to the user, and the user will enter text into our programs through the keyboard. We will also make our first program, which greets the user with the text, "Hello World!" and asks for the user's name.

Strings

In Python, we work with little chunks of text called strings. We can store string values inside variables just like we can store number values inside variables. When we type strings, we put them in between two single quotes ('), like this:

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

The single quotes are there only to tell the computer where the string begins and ends (and are not part of the string value).

Now, 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 the variable (in this case, the string 'Hello').

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

Strings can have almost any keyboard character in them. (Strings can't have single quotes inside of them without using escape characters. Escape characters are described later.) Instead of using single quotes, you can also use the double quotes to begin and end strings. These are all examples of strings:

```
'hello'
'Hi there!'
"KITTENS"
'7 apples, 14 oranges, 3 lemons'
"Anything not pertaining to elephants is irrelephant."
```

```
'A long time ago in a galaxy far, far away...'
'O*&#wY%*&OCfsdYO*&gfC%YO*&%3yc8r2'
```

As we did with numerical values in the previous chapter, we can also combine string values together with operators to make expressions.

Escape Characters

It can be tricky to type in some characters that you'd like to be in a string. For example, putting a single quote character in a string can cause Python to read that as the end of the string, with some extra text after it. For example, type the following into the interactive shell:

```
>>> 'That is Susie's cat.'
SyntaxError: invalid syntax
```

To make Python look at the single quote in the string as a single quote instead of the end of the string, we need to escape it. Escape characters are characters preceded by a backslash \. Here's a list of common escape characters:

Escape Character	Meaning
\'	Single quote character
\"	Double quote character
\t	Tab character
\n	Newline character
\\	Backslash character

Now try typing this into the interactive shell:

```
>>> 'That is Susie\'s cat.'
"That is Susie's cat."
```

String Concatenation

You can add one string to the end of another by using the + operator, which is called string concatenation. Try entering 'Hello' + 'World!' into the shell:

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

To keep the strings separate, put a space at the end of the 'Hello' string, before the single quote, like this:

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

The + operator works differently on strings and integers because they are different data types. All values have 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 * multiplication operator can be used with a string and integer value to do string replication. For example, type the following into the interactive shell:

```
>>> 'Hello' * 5
'HelloHelloHelloHelloHello'
>>>
```

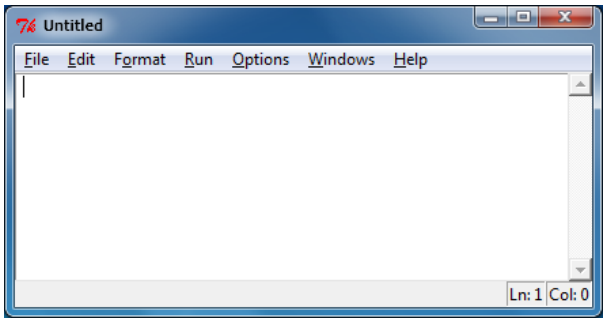
String replication is a neat trick if you need to create a large string made up of the same smaller string over and over again.

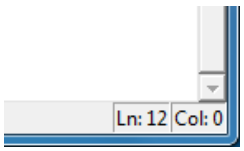
Writing Programs in IDLE's File Editor

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 all at once. Let's write our first program!

The name of the program that provides the interactive shell is called IDLE, the Interactive DeveLopement Environment. IDLE also has another part called the file editor.

Click on the File menu at the top of the Python Shell window, and select 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 tells you where the cursor is. The cursor is currently on line 12.

The Hello World Program

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.

When you enter your program, don't enter the numbers at the left side of the code. They're there so we can refer to each line by number in our explanation. If you look at the bottom-right corner of the file editor window, it will tell you which line the cursor is currently on.

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. (Remember, don't type in the line numbers!)

IMPORTANT NOTE! The following program should be run by the Python 3 interpreter, not the Python 2.6 (or any other 2.x version). Be sure that you have the correct version of Python installed. (If you already have Python 2 installed, you can have Python 3 installed at the same time.) To download Python 3, go to http://python.org/download/releases/3.1.1/ and install this version.

hello.py

This code can be downloaded from http://inventwithpython.com/hello.py

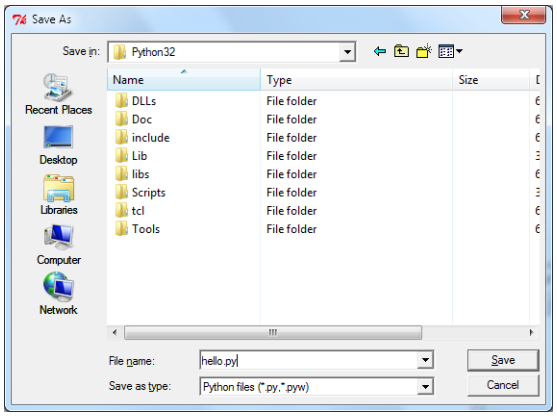
If you get errors after typing this code in, compare it to the book's code with the online diff tool at http://inventwithpython.com/diff or email the author at al@inventwithpython.com

```
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:

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, choose the File menu at the top of the File Editor window, and then click on Save As. The Save As window should open. Enter hello.py in the File Name box then press Save.



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, only the typing you've done since your last save will be lost. Press Ctrl-S to save your file quickly, without using the mouse at all.

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

If you get an error that looks like this:

```
Hello world!
What is your name?
Albert
Traceback (most recent call last):
```

```
  File "C:/Python26/test1.py", line 4, in <module>
    myName = input()
  File "<string>", line 1, in <module>
NameError: name 'Albert' is not defined
```

...then this means you are running the program with Python 2, instead of Python 3. You can either install Python 3, or convert the source code in this book to Python 2. Appendix A lists the differences between Python 2 and 3 that you will need for this book.

Opening the Programs You've Saved

To load a saved program, choose File > Open. 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.

Now it's time to run our program. From the File menu, choose 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 3-5:

Figure 3-5: What the interactive shell looks like when running the "Hello World" program.

Now, when you push Enter, the program should greet you (the user) 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.)

How the "Hello World" Program Works

How does this program work? Well, each line that we entered is an instruction to the computer that is interpreted by Python in a way that the computer will understand. 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 flow of execution, or just the execution for short.

Now 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.
```

This line is called a comment. Any text following a # sign (called the pound sign) is 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.

Programmers usually put a comment at the top of their code to give their program a title. The IDLE program displays comments in red 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. 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 string 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.

There is much more to learn how making our own functions in the next chapter, but for now we will only learn about using the built-in functions that come with Python.

The print() Function

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

This line is a call to the print function, usually written as print() (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()
```

This line has an assignment statement with a variable (myName) and a function call (input()). When input() is called, the program waits for input; for the user to enter text. The text string that the user enters (your name) becomes the function's output value.

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 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 to the string 'Albert'.

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

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

On the last line we have a print() function again. This time, 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 (that is, stopped running) and all of the variables are forgotten by the computer, including the string we stored in myName. If you try running the program again with a different name, like Carolyn, it will think that's your name.

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

Remember, the computer only does exactly what you program it to do. In this, our first 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
```

This is an important thing to realize about computer programs: The computer doesn't understand anything about the programs you write. All it knows is that the print() function should make a string of text appear on the screen, or that string concatenation joins together two strings. It doesn't know anything about what the strings mean or why you want to print them to the screen. The variable name myName is so that the programmer can remember what data it is supposed to store, but one name is as good as any other for the computer.

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 nAmE. The computer will run the program the same (as long as you consistently use abrahamLincoln or nAmE).

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 name. 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. It is very common to accidentally make bugs in your programs while you write them. This is why it is important that the variable names you choose make sense.

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 whatIHadForBreakfastThisMorning is much easier to read than whatihadforbreakfastthismorning. 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 in variable names makes the program more readable.

Define Variables Before You Use Them

Be sure that your program has an assignment statement to define (that is, create) a variable before you try to use it in your program. Let's say the Hello World program was changed to look like this, with lines 4 and 5 swapped.

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

Now the myName variable is used on line 4, but it isn't defined with an assignment statement until line 5. If you tried to run this program, you would get this error message:

```
Hello world!
What is your name?
Traceback (most recent call last):
  File "C:\test1.py", line 4, in <module>
    print('It is good to meet you, ' + myName)
NameError: name 'myName' is not defined
```

The NameError error appears whenever you try to use a variable that your program hasn't created yet. If you get this error, check your program again to see where you tried to use a variable too early.

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. The user will enter text to the program through the keyboard with the input() function. And the computer will display text on the screen when the print() function is executed.

Strings are just a different data type that we can use in our programs. We can use the + operator to concatenate strings together. Using the + operator to concatenate two strings together to form a new string is just like using the + operator to add two integers to form a new integer (the sum).

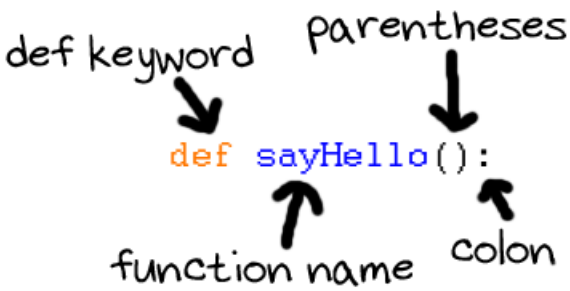
In the next chapter, we will learn more about variables so that our program will remember the text and numbers that the user enters into the program. Once we have learned how to use text, numbers, and variables, we will be ready to start writing programs.

Chapter 5 – Functions

Our Hello World program made use of the print() and input() functions. Functions are kind of like a mini-program within our program. When we call a function, we execute the code inside that function. When the function's code is done, the program execution moves to the next line after the function call.

The code inside the print() function handles displaying text on the screen. The code inside the input() function handles reading the keyboard presses the user makes until she presses the Enter key, and then returns a string that matches the text the user typed. You only need to know what a function does, not how it does it.

In programming, we can write our own functions and then call them. Just like the assignment statement creates variables, the def statement creates functions. A def statement has the def keyword, followed by the name of the function, followed by a set of parentheses and colon.



The code inside of the function includes all the indented lines of code following the def statement. These indented lines are called a block. The spacing at the front of the line is called the indentation.

For example, this sentence is indented.

This sentence is indented even more.

This sentence is indented, but indented less than the previous one.

Open the file editor like you did for the Hello World program, an type in the following code (not including the line numbers of course) and save the file as functions1.py. Select the menu item Run > Run Module or press F5 to run the program:

```
1. def sayHello():
```

```
2.     print('Hello!')
3.     print('Howdy!')
4.
5. sayHello()
6. sayHello()
7. sayHello()
```

You can count the number of spaces by looking at the line above them. Each letter in the source code is the same width. Line 2 begins after line 1's "def ", which is four characters (three letters and a space). So there must be four spaces in front of line 2. In the following picture, the spaces have been replaced by one dot for each space:

```
1. def sayHello():
2. ····print('Hello!')
3. ····print('Howdy!')
```

The two print() statements on line 2 and 3 following the def statement is the function's code. These lines are in the def statement's def-block. Line 5 (and every line after it) is not in the def-block, because it is not indented. Indentation marks the start and end of blocks. Blocks are lines of code that are grouped together. A block *begins* when the indentation of the code increases, and a block *ends* when the indentation decreases back to the original level.

The def statement on line 1 defines (that is, creates) a new function called sayHello(). We can call this function and execute the code in it just like we call the print() or input() functions. When we call the sayHello() function, the program execution jumps to the sayHello() function's def-block, executes the code one line at a time going down, and then jumps to the line after the original function call.

Lines 5, 6, and 7 have function calls to the sayHello() function. (They do not define a function because they don't have the def keyword in front of them. And line 1 doesn't call the sayHello() function because it does have the def keyword in front of it.)

When line 5 executes, the program execution jumps to line 2, executes lines 2 and 3. Then it has reached the end of the sayHello() function, so it jumps back to the line after the original function call, which is line 6.

Line 6 also calls the sayHello() function. So again the program execution jumps to line 2, executes the two print() function calls, and then jumps back to the line after the function call, which is line 7.

Line 7 is yet another call to sayHello(). The program execution goes back to line 2, runs the two print() function calls, and then jumps to the line after the function call on line 7. But since there are no more lines after line 7, the program ends.

The sayHello() function prints "Hello!" and "Howdy!", and our program calls sayHello() three times. So "Hello!" and "Howdy!" appear three times on the screen like this:

```
Hello!
Howdy!
Hello!
Howdy!
Hello!
Howdy!
```

If we want to run the same lines of code several times in a program, we can put that code into a function and then call the function several times. Otherwise we would have to type out the same code over and over again.

Functions Calling Functions, Dreams within Dreams

Let's say that when you go to sleep tonight, you have a dream that you are in somebody's mansion. You wander around the hallways of the mansion and walk into the kitchen, and take a look at all the food in the fridge. It seems like you are in an entirely new world, even though you are just dreaming.

Then you wake up and pop out of the dream world and back into the real world.

Let's say that the next night you dream that you are in the mansion again. This time, instead of going into the kitchen, you go into one of the bedrooms in the mansion, climb into bed, and dream that you go to sleep. In the dream, you are dreaming that you are sleeping and having another dream that you are on a submarine. You are having a dream within a dream.

When you look out a window on the submarine, you see fish swimming in water. Everything is cramped and made of metal. In the control room of the submarine are buttons and levers that steer the submarine. In another part of the submarine there are sleeping bunks for the crew of the submarine. In your dream within your dream, you dream that you climb into one of these bunks. You dream that you go to sleep and start having another dream.

You dream that you are in a jungle, and a velociraptor (a very ferocious dinosaur) starts to chase you. Just before it catches and eats you, you wake up from this nightmare.

Then you find yourself on the submarine in the crew bunk. And you wake up again.

Then you find yourself in bed in the mansion. And you wake up again.

Then you find yourself awake in the real world. That is, unless you are not actually reading this book, but having a dream that you are reading this book. In that case, please wake up and continue reading this book in the real world.

When a function calls another function, it is very much like your dream within a dream. When the inner function is done executing and returns (or "wakes up"), it will return to the function that called it. Just like you can have a dream that you dream another dream, a function can call another function which calls yet another function.

In fact, you can have a series of 1000 functions calling functions before the program crashes with an error message. For more information about why this limit is 1000, read http://becomeacodebreaker.com/moreinfo.

In the following program, we have the sayHello() function but also a new function called printLine().

```
 1. def sayHello():
 2.     print('Hello!')
 3.     printLine()
 4.     printLine()
 5.     print('Howdy!')
 6.
 7. def printLine():
 8.     print('----------')
 9.
10. sayHello()
11. sayHello()
12. sayHello()
```

When you run this code, it looks like this

```
Hello!
----------
----------
Howdy!
Hello!
----------
----------
Howdy!
Hello!
----------
----------
```

```
Howdy!
```

The printLine() function just prints out one line each time it is called. The sayHello() function calls printLine() twice, and we call sayHello() three times. This is why the ---------- line shows up six times when we run this program.

Let's change the program again by adding a print3Lines() function, which will call the printLine() function three times. Change the sayHello() function to call print3Lines() instead of printLine(), and then save this program as functions3.py. Run the program by pressing F5.

```
 1. def print3Lines():
 2.     printLine()
 3.     printLine()
 4.     printLine()
 5.
 6. def sayHello():
 7.     print('Hello!')
 8.     print3Lines()
 9.     print('Howdy!')
10.
11. def printLine():
12.     print('----------')
13.
14. sayHello()
15. sayHello()
16. sayHello()
```

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

```
Hello!
----------
----------
----------
Howdy!
Hello!
----------
----------
----------
Howdy!
Hello!
----------
----------
----------
Howdy!
```

Our program now calls sayHello(), which calls print3Lines(), which calls printLine(). Whenever any function call is done, it returns to the code that called it. When printLine() is done, the program execution returns to print3Lines(), not to sayHello().

When we ran the Hello World program in the last chapter, the program execution started at the top and then went down, executing each line. The functions3.py program sort of does the same thing. It starts at the top at line 1. But the def statement is only defining the print3Lines() function (just like an assignment statement defines a variable), it is not calling the function. So the execution skips all the lines in the def-block. The second line the program executes is the def statement on line 6 for sayHello(). Again the execution skips the function's code since it is only defining the function, then executes the def statement for printLine() on line 11.

Then the execution skips down to the function call to sayHello() on 14. A function call makes the execution jump to the start of the function, wherever it is in the program. The sayHello() function begins on line 6, so the execution starts there and goes down. It executes the print() call on line 7, jumping inside, running the print() function's code to make text appear on the screen, then jumping back out. (But we don't need to know how print() works.) Then it goes down to line 8, which calls the print3Lines() function which makes the execution jump to line 2.

But remember that just like we have to define variables with an assignment statement before we can use them, we also have to define functions with a def statement before we call them. The following program would have an error when we try to run it:

```
1. sayHello()
2. def sayHello():
3.     print('Hello!')
```

The error would look like this:

```
Traceback (most recent call last):
  File "C:\test1.py", line 1, in <module>
    sayHello()
NameError: name 'sayHello' is not defined
```

Arguments, Parameters and Return Values

We can change how a function behaves by passing values to it. For example, the print() function does not always print the same thing. It prints something different on the screen depending on what string value you pass to it. The values passed to a function are called arguments, and they go in between the parentheses in a function call. For example, in the function call print('Hello') the string value 'Hello' is the argument.

Function calls can also evaluate to values. In the line of code myName = input(), the function call evaluates to the string value of whatever the user typed in. So if the user typed in Albert, then the input() function call evaluates to the string 'Albert' and the line of code is the same as myName = 'Albert'. The value that a function call evaluates to is called a return value.

We can pass arguments to the functions we create with def statements too. And we can control what return value a function call to our function evaluates to. Look at this program which defines a function named addTwoNumbers() and then calls it:

```
1. def addTwoNumbers(firstNumber, secondNumber):
2.     return firstNumber + secondNumber
3. num = addTwoNumbers(3, 4)
4. print(num)
```

When we run this program, it will define the addTwoNumber() function, then skip down past the def-block and calls the addTwoNumbers() function.

When we call addTwonumbers() on line 3, you can see that two arguments are being passed: the integer values 3 and 4. Looking at the function's def statement on line 1, you can see that there are two variable names in between the parentheses (and separated by a comma). These variables are parameters. Parameters are variables that are assigned to the argument values that are passed when the function is called.

Execution jumps inside of the addTwoNumbers() function when it is called. There is only one line in this function, a return statement. The return statement determines what value the function call will evaluate to. When the return statement is executed, the program execution jumps back to the original function call and the call evaluates to the return value.

In our programs case, the return value is what firstNumber + secondNumber evaluates to, which will be the integer value 7. So the function call on line 3 gets evaluated as 7, and this integer is stored in the num variable. On line 4, the print() call will display 7 on the screen.

To summarize, a function call can pass values to the function. These values are called arguments. The variables in between the parentheses in the def-statement are where these arguments will be stored. These variables are called parameters. Parameters are just a name for a specific kind of variable and arguments are just a name for a specific kind of value. This is just like conditions are just a name for expressions when they are in an if statement. All parameters are just variables and all arguments are just values.

The return keyword is only found inside def-blocks. The return statement usually comes at the end of a function, because once the return statement is executed, the program execution immediately jumps out of the def-block.

If there was any code after the return statement, it would not get executed. For example, in the following function, the “Hello!” text would never ever be printed on the screen:

```
def neverSayHello():
    print('My name is Zophie.')
    return 10
    print('Hello!') # this line is never executed
```

Global Scope and Local Scope

A variable that is created inside a function only exists while that function is being called. Look at the following code:

```
def func():
    spam = 'Hello'

func()
print(spam)
```

If you run this program, you’ll get the following error:

```
NameError: name 'spam' is not defined
```

The reason is that the variable spam only exists while the function func() is being called, and it stops existing when the function returns. So when the print() call tries to display the spam variable, Python complains that no such variable exists.

The code and variables inside a function are said to be in a local scope. The code and variables outside all functions are in a global scope. In the above small program, the def statement, func() function call, and print() function call are in the global scope. The spam assignment statement is inside the func() function’s local scope. Scopes are also called namespaces.

When the program execution leaves that function and returns to the line with original function call, all the variables are forgotten. (This is just like how all the variables in the program are forgotten when the program terminates.)

Variables defined in the global scope can be read outside and inside functions, but can only be modified outside of all functions. Variables defined in a function's local scope can only be read or modified inside that function.

Specifically, we can read the value of global variables from the local scope, but attempting to change the value in a global variable from the local scope will leave the global variable

unchanged. What Python actually does is create a local variable with the same name as the global variable. But Python will consider these to be two different variables.

For example, look at the following small program:

```
def func():
    spam = 'Hello'

spam = 'Monkeys!'
func()
print(spam)
```

The spam variable in func()'s local scope is completely different than the spam variable in the global scope. So when the spam variable inside func() gets assigned the 'Hello' string value, this is not changing the global scope variable spam that has 'Monkeys!' stored in it. They have the same name, but they are in different scopes, so they are different variables. (Just like how variables with the same name but different cases like spam, SPAM, spAM, and sPaM are all different variables.)

Also, global variables cannot be read from a local scope if you modify that variable inside the local scope. For example, if you had a variable named spam in the global scope but also modified a variable named spam in the local scope (say, with an assignment statement) then the name "spam" can only refer to the local scope variable.

Look at this example to see what happens when you try to change a global variable from inside a local scope. Remember that the code in the funky() function isn't run until the funky() function is called. The comments explain what is going on:

```
def funky():
    # We create a local variable named "spam"
    # instead of changing the value of the global
    # variable "spam":
    spam = 99

    # The name "spam" now refers to the local
    # variable only for the rest of this
    # function:
    print(spam)    # 99

# A global variable named "spam":
spam = 42
print(spam) # 42

# Call the funky() function:
```

```
funky()

# The global variable was not changed in funky():
print(spam)    # 42
```

When run, this code will output the following:

```
42
99
42
```

Functions are very useful concepts in our programs, because it allows us to organize our code. You will see how functions are used in actual software in the cryptography programs in this book.

Chapter 6 – The Caesar Cipher

Implementing a Program

The Caesar Cipher isn't a computer program. It is just a series of steps to turn a string of text into another string of text. In Chapter 1, we used a cipher wheel and then 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 benefit of this is that the computer can encrypt and decrypt millions of times faster than a person with a cipher wheel can and will never make a single mistake. Another benefit is that while it takes a programmer to write a Caesar Cipher program, you don't need to be a programmer to use the program to encrypt and decrypt text. The user doesn't even need to know anything about programming or how the Caesar Cipher works! They just need to know how to use the program, which is easier than learning how to use a cipher wheel.

This chapter has two parts. The first part explains several programming concepts that will be used by our Caesar Cipher program. The second part explains the source code of the Caesar Cipher program itself.

Import Statements

Import statements allow your program to use code that is in other Python files. This lets us use Python code that other people have already written in our own programs. When we import these Python files, we call those files **modules**. An import statement is made up of the import keyword followed by the name of the module. You can have several import statements importing one module each, or use an import statement with multiple modules where the modules are separated by commas. Try typing this into the interactive shell:

```
>>> import random
>>> import sys, string
>>>
```

This imports a module named random, which comes with Python. The code for this module exists in a file named random.py. To import multiple modules in the same import statement, put a comma in between the module names. Now that we have imported the random module, we can call functions that exist inside of it such as, for example, the random.randint() function.

The random.randint() Function

The random.randint() function is inside the random module. This function takes two integers and then returns a random number between those two integers. Try typing the following into the interactive shell:

```
>>> import random
>>> random.randint(1, 20)
12
>>> random.randint(1, 20)
18
>>> random.randint(1, 20)
3
>>> random.randint(1, 20)
18
>>> random.randint(1, 20)
20
>>>
```

Of course, when you call the random.randint() function on your computer, you will probably get different numbers. This is because each time the randint() function is called, it returns some random number, just like when you roll dice you will get a random number each time.

The random.randint() function will suggest a key to use so that no bias enters the user's choice of key. The possible keys for the Caesar Cipher are the integers from 0 to 25, but we don't want to suggest key 0 since the encrypted ciphertext will be the same as the original plaintext. So we call random.randint() and pass the integers 1 and 25.

The Clipboard and the pyperclip Module

Copying text to and pasting text from the clipboard saves you the trouble of typing a lot of text yourself. To copy text, most applications let you highlight the text you wish to copy and then press Ctrl-C to copy this text to the clipboard. To paste text, most applications let press Ctrl-V. Instead of using the keyboard shortcuts, you can also select Copy or Paste from the Edit menu.

The IDLE program works this same way. When we are encrypting and decrypting a lot of different pieces of text, it will be easier to use the clipboard whenever possible. But it would also be useful if our programs could copy and paste text to and from the clipboard.

Python does not come with a function to programmatically copy and paste text. However, there is a free module you can download that provides these functions. The pyperclip module (pronounced "piper" and "clip") provides and copy() function (to copy text to the clipboard) and a paste() function (to paste text from the clipboard).

To download pyperclip, open your web browser to the URL http://becomeacodebreaker.com/pyperclip.py and click on the menus File, then Save As. If you are running Windows, you want to save this file in the C:\Python31\Lib\site-packages folder. TODO for other operating systems. This will let any Python script you run be able to import the pyperclip module and call its copy() and paste() functions.

After downloading pyperclip.py, try the following in the interactive shell:

```
>>> import pyperclip
>>>
```

If nothing appears after you enter the import statement, then you have downloaded and installed pyperclip correctly. If you see an error message, then make sure you typed the import statement correctly ("paperclip" with a "y" instead of an "a") copied the pyperclip.py file to the C:\Python31\Lib\site-packages folder. The error message looks like this:

```
>>> import pyperclip
Traceback (most recent call last):
  File "<pyshell#3>", line 1, in <module>
    import pyperclipp
ImportError: No module named pyperclip
>>>
```

Try copying some text to the clipboard with the pyperclip.copy() function:

```
>>> pyperclip.copy('Hello world!')
>>>
```

You should now be able to press Ctrl-V in IDLE and have the text "Hello world!" appear. This is a lot faster than typing "Hello world!" out on the keyboard. If you copy some other text to the clipboard, then the "Hello world!" text in the clipboard will be overwritten. (Think of the clipboard as a variable.)

The pyperclip.paste() function will return a string of whatever is on the clipboard():

```
>>> pyperclip.paste()
'Hello world!'
>>>
```

These functions will be used in our source code so that the user can paste the encrypted or decrypted text that the cryptography programs make.

Bool Values and the Boolean Data Type

There are many different values of the integer data type: 2, 40, 100, -5, 0, 1000000. And there are also many possible string values as well: 'hello', 'spam', 'A long time ago in a galaxy far, far away…'.

But the Boolean data type has only two values: True and False. These values are case-sensitive and they are not string values; in other words, you do not put a ' quote character around them. We will use Boolean values (also called bools) with comparison operators to form conditions.

Comparison Operators

The comparison operator is used to compare two values and evaluate to a True or False Boolean value. A list of all the comparison operators is in Table 4-1.

Operator Sign	Operator Name
<	Less than
>	Greater than
<=	Less than or equal to
>=	Greater than or equal to
==	Equal to
!=	Not equal to

Conditions

A condition is an expression that combines two values with a comparison operator (such as < or !=) and evaluates to a Boolean value. A condition is just another name for an expression that evaluates to True or False. You'll find a list of other comparison operators in Table 4-1.

Conditions always evaluate to a Boolean value: either True or False. For example, the condition in our code, mode != 'q' asks "is the value stored in mode not equal to 'q'?" If so, then the condition evaluates to True. If not, the condition evaluates to False.

Experiment with Booleans, Comparison Operators, and Conditions

Enter the following expressions in the interactive shell to see their Boolean results:

```
>>> 0 < 6
True
>>> 6 < 0
False
```

```
>>> 50 < 10
False
>>> 10 < 11
True
>>> 10 < 10
False
```

The condition 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 condition 6 < 0 evaluates to False. 50 is not less than 10, so 50 < 10 is False. 10 is less than 11, so 10 < 11 is True.

But what about 10 < 10? Why does it evaluate to False? It is False because the number 10 is not smaller than the number 10. They are exactly the same size. If a girl named Alice was the same height as a boy named Bob, you wouldn't say that Alice is taller than Bob or that Alice is shorter than Bob. Both of those statements would be false.

Try entering some conditions into the shell to see how these comparison operators work:

```
>>> 10 == 10
True
>>> 10 == 11
False
>>> 11 == 10
False
>>> 10 != 10
False
>>> 10 != 11
True
>>> 'Hello' == 'Hello'
True
>>> 'Hello' == 'Good bye'
False
>>> 'Hello' == 'HELLO'
False
>>> 'Good bye' != '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 equal. It's easy to accidentally use one when you meant to use the other, so be careful of what you type in.

Two values that are different data types will always be not equal to each other. For example, try entering the following into the interactive shell:

```
>>> 42 == 'Hello'
False
>>> 42 != '42'
False
```

Boolean Operators

The condition on line 21 has the or operator. The or operator is a new type of operator called a Boolean operator. Boolean operators compare two Boolean values (also called bools) and evaluate to a single Boolean value. Do you remember how the * operator will combine two integer values and produce a new integer value (the product of the two original integers)? And do you also remember how the + operator can combine two strings and produce a new string value (the concatenation of the two original strings)? The and and or Boolean operators combine two Boolean values to produce a new Boolean value. The not Boolean operator only works on one Boolean value.

The or Boolean Operator

If either or both of the two Boolean values that the or operator combines is True, then the or operator evaluates to True. If both of the Boolean values the or operator combines are False, then the or operator evaluates to False.

Try typing the following into the interactive shell:

```
>>> True or True
True
>>> True or False
True
>>> False or True
True
>>> False or False
False
>>> 5 < 10 or False
True
>>> 5 > 10 or False
False
```

You can think of the or operator as saying, “If the bool on the left is true or the bool on the right is true, then this expression is true. Otherwise it is false.” Notice how the 5 < 10 gets evaluated to True, making 5 < 10 or False the same as True or False. But 5 > 10 is False, so 5 > 10 or False is the same as False or False.

A truth table shows all the possible combinations of how a Boolean operator can result. Here is the truth table for the or operator:

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

The and Boolean Operator

The and operator is similar to the or operator, except it will only evaluate to True if both the bool on the left is True and the the bool on the right is True. If one or both bools are False, then the and operator evaluates to False. Here is the truth table for the and operator:

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

The not Boolean Operator

Unlike the and and or operators, the not operator only works on one Boolean value. It returns the opposite value. Try typing the following into the interactive shell:

```
>>> not True
False
>>> not False
True
>>> not 10 == 10
False
>>> not 10 < 7
True
```

Because 10 == 10 is True, not 10 == 10 will evaluate to False. And because 10 < 7 is False, then not 10 < 7 would evaluate to True.

Here is a truth table for the not operator:

not A	is	Entire Statement
not True	is	False

not False	is	True

Remember that since both not True and not False evaluate to Boolean values themselves, you could even use the not operator on that value as well. Try typing this into the interactive shell:

```
>>> not not True
True
>>> not not False
False
>>> not not not not not not not not 4 < 6
True
```

You can write code this way, but please don't. This code is very silly.

Looping with while Statements

Sometimes in our programs, we want the program to do something over and over again. Rather than typing the same code over and over again, we can use a loop. The while statement marks the beginning of a loop. A while statement has the while keyword, followed by a condition, followed by a colon. The line after the while statement should be a new block of code.

When the execution reaches a while statement, it evaluates the condition next to the while keyword. If the condition evaluates to True, the execution moves inside the while-block. If the condition evaluates to False, the execution skips past the while-block.

Create a new file editor window and type in the following program. Save it as hellohello.py and run it by pressing F5.

```
1. counter = 0
2. while counter < 10:
3.     print('Hello!')
4.     counter = counter + 1
```

This program starts by setting a variable named counter to the integer value 0. The next line is the while statement. The condition, counter < 10, will evaluate to True since counter is set to 0. This means the program execution will enter the while statement's block.

The block has two lines of code. The first line prints 'Hello!' to the screen and the second increases the value in counter by 1. After these two lines execute, the execution has reached the end of the block so it jumps back to line 2 to re-evaluate the condition. Even though counter is now set to 1, the counter < 10 condition is still True. So the block of code gets executed again. In

fact, the block will execute ten times. Each execution through a loop's block of code is called an iteration. The above code example has ten iterations when it is run.

This is how a loop works. As long as the condition is True, the program keeps executing the code inside the while-block repeatedly until we reach the end of the while-block and the condition is False. In fact, if the short hellohello.py program did not have line, then the condition would never become False and the loop would go on forever. This is a type of bug called an infinite loop. (You can have the Python interpreter force the program to stop by pressing Ctrl-C.)

Think of the while statement as saying, "while this condition is true, keep looping through the code in this block".

The if Statement

Open a new file editor window and type in the following program, save it as favColor1.py and press F5 to run it:

```
1. print('What is your favorite color?')
2. favColor = input()
3.
4. if favColor == 'green':
5.     print('Green is my favorite color too.')
6.
7. print('See you later!')
```

When this program is run, it asks you to type in your favorite color. It may look something like this:

```
What is your favorite color?
green
Green is my favorite color too.
See you later!
```

Of course, the user doesn't have to type in a color; they can type in anything. But if the string they type in is 'green', then the line "Green is my favorite color too." is displayed. Otherwise, the program does not execute the print() call on line 5 and the user never sees this text.

```
What is your favorite color?
blue
See you later!
```

The reason line 5 is sometimes run and sometimes not is because it is in a block following an if statement (in this case, on line 4). An if statement is made up of the if keyword, followed by a condition, then a colon. After the if statement comes a block of code that is run if the condition evaluates to True. (You can kind of read line 4 as regular English: "If the value in the favColor variable is equal to the string value 'green', then print 'Green is my favorite color too.') If the condition is False, then that block of code is skipped over, and the program execution continues to go down as normal.

This lets us put code into our program that only runs under certain conditions. In the case of line 4, the block only runs if favColor is equal to 'green'. And favColor is only equal to 'green' if the user typed in 'green' during the input() call. In this way, the user can control what happens in the program.

The else Statement

Try typing in the following code and save it as favColor2.py. This program is slightly different than the last one.

```
1. print('What is your favorite color?')
2. favColor = input()
3.
4. if favColor == 'green':
5.     print('Green is my favorite color too.')
6. else:
7.     print('That is a good color also.')
8.
9. print('See you later!')
```

When you run this program by pressing F5, it looks like this:

```
What is your favorite color?
gophers
That is a good color also.
```

The print() statement runs if (and only if) the condition in the if statement on line 4 was False. An else statement doesn't have a condition following it: it is always just the else keyword with a colon, followed by a block of code. If you read the code like regular English, it says "if favColor is equal to 'green', then print 'Green is my favorite color too.' or else print 'That is a good color also.' An else statement cannot be in the code by itself, it must come after an if statement (or an elif statement, which is described next.)

The elif Statement

Try typing in the following code and save it as favColor2.py. This program is slightly different than the previous two.

```
1.  print('What is your favorite color?')
2.  favColor = input()
3.
4.  if favColor == 'green':
5.      print('Green is my favorite color too.')
6.  elif favColor == 'blue':
7.      print('Blue is my second favorite color.')
8.  elif favColor == 'yellow':
9.      print('I do not like the color yellow.')
10. else:
11.     print('That is a good color also.')
12.
13. print('See you later!')
```

Line 6 and line 8 are elif statements (which you can pronounce as "else-if"). An elif statement is just like an if statement. Elif statements have a condition which is evaluated and is followed by an indented block of code. Elif statements come after if statements. If the if statement's condition is False and the elif statement's condition is True, then the block after the elif statement is executed.

An if statement can be followed by any number of elif statements, and may or may not have an else statement at the end. If there are any other elif statements or an else statement after an elif statement with a true condition, the else statement's block of code is not executed. For every set of if-elif-else statements, one and only one block will be executed. You can think of the if and elif statements as saying, "If this condition is true, execute this first block of code. Or else if this next condition is true, then execute this next block of code."

Unlike a while statement, when the execution is done executing the code in the block, it just keeps going down instead of returning to the start of the block.

The len() Function

The len() function is passed a string and returns the length of the string as an integer. For example, type the following into the interactive shell:

```
>>> len('hello')
5
```

```
>>> len('')
0
>>> len('hello') > 10
False
>>>
```

Indexes and Slices

A string slice is string that is made from a bigger string. You can pick out a substring by using indexing and slicing. A substring is just a name for a smaller string that is inside another string. Type the following into the interactive shell:

```
>>> spam = 'Hello world!'
>>> spam[0]
'H'
>>> spam[1]
'e'
>>> eggs = spam[6]
>>> print(eggs)
'W'
```

By adding some square brackets and a number (called the index) after a string value, you can pick out a single character from the string. The number that goes in between the brackets is called the index. Like a function call, the index is part of an expression and evaluates to a value. spam[0] will evaluate to a string value that has the first character in the string in spam: 'H'. spam[1] will evaluate to the second character: 'e'. Notice that the indexes begin as 0 for the first character, not at 1.

If the index is equal to or greater than the length of the string, then an error message will appear:

```
>>> 'Hello'[9999]
Traceback (most recent call last):
  File "<stdin>", line 1, in <module>
IndexError: string index out of range
>>>
```

There is a neat trick where you can enter a negative number as the index to count from the end of the string instead of the beginning. So the index -1 is the last character, -2 is the second to last character, and so on. Try it out in the interactive shell:

```
>>> spam = 'Hello'
>>> spam[-1]
```

```
'o'
>>> spam[-4]
'e'
>>>
```

To get more than one character at a time, you use slicing, which looks similar to indexing. Type the following into the interactive shell:

```
>>> spam = 'Hello world!'
>>> spam[0:2]
'He'
>>> spam[6:12]
'World!'
>>>
```

The first number is the index of where the substring the slicing evaluates to should start. The substring goes up to, but not including, the second index. This is why spam[0:5] evaluates to 'He' instead of 'Hel'.

If you leave out the first number from the slice, Python will automatically use the start of the string:

```
>>> spam = 'Hello World!'
>>> spam[:10]
'Hello Worl'
```

If you leave out the second number from the slice, Python will automatically use the end of the string:

```
>>> spam = ' Hello World!'
>>> spam[7:]
'orld!'
```

Return Values and the return Statement

A function call evaluates to a value. The value that a function call evaluates to is called a return value. To choose the value that is returned by a function we create with a def statement, we must use a return statement. A return statement is simply the return keyword followed by a value or expression which will evaluate to the return value. After executing the return statement, the program execution returns to the line of code that called the function. Consider the following short program:

```
1. def spam():
2.     print('Spam!')
3.     return 42
4.     print('Eggs!')
5. print(10 + spam())
```

When you run this program, it prints this to the screen:

```
Spam!
52
```

The program starts executing at the top (line 1). This is a def statement, which only defines the function's code. The program continues past the function's block to line 5, which is a print() call. The value that will be printed to the screen is the value that the expression 10 + spam() evaluates to. In order to evaluate this expression, we must call the spam() function that we defined on line 1.

The program execution jumps inside of the spam() function to line 2, which makes the string 'Spam!' appear on the screen. The next line is the return statement on line 3. This return statement makes the function call's return value the integer 42. Program execution now returns back to line 5, and the computer evaluates the 10 + spam() expression as 10 + 42, which further evaluates to 52. This 52 value is then printed to the screen.

Notice that line 4 never executes and 'Eggs!' never appears on the screen when spam() is called. This is because the program execution will always leave the function before it reaches line 4.

The in Operator

The in operator makes it easy to see if one string value is inside another string or not. Expressions that use the in operator return a Boolean value: True if the string value is in the other string and False if the value is not. Try typing 'antelope' in 'mantelope' into the shell:

```
>>> 'antelope' in 'mantelope'
True
>>>
```

The expression 'antelope' in 'mantelope' returns True because the string 'antelope' can be found as a substring in the string 'mantelope'. But if we type the expression 'antelope' in 'manwich', then this expression will return False.

```
>>> 'antelope' in 'manwich'
```

```
False
>>>
```

A blank string will always be found in any string value:

```
>>> '' in 'shark sandwich'
True
>>> '' in ''
True
>>>
```

Methods

Methods are just like functions, but they are always attached to a value. For example, all string values have a lower() method, which returns a copy of the string value in lowercase. You cannot just call lower() by itself and you do not pass a string argument to lower() by itself (as in lower('Hello')).

The lower() and upper() String Methods

Try entering 'Hello world!'.lower() into the interactive shell to see an example of this method. There is also an upper() method for strings, which changes all the characters in a string to uppercase. Try entering 'Hello world'.upper() into the shell:

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

Because the upper() method returns a string, 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 that string's lower() method. 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!'
>>>
```

Remember, if a string is stored in a variable, you can call a string method on that variable. Look at this example:

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

Notice that the lower() and upper() methods don't change the value of the string they were called on.

The isupper() and islower() String Methods

The isupper() and islower() string methods (which are on line 36 and 41) work in a way that is very similar to the isdigit() and isalpha() methods. isupper() will return True if the string it is called on contains at least one uppercase letter and no lowercase letters. islower() returns True if the string it is called on contains at least one lowercase letter and no uppercase letters. Otherwise these methods return False. The existence of non-letter characters like numbers and spaces does not affect the outcome. Although strings that do not have any letters, including blank strings, will also return False. Try typing the following into the interactive shell:

```
>>> 'HELLO'.isupper()
True
>>> 'hello'.isupper()
False
>>> 'hello'.islower()
True
>>> 'Hello'.islower()
False
>>> 'LOOK OUT BEHIND YOU!'.isupper()
True
>>> '42'.isupper()
False
>>> '42'.islower()
False
>>> ''.isupper()
```

```
False
>>> ''.islower()
False
>>>
```

The find() String Method

Using the in operator, you can see if a string exists in another string but you would not be able to tell where in the string it was. The find() method returns the index of the first location of a substring in another string as an integer value (or returns -1 if it is not found.) Try typing the following into the interactive shell:

```
>>> 'Hello'.find('H')
0
>>> 'Hello'.find('e')
1
>>> 'Hello'.find('ello')
1
>>> 'Hello'.find('l')
2
>>> 'Hello'.find('moose')
-1
>>> 'Hello'.find('ello world!')
-1
>>> 'Hello'.find('h')
-1
>>>
```

'Hello'.find('H') evaluates to the integer 0 because the substring 'H' is found at the first character in the string 'Hello'. Remember that the first index in a string begins with 0, not with 1. 'Hello'.find('e') returns the integer 1 because 'e' is found in the second position in 'Hello'.

You can specify any string to look for, not just single-character strings. 'Hello'.find('ello') also returns 1 because find() will return the index where the substring begins. Remember that the find() method will also return the index of where the substring is first found in the string. So even though 'l' appears twice in 'Hello', 'Hello'.find('l') will evaluate to 2 because that is the index where it is first found.

If the substring is not found in the string, than the find() method returns -1. The entire substring must be found in the string, so 'Hello'.find('ello world!') returns -1 because the even though “ello” is found in the string 'Hello', the “ world!” part is not in 'Hello'. Also, the case of the substring must match. This is why 'Hello'.find('H') returns 0 but 'Hello'.find('h') returns -1.

Converting Between Integers and Strings with int() and str()

The expression 10 == '10' returns False, because values of different data types are never equal. We will either have to convert the string to an integer or the integer to a string. We can do this with the int() and str() functons, respectively. The return value of the int() function is the integer form of the string that was passed to it. The return value of the str() function is the string form of the integer that was passed to it. Try the following in the interactive shell:

```
>>> int('10')
10
>>> str(10)
'10'
>>> int('6') + 5
11
>>> str(20) + '11'
2011
>>> int('10') == 10
True
>>> str(10) == '10'
True
>>>
```

Notice that the expression int('6') + 5 works just fine. You cannot add a string and an integer value together. But int('6') evaluates to the integer value 6, and this integer is added to the integer 5 to return the integer value 11.

The expression str(20) + '11' evauates to the string '2011'. This is because str(20) returns the string value '20' and '11' is a string value. Remember that when you add two string values together, they are concatenated (even if they are strings that look like integers, they are still strings.)

Note that if you try to pass a string to int() that is not in the form of an integer, Python will send an error and cause the program to crash:

```
>>> int('forty two')
Traceback (most recent call last):
  File "<pyshell#10>", line 1, in <module>
    int('forty two')
ValueError: invalid literal for int() with base 10: 'forty two'
>>>
```

The string.ascii_uppercase and string.ascii_lowercase Strings

The string module has a couple of variables in it called ascii_uppercase and ascii_lowercase. The values in these variables are 'ABCDEFGHIJKLMNOPQRSTUVWXYZ' and 'abcdefghijklmnopqrstuvwxyz' respectively.

We will be using these strings in our program. We could have just made an assignment statement and typed out these strings ourselves. But since they already exist in the string module, it is easier to import the module and use the variables there. It also reduces the chance that we accidentally mistype the alphabet, which would cause bugs in our program.

String Interpolation with %s

Normally, if you want to use the string values inside variables in another string, you have to use the + concatenation operator:

```
>>> name = 'Alice'
>>> event = 'party'
>>> where = 'the pool'
>>> day = 'Saturday'
>>> time = '6:00pm'
>>> print('Hello, ' + name + '. Will you go to the ' + event + ' at ' + where +
' this ' + day + ' at ' + time + '?')
Hello, Alice. Will you go to the party at the pool this Saturday at 6:00pm?
>>>
```

As you can see, it can be very hard to type a line that concatenates several strings together. Instead, you can use string interpolation, which lets you put placeholders like %s (these placeholders are called conversion specifiers), and then put all the variable names at the end. Each %s is replaced with the value in the variable at the end of the line. For example, the following code does the same thing as the above code:

```
>>> name = 'Alice'
>>> event = 'party'
>>> where = 'the pool'
>>> day = 'Saturday'
>>> time = '6:00pm'
>>> print('Hello, %s. Will you go to the %s at %s this %s at %s?' % (name,
event, where, day, time))
Hello, Alice. Will you go to the party at the pool this Saturday at 6:00pm?
>>>
```

String interpolation can make your code much easier to type and read, rather than using several + concatenation operators.

The final line has the print() call with a string with conversion specifiers, followed by the % sign, followed by a set of parentheses with the variables in them. The first variable name will be used for the first %s, the second variable with the second %s and so on. The Python interpreter will give you an error if you do not have the same number of %s conversion specifiers as you have variables.

Another benefit of using string interpolation instead of string concatenation is that interpolation works with any data type, not just strings. All values are automatically converted to the string data type. (This is what the s in %s stands for.) If you typed this code into the shell, you'd get an error:

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

You get this error because string concatenation can only combine two strings, and spam is an integer. You would have to remember to put str(spam) in there instead. But with string interpolation, you can have any data type. Try entering this into the shell:

```
>>> spam = 42
>>> print('Spam == %s' % (spam))
Spam == 42
>>>
```

As you can see, using string interpolation instead of string concatenation is much easier because you don't have to worry about the data type of the variable. Also, string interpolation can be done on any strings, not just strings used in print() function calls.

String interpolation is also known as string formatting.

The find() Method

Sample Run of the Caesar Cipher Program

Here is what the Caesar Cipher program looks like when you run it.

```
Caesar Cipher
e - Encrypt Message
d - Decrypt Message
k - Generate Key
c - Copy "" to clipboard
q - Quit
> k
24


Caesar Cipher
e - Encrypt Message
d - Decrypt Message
k - Generate Key
c - Copy "" to clipboard
q - Quit
> e
Enter your message:
> Hello world! This is my secret message.
Enter the key number (0-25)
> 24
Your translated text is:
Fcjjm umpjb! Rfgq gq kw qcapcr kcqqyec.

Caesar Cipher
e - Encrypt Message
d - Decrypt Message
k - Generate Key
c - Copy "Fcjjm umpj..." to clipboard
q - Quit
> c

Caesar Cipher
e - Encrypt Message
d - Decrypt Message
k - Generate Key
c - Copy "Fcjjm umpj..." to clipboard
q - Quit
> d
Enter your message:
Fcjjm umpjb! Rfgq gq kw qcapcr kcqqyec.
Enter the key number (0-25)
> 24
Your translated text is:
Hello world! This is my secret message.
```

```
Caesar Cipher
e - Encrypt Message
d - Decrypt Message
k - Generate Key
c - Copy "Hello worl..." to clipboard
q - Quit
> q
```

Source Code of the Caesar Cipher

Type in the following code into the file editor, and then save it as caesar.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 caesar.py file. You can download this file from http://becomeacodebreaker.com/pyperclip.py

```
 1. # Caesar Cipher, http://becomeacodebreaker.com
 2.
 3. import random, pyperclip, string
 4.
 5. def main():
 6.     # main() is called from the very bottom of this code
 7.     lastMessage = ''
 8.     mode = ''
 9.     while mode != 'q':
10.         # main loop
11.         print()
12.         print('Caesar Cipher')
13.         mode = getMode(lastMessage)
14.         if mode == 'k':
15.             # Generate a random key
16.             print(random.randint(1, 25))
17.             print()
18.         elif mode == 'c':
19.             # Copy the last message to clipboard.
20.             pyperclip.copy(lastMessage)
21.         elif mode == 'e' or mode == 'd':
22.             # Encryption & decryption
23.             message = getMessage()
24.             key = getKey()
25.
26.             print('Your translated text is:')
27.             translated = getTranslatedMessage(mode, message, key)
28.             print(translated)
29.             lastMessage = translated
```

```
30.
31.
32. def getMode(clipboardText):
33.     # This function returns 'e', 'd', 'k', 'c' or 'q',
34.     # depending on what the user enters.
35.     if len(clipboardText) > 10:
36.         # Truncate the string in clipboardText if it is too long.
37.         clipboardText = clipboardText[:10] + '...'
38.     mode = 'x'
39.     while mode not in 'edkcq':
40.         # Keep looping until a valid letter is typed in.
41.         print('e - Encrypt Message')
42.         print('d - Decrypt Message')
43.         print('k - Generate Key')
44.         print('c - Copy "%s" to clipboard' % (clipboardText))
45.         print('q - Quit')
46.         mode = input('> ').lower()
47.     return mode
48.
49.
50. def getMessage():
51.     # This function returns the message the user types in.
52.     print('Enter your message:')
53.     return input('> ')
54.
55.
56. def getKey():
57.     # This function returns the key that the user typed in.
58.     # For the Caesar Cipher, this must be between 0 and 25.
59.     while True:
60.         print('Enter the key number (0-25)')
61.         key = int(input('> '))
62.         if (key >= 0 and key <= 25):
63.             return key
64.
65.
66. def getTranslatedMessage(mode, message, key):
67.     # This function returns the encrypted/decrypted form of
68.     # the message passed to it, using the key passed.
69.     if mode == 'd':
70.         # make the key negative for decryption
71.         key = -key
72.     translated = ''
73.
74.     # Loop through each symbol in the message.
75.     i = 0
```

```
    while i < len(message):
        symbol = message[i]
        num = string.ascii_uppercase.find(symbol.upper())
        if num != -1:
            # the symbol is a letter
            num = num + key
            if num > 25:
                num = num - 26
            elif num < 0:
                num = num + 26

            # Add the translated symbol to the end of the translated
message.
            if symbol.islower():
                translated = translated + string.ascii_lowercase[num]
            else:
                translated = translated + string.ascii_uppercase[num]
        else:
            # The symbol is not a letter, just add it "as is".
            translated = translated + symbol
        i = i + 1
    return translated


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

Checking Your Source Code with the Online Diff Tool

Even though it's over a hundred lines of code, I recommend typing in this program yourself. It will help give you a better idea of what code is in this program. You might make some mistakes while typing it in yourself though. 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://becomeacodebreaker.com/diff/caesar/ 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.

Usually any problems are just a simple typo such as the wrong amount of indentation or a missing colon character. However, if you see an error message like this after typing something into your program:

```
NameError: name 'e' is not defined
```

The most likely cause is that you are using Python 2 instead of Python 3. Scroll to the top of the interactive shell window to see the text that appeared when you first started IDLE. The Python version should be listed there. If it says something like Python 2.6.6, then you are running Python 2 rather than Python 3. Your computer can have Python 2 and Python 3 installed at the same time, so be sure you ran the Python 3 version of IDLE instead of the Python 2 version.

Remember that before you can run this program you must download the pyperclip.py module (which does not come with Python.) It also must be in the same directory as caesar.py. Otherwise, you see this error message:

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

Tracing Through the Program Online

If you want help understanding how exactly Python executes this program, the webpage http://becomeacodebreaker.com/traces has an app that will show step by step what each line of code for this program does. This is a helpful way to see what the computer is doing after each line of code.

An Overview of the Code

Before going into a line by line explanation, let's describe what the each function of the code generally does.

- main() – This function will be called when the program first starts running. This function will call all the other functions in our program. It has a loop that asks the user for a command and then carry it out. This loop will keep carrying out the user's commands again and again until they enter 'q' to quit. Then the function returns.
- getMode() – This function will ask the user if they want to encrypt, decrypt, generate a random key, or copy the last message that was encrypted or decrypted to the clipboard. The code makes sure the user enters a proper command, and then returns a string of this command such as 'e' for encrypt or 'q' for quit.
- getMessage() – This function asks the user for the message they want to encrypt or decrypt, and then returns what the user typed in.
- getKey() – This function asks the user to enter the encryption/decryption key, and makes sure that the user entered a valid key (that is, an integer between 0 and 25).
- getTranslatedMessage() – This function has the code that implements all the encryption and decryption. It takes arguments for whether to encrypt or decrypt, the message to encrypt or decrypt, and the key that will be used.

How the Program Works

This chapter has covered quite a few programming concepts. Now you can see these concepts being used in an actual program. Here we will explain what each of the lines of code in this program does.

```
1. # Caesar Cipher, http://becomeacodebreaker.com
2.
3. import random, pyperclip, string
```

The first line of code is just a comment that reminds the programmer what this program is. Remember that the Python interpreter ignores any text in your code if it is a comment (that is, it follows the # pound sign.)

Line 3 imports three modules that our program will use. The random module has the function randint() which we will use to generate random encryption keys. The pyperclip module has functions for setting the computer's clipboard. And the string module contains the ascii_uppercase and ascii_lowercase strings that this program will use.

The main() Function Trick

```
5. def main():
6.     # main() is called from the very bottom of this code
7.     lastMessage = ''
```

The main part of our program will go inside a function we make called main(). This function is called at the very end of the program on line 105. The first line in the main() function is a comment reminding us where the main() function is called from.

The reason we put this code inside of a function has to do with being able to use the caesar.py program as a module we import into other programs. This will be explained in the next chapter. For now, just know that the main part of our code (which will call the other functions in this program) will be in the main() function.

Line 7 creates a new variable called lastMessage that contains a blank string. This variable will be used to store the last message that the user encrypted or decrypted. Since at the start of the program the user hasn't encrypted or decrypted anything, we will just set this variable to a blank string.

```
8.     mode = ''
9.     while mode != 'q':
10.        # main loop
```

```
11.         print()
12.         print('Caesar Cipher')
13.         mode = getMode(lastMessage)
```

Now we want code that will get what mode the user wants to set the program in (encrypt, decrypt, copy the last results to the clipboard, or generate a new key). On line 9 we enter a loop that will keep asking the user what mode they want, performing operations, and then repeating until the user enters 'q' for the mode (which means they want to quit.)

Line 8 sets the mode variable to a blank string, because the mode variable must exist before it can be used in the while statement's condition. It also ensures that the condition is True the first time the while statement is executed, so that execution enters the loop on line 10.

The getMode() function is a function that we will define later. It is passed the last encrypted or decrypted message (stored in lastMessage) so that getMode() can store it in the clipboard if the player wants to. The getMode() function returns a single character string depending on what the player wants to do. It returns 'q' if the player wants to quit, 'e' for encrypting, 'd' for decrypting, 'k' for generating a random key, and 'c' for copying the last message to the clipboard.

We could have getMode() return other strings instead of just the single character strings, such as 'quit' instead of 'q' and 'copy' or 'clipboard' instead of 'c'. We would have to change the code in the main() function also (line 9 would have to be while mode != 'quit'). I just have the personal preference for short strings because it is less likely to mistype them. If getMode() returned 'copy' but main() was looking for 'clipboard', then this program would have a bug. But 'copy' or 'clipboard' are both more descriptive than just 'c', so use whichever style you prefer in your programs.

The Importance of Randomness in Cryptography

```
14.         if mode == 'k':
15.             # Generate a random key
16.             print(random.randint(1, 25))
17.             print()
```

If getMode() returned the string value 'k', then the user wants the program to generate a random key. We will call the random.randint() function to return a random integer value between 1 and 25, and then print it to the screen. We should learn why randomly selected keys are important.

The secret messages you encrypt with the Caesar Cipher will only be secret if you are the only person who knows the key. If you tell anyone else the key, they will be able to decrypt your ciphertext and read the message. Maybe you want to do this so you can share secret messages

with other people. Maybe you don't want anyone else to know the key so that you can write down messages for yourself that nobody else can read.

The key for a Caesar Cipher ciphertext is one of the integers from 0 to 25. Let's say Alice wants to encrypt a message, and her lucky number is 7. To make it easy to remember which key she uses, Alice chooses to encrypt her message with the number 7 as the key. Alice's friend Eve does not know the key, but she knows that Alice's lucky number is 7, so after getting a hold of the Alice's ciphertext Eve tries to decrypt it with the key 7. Eve could then read Alice's message!

For Alice's next encrypted message, she realizes that since Eve knows her lucky number is 7, she should use a key other than 7. But this is a mistake too! Eve could probably guess that whatever Alice's key is, she probably didn't use the key 7 again. This isn't as bad as knowing what the key is, but it gives Eve a hint as to what the key is not. That little bit of information can help Eve break Alice's ciphertext.

The problem is that there is bias in which key Alice decided to use for her messages. A bias is the preference for or against something. A key is only strong if it was randomly selected, because a randomly selected number has no bias. A non-randomly selected key is predictable, which can help a code breaker predict what the key is. (We will learn how to predict keys and break ciphers in later chapters in this book.)

One way we can have random numbers is by letting the computer choose the number for us. Since the user doesn't have any say in what the key is, then the user can't be biased at all. This is what the our call to random.randint() is for. (This reasoning is the same reason why you don't want to use English words for your online passwords. Read http://becomeacodebreaker.com/moreinfo for a detailed explanation.)

```
18.         elif mode == 'c':
19.             # Copy the last message to clipboard.
20.             pyperclip.copy(lastMessage)
```

If mode was set to the string value 'c' (and the previous if statement's condition was False), then the programwill call the pyperclip.copy() function to copy the last encrypted or decrypted message (stored in lastMessage) to the clipboard. Of course, this only makes sense to do when lastMessage is not blank.

```
21.         elif mode == 'e' or mode == 'd':
22.             # Encryption & decryption
23.             message = getMessage()
24.             key = getKey()
25.
26.             print('Your translated text is:')
```

```
27.             translated = getTranslatedMessage(mode, message, key)
28.             print(translated)
29.             lastMessage = translated
```

Encrypting and decrypting are both handled by our getTranslatedMessage() function. Either way, we will want to call getMessage() and then getKey() to let the user type in the message and key they want to use. The returned values from these calls are stored in message and key, which are then passed to the getTranslatedMessage() function. The return value of that function call is the string of encrypted (if mode was 'e') or decrypted (if mode was 'd') text. We will print this text to the screen with the print() call on line 28. On line 29 we will store the text in the lastMessage variable in case the user wants to copy the text to the clipboard.

The Importance of Input Validation

```
32. def getMode(clipboardText):
33.     # This function returns 'e', 'd', 'k', 'c' or 'q',
34.     # depending on what the user enters.
35.     if len(clipboardText) > 10:
36.         # Truncate the string in clipboardText if it is too long.
37.         clipboardText = clipboardText[:10] + '...'
38.     mode = 'x'
39.     while mode not in 'edkcq':
40.         # Keep looping until a valid letter is typed in.
41.         print('e - Encrypt Message')
42.         print('d - Decrypt Message')
43.         print('k - Generate Key')
44.         print('c - Copy "%s" to clipboard' % (clipboardText))
45.         print('q - Quit')
46.         mode = input('> ').lower()
47.     return mode
```

The getMode() function is from line 32 to line 47. This function prints some text to the screen offering the user several choices about what the program should do, and then lets the user type in a command to the program. But more importantly, this function's code makes sure that what the user typed in was a valid command. If it was not a valid command, the program will ask the user again. The program will keep asking the user until a valid command is entered, and then the function returns this command as the function's return value. This guarantees that the command returned from the getMode() function will always be a valid one.

If we just had a simple call to input() with nothing else, then the user could enter a string that our program wouldn't expect. This could cause a bug in our program, so it's a good idea to have the program be able to handle bad input from the user.

```
50. def getMessage():
51.     # This function returns the message the user types in.
52.     print('Enter your message:')
53.     return input('> ')
```

The getMessage() function is a pretty simple function. It just tells the user to enter the message they want to encrypt or decrypt with the print() call on line 52, and then returns what they typed in. This function doesn't have any input validation code because there aren't any restrictions on the message text.

```
56. def getKey():
57.     # This function returns the key that the user typed in.
58.     # For the Caesar Cipher, this must be between 0 and 25.
59.     while True:
60.         print('Enter the key number (0-25)')
61.         key = int(input('> '))
62.         if (key >= 0 and key <= 25):
63.             return key
```

The getKey() function asks the user to enter the encryption or decryption key they want to use. For the Caesar Cipher, this must be an integer between 0 and 25. (Even though the encryption key 0 will produce a ciphertext that is the same as the plaintext, we want to give the user this option anyway.) The while statement on line 59 simply has the True value as its condition. This means that once the program execution enters the loop, if will keep looping because the condition will always evaluate to True. Ordinarily, this would cause an infinite loop bug. But we have a return statement on line 63 which will exit the function (and the loop) when the user types in a key that is between 0 and 25.

Remember that the input() function will return a string, not an integer. This is why we also call the int() function on the return value from input(). The int() function will return an integer form of the number that the user typed in and store this in a variable named key.

Notice that the if statement on line 62 uses the >= and <= operator, which means "greater than or equal" to 0 and "less than or equal" to 25.

So if the user enters the number 100, then the if statement's condition on line 62 would evaluate to False (the key would be greater than or equal to 0, but it would not be less than or equal to 25). Because there is no more code in the while statement's block, the execution jumps back to the while statement and the loop's code runs again to re-ask the user to enter a key.

This ensures that the getKey() function will always return an integer between 0 and 25.

```
66. def getTranslatedMessage(mode, message, key):
67.     # This function returns the encrypted/decrypted form of
68.     # the message passed to it, using the key passed.
69.     if mode == 'd':
70.         # make the key negative for decryption
71.         key = -key
72.     translated = ''
```

The getTranslatedMessage() function does all of the encrypting and decrypting work. Encryption and decryption are both fairly similar in the Caesar Cipher, so we can have one function do both. Remember that encrypting involves adding the key whereas decrypting involves subtracting the key. But adding a negative number is the same as subtraction, so if we change the key to its negative form (which is what line 71 does), then we can have the code just add the key and it will perform the correct operation for both encrypting and decrypting.

We will store the encrypted or decrypted (we use the word "translated" to mean either) text in the variable translated. As we translate more of the message, we will add it to this string.

```
74.     # Loop through each symbol in the message.
75.     i = 0
76.     while i < len(message):
77.         symbol = message[i]
```

The translation process must be done to each individual letter in the string in the message variable, so we will set up a loop to do this translation over and over. We will first store the integer 0 in the variable i. This integer in i will point to the index of the current letter in message that is being translated. After we are done translating this letter, we will increment i to point to the next index.

Just as a shortcut so that we don't type message[i] over and over again, we will store this letter in a variable named symbol, which is much easier to understand when we read the code.

```
78.         num = string.ascii_uppercase.find(symbol.upper())
79.         if num != -1:
80.             # the symbol is a letter
81.             num = num + key
82.             if num > 25:
83.                 num = num - 26
84.             elif num < 0:
85.                 num = num + 26
```

The first thing we need to do in the Caesar Cipher is determine the number of the letter we are trying to translate. That is, 0 for "A", 1 for "B", and so on. The index of the letters in the string

value in string.ascii_uppercase and string.ascii_lowercase matches this perfectly. We can use the find() method to locate the index of the symbol we are translating.

Of course, the symbol could be lowercase or uppercase. To handle both cases, we will search the string in string.ascii_uppercase for the substring symbol.upper(). The symbol.upper() expression will evaluate to the uppercase version of symbol, no matter if symbol is lowercase or already uppercase. We store the return value of the find() method call in the variable num.

If num does not equal -1, then we know that symbol.upper() was found in string.ascii_uppercase. This means that symbol must be a letter (since string.ascii_uppercase only has letters in it) and not a number or punctuation mark. So if the condition on line 79 is True, that means we should go ahead and translate the symbol.

Line 81 does the encryption or decryption. Remember that if we are decrypting, the key has been set to a negative number, so the expression num + key will actually subtracts the original key from num. The integer value in the num variable is updated to the translated form on line 81.

Line 82 and 84 checks for the special case where our calculated number is larger than 25 (in which case, we want to subtract 26 as we do on line 83) or smaller than 0 (in which case, we want to add 26 as we do on line 85.)

```
 87.             # Add the translated symbol to the end of the translated
message.
 88.             if symbol.islower():
 89.                 translated = translated + string.ascii_lowercase[num]
 90.             else:
 91.                 translated = translated + string.ascii_uppercase[num]
```

Now that the num variable stores the index of the translated letter, we need to find the letter itself and add it to the end of the string in translated. If the original symbol was lowercase, then we want to use the lowercase version of the translated symbol. If the original symbol was uppercase, then we want to use the uppercase version of the translated symbol. The string.ascii_lowercase and string.ascii_uppercase have all the letters in the same indexes, so we will use string.ascii_lowercase for the lowercase translated symbols and string.ascii_uppercase for the uppercase translated symbol.

We want to keep the same casing in the translated message as in the original message. That way, when we decrypt some ciphertext, the decrypted plaintext will be in the same case as the original plaintext before encryption.

```
 92.         else:
 93.             # The symbol is not a letter, just add it "as is".
```

```
94.             translated = translated + symbol
```

The else statement on line 92 is for the if statement on line 79. Line 79's if statement checked that the symbol was a letter. If that if statement's condition is False, then the code in the else statement's block will be executed. This code will handle the case where the symbol is a number or space or punctuation character. The Caesar Cipher will not encrypt or decrypt these symbols, so line 94 just adds the symbol to the end of the translated string.

```
95.         i = i + 1
96.     return translated
```

Now that we are done translating the symbol, we want the integer in the i variable to point to the next index, so we add 1 to it. Line 95 is the end of the while loop that was started on line 76 (you can tell because line 96's indentation is less than line 95's indentation), so the program execution jumps back to line 76 and re-checks the while statement's condition.

The while statement's condition was i < len(message). If the integer in i has increased to the length of message, than we have finished translating the entire message. (Remember that the index of the last character in a string is one less than the length of the string because the indexing starts at 0, not 1.)

```
99. if __name__ == '__main__':
100.     main()
```

After the import statement on line 3, the if statement on line 99 is executed because it is one of the few lines of code not in a function. When the program is run, the import statement is executed, and then the def statements for each of the functions in the program. But the code inside the functions is not executed, so execution keeps skipping down until it reaches line 99.

The reason we have this if statement is related to why we have a main() function and will be explained more in the next chapter. For now, you just need to know that the __name__ variable (with two underscore characters in front and two underscore characters in back of "name") is a special variable in Python that is set to the string '__main__' (again, with two underscore characters in front and in back of "main").

We expect this condition to be true, so the if statement's block is executed and the main() function is called, which runs the rest of the code in our program.

A Really Stupid Mistake

Remember that anyone with the key can read an encrypted message. The purpose to encrypting messages is so that even if someone finds it, they would not be able to read it. So if you email or print out an encrypted message to send to your friend, your friend needs to know the key you used to encrypt it.

A really stupid mistake is to send the key with the encrypted message, like this:

```
Hello Cal, the key I used to encrypt this message is 17: Kyv jvtivk grjjnfiu
wfi dp feczev rttflek zj Ifjvslu.
```

If somebody finds this message, the encryption is worthless because they now know the key and can decrypt it.

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 the Caesar Cipher. And more importantly, you can understand how this code works.

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, and in the next chapter we will use it to break the Caesar Cipher so we can read ciphertext that other people encrypted. Normally this requires too much work for a human being to do. But the computer, given the proper program, can do it easily. Now let's move on to the next chapter, and learn how to become a code breaker.

Chapter 7 – Breaking the Caesar Cipher with the Brute Force Technique

Breaking Ciphers

Now we will write a program to break the Caesar Cipher, using a cryptanalytic technique called brute force. You shouldn't use the Caesar Cipher to keep your information secret. It is very easy for someone to decrypt the ciphertext, even if they do not have the encryption key. Ideally, nobody else would ever even see the encrypted text. But for every cipher out there, you must assume that everyone has access to the encrypted text. There is a law in cryptography called Kerckhoff's Principle (named after the19th century cryptographer Auguste Kerckhoffs) which 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."

This chapter introduces a few new programming concepts. But there aren't as many as in the last chapter, so this chapter will teach them while explaining how the Caesar Cipher breaking program works.

Brute Force

The Caesar Cipher has only 25 different keys to choose from. This makes it a less than 4% chance that someone could guess what the key is. But according to Kerckhoff's Principle, we should assume that other people have the encrypted ciphertext. A cryptanalyst can guess one key, start to decrypt it, and if it is not the correct key, then move on to the next key.

There are only 25 different possible keys. It would be easy to write a program that quickly decrypts the ciphertext with every possible key, and then let the cryptanalyst browse the decryptions to find the plaintext. This technique is called brute force. It isn't a very smart technique, but through sheer effort, the Caesar Cipher can be broken.

Sample Run of the Caesar Cipher Breaker Program

Here is what the Caesar Cipher program looks like when you run it. Notice that the decrypted output for key 5 is plain English, so the original encryption key must have been 5.

```
Caesar Cipher Breaker

Enter the message to break:
Enter your message:
> Fsdymnsl ymfy nx ytt xyzuni yt gj xutpjs nx xzsl.
Breaking...
Key 0: Fsdymnsl ymfy nx ytt xyzuni yt gj xutpjs nx xzsl.
Key 1: Ercxlmrk xlex mw xss wxytmh xs fi wtsoir mw wyrk.
Key 2: Dqbwklqj wkdw lv wrr vwxslg wr eh vsrnhq lv vxqj.
Key 3: Cpavjkpi vjcv ku vqq uvwrkf vq dg urqmgp ku uwpi.
Key 4: Bozuijoh uibu jt upp tuvqje up cf tqplfo jt tvoh.
Key 5: Anything that is too stupid to be spoken is sung.
Key 6: Zmxsghmf sgzs hr snn rstohc sn ad ronjdm hr rtmf.
Key 7: Ylwrfgle rfyr gq rmm qrsngb rm zc qnmicl gq qsle.
Key 8: Xkvqefkd qexq fp qll pqrmfa ql yb pmlhbk fp prkd.
Key 9: Wjupdejc pdwp eo pkk opqlez pk xa olkgaj eo oqjc.
Key 10: Vitocdib ocvo dn ojj nopkdy oj wz nkjfzi dn npib.
Key 11: Uhsnbcha nbun cm nii mnojcx ni vy mjieyh cm moha.
Key 12: Tgrmabgz matm bl mhh lmnibw mh ux lihdxg bl lngz.
Key 13: Sfqlzafy lzsl ak lgg klmhav lg tw khgcwf ak kmfy.
Key 14: Repkyzex kyrk zj kff jklgzu kf sv jgfbve zj jlex.
Key 15: Qdojxydw jxqj yi jee ijkfyt je ru ifeaud yi ikdw.
Key 16: Pcniwxcv iwpi xh idd hijexs id qt hedztc xh hjcv.
Key 17: Obmhvwbu hvoh wg hcc ghidwr hc ps gdcysb wg gibu.
Key 18: Nalguvat gung vf gbb fghcvq gb or fcbxra vf fhat.
Key 19: Mzkftuzs ftmf ue faa efgbup fa nq ebawqz ue egzs.
Key 20: Lyjestyr esle td ezz defato ez mp dazvpy td dfyr.
Key 21: Kxidrsxq drkd sc dyy cdezsn dy lo czyuox sc cexq.
Key 22: Jwhcqrwp cqjc rb cxx bcdyrm cx kn byxtnw rb bdwp.
Key 23: Ivgbpqvo bpib qa bww abcxql bw jm axwsmv qa acvo.
Key 24: Hufaopun aoha pz avv zabwpk av il zwvrlu pz zbun.
Key 25: Gteznotm zngz oy zuu yzavoj zu hk yvuqkt oy yatm.
Which key to copy to clipboard? (Enter 0-25, or q to quit.)
> 5
```

Source Code of the Caesar Cipher Breaker

Type in the following code into the file editor, and then save it as caesar.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 caesar.py file. You can download this file from http://becomeacodebreaker.com/pyperclip.py

```
1. # Caesar Cipher Breaker, http://becomeacodebreaker.com
2.
```

```
 3. import caesar, pyperclip
 4.
 5. def main():
 6.     print('Caesar Cipher Breaker')
 7.     print()
 8.     print('Enter the message to break:')
 9.     message = caesar.getMessage()
10.
11.     print('Breaking...')
12.     key = 0
13.     while key < 26:
14.         # print the decrypted form of all possible keys
15.         print('Key ' + str(key) + ': ' + caesar.getTranslatedMessage('d',
message, key))
16.         key = key + 1
17.
18.     # copy one of the decrypted outputs to the clipboard
19.     print('Which key to copy to clipboard? (Enter 0-25, or q to quit.)')
20.     response = input('> ')
21.     if response.isdigit() and int(response) >= 0 and int(response) < 26:
22.         pyperclip.copy(caesar.getTranslatedMessage('d', message,
int(response)))
23.
24. if __name__ == '__main__':
25.     main()
```

Importing the Caesar Cipher Program

This program is much shorter than the original Caesar Cipher program. That is because it makes use of much of the code in the original caesar.py program. If we had to retype all of the functions we already made, then our Caesar Cipher Breaker program would be about three times its current size. Also, if we find a bug in our original Caesar Cipher program and fix it, it will automatically be fixed in the Caesar Cipher Breaker program as well.

In order to use these functions though, we first need to import the caesar.py program. In order to do this, the caesar.py file must be in the same folder as our caesarBreaker.py file. Otherwise you will get an ImportError error message when you run this program.

```
1. # Caesar Cipher Breaker, http://becomeacodebreaker.com
2.
3. import caesar, pyperclip
```

Our program also imports the pyperclip module.

What happens when we import a file is that the code in it is run, just like when we call a function. So when our caesarBreaker.py program imports caesar.py, all of the functions in that file are defined so that we can call them from caesarBreaker.py.

But why doesn't the Caesar Cipher program itself run? That's because when you run a program by loading it in IDLE and pressing F5, the __name__ variable is automatically set to the string value '__main__'. However, when a program is imported, the __name__ variable is automatically set to a string that contains the filename (without the file extension).

So if we run the Caesar Cipher program directly, the __name__ variable is set to '__main__' and the code at the end of the file will call the main() function which runs the rest of the program. However, when we import caesar.py the __name__ variable will be set to 'caesar' and the main() function will not be called. But all the functions will be defined so that we can use them in the program that imported the file.

```
 5. def main():
 6.     print('Caesar Cipher Breaker')
 7.     print()
 8.     print('Enter the message to break:')
 9.     message = caesar.getMessage()
```

The Caesar Cipher Breaker program's main function will call the getMessage() function that it imported from caesar.py by including the caesar prefix in front of the function name. This will return the message that the player types in.

```
11.     print('Breaking...')
12.     key = 0
13.     while key < 26:
14.         # print the decrypted form of all possible keys
15.         print('Key ' + str(key) + ': ' + caesar.getTranslatedMessage('d',
message, key))
16.         key = key + 1
```

This code will loop through every key by setting the key variable to each integer from 0 to 26 for each iteration through the for loop's block. It calls the getTranslatedMessage() function we made in the caesar.py program that implements the Caesar Cipher. The decryptions with the wrong key will look like garbage text. But one of them will eventually come out looking like normal English. Either way, we display the results on the screen with the print() call on line 15.

The isdigit() and isalpha() String Methods

The isdigit() and isalpha() string methods are very similar to the isupper() and islower() methods. The isdigit() method will return the Boolean value True if it is called on a string that has only numbers in it, and no letters or punctuation characters. The isalpha() method will return True if it is called on a string that has only letters in it, and no numbers or punctuation characters. If the strings are empty strings, then these methods will return False (just like isupper() and islower(). Try typing the following into the interactive shell:

```
>>> '123'.isdigit()
True
>>> 'forty two'.isdigit()
False
>>> '42.0'.isdigit()
False
>>> ''.isdigit()
False
>>> 'Hello'.isalpha()
True
>>> 'Hello!!!'.isalpha()
False
>>> 'Hello world'.isalpha()
False
>>> ''.isalpha()
False
>>>
```

Notice in that '42.0'.isdigit() returns False. Remember that isdigit() returns True if there are only number characters in the string. 42.0 is a number, but the decimal point causes isdigit() to return False.

Short-Circuit Evaluation

Imagine I told you to bring an umbrella if the weather report says it would rain *or* if it was raining outside. As code, this would look something like this:

```
if weatherReportSaysRain() or isRainingOutside():
    getUmbrella()
```

When you check the weather report, it says it will rain. Do you even have to check if it is raining outside before getting your umbrella? You don't, because you know that no matter if it is raining outside or not, you are still going to pack an umbrella because the weather report said it would rain.

How about if I told you that if it was lunchtime *and* today was Tuesday, you should go buy ice cream. As code, this might look like:

```
if isLunchtime() and todayIsTuesday():
    buyIceCream()
```

You check your watch, and it is not lunch time. Do you also have to check a calendar to see if it is Tuesday? You don't, because no matter if it is Tuesday or not, you will not be buying ice cream since it is not lunchtime.

Boolean expressions that use the and and or operators sometimes don't have to evaluate the entire expression before they know what value the expression will evaluate to. The and operator needs both Boolean values it connects to be True in order to return True. So if the first value is False, it does not need to check what the other value is before evaluating the entire expression as False. The same is true with the or operator. If the first value is True, then it doesn't matter what the other value is, the entire expression will evaluate to False.

This shortcut is called short-circuit evaluation. It is important to know how short-circuit evaluation works for how we write the conditions in our if and while statements. You remember that if we pass a string that is not a number to the int() function, Python will crash the program and give an error message:

```
>>> int('42')
42
>>> spam = 'hello'
>>> int(spam)
Traceback (most recent call last):
  File "<pyshell#43>", line 1, in <module>
    int(spam)
ValueError: invalid literal for int() with base 10: 'hello'
>>>
```

We want the program to be able to handle this without crashing. Notice that the following expression does not cause an error:

```
>>> spam = 'hello'
>>> spam.isdigit() and int(spam)
False
>>>
```

The reason is because spam.isdigit() returns False, so the Python interpreter does not bother calling the int() function. This is good, because if the value in spam is not a string of just number

characters, then the int(spam) code would cause an error. But short-circuit evaluation helpfully prevents this from ever happening.

```
18.     # copy one of the decrypted outputs to the clipboard
19.     print('Which key to copy to clipboard? (Enter 0-25, or q to quit.)')
20.     response = input('> ')
21.     if response.isdigit() and int(response) >= 0 and int(response) < 26:
22.         pyperclip.copy(caesar.getTranslatedMessage('d', message,
int(response)))
```

The user can now read the list of decrypted outputs until he finds the one decrypted with the correct key. The program lets the user type this key in so they can easily move the correct plaintext text to the clipboard. The if statement on line 21 performs some input validation to make sure the user typed in a number between 0 and 25, so that the int() call on line 22 does not cause an error due to a non-number string stored in the response variable.

If the user did not type in a number, the if statement is skipped. Since this is the last line of the main() method, the program execution returns to line 25 where it was originally called.

```
24. if __name__ == '__main__':
25.     main()
```

In the last chapter, we had these same two lines at the end of the Caesar Cipher program. Remember that the __name__ (with two underscore characters before and two underscore characters after "name") is a special variable in Python programs that is automatically set to the string value '__main__' when they are run.

However, when a Python program is imported by another Python program, the imported program's __name__ variable will be set to the filename of the program. So when we run caesarBreaker.py, it will import caesar.py with the import statement on line 3. When caesar.py is imported, the code in it is run just like when we run it from IDLE, except it sees the __name__ variable as containing the string 'caesar' (since the filename is caesar.py).

This is how we can have the program run itself when we directly run the program's source file, but just have it only run the def statements when we import it.

(The .py files must be in certain directories in order for them to be imported by other programs. If you are getting an error message when you execute the import statement, make sure that the file is in the same directory as the program doing the importing. You can learn more about the details of where the imported program can be here: http://becomeacodebreaker.com/moreinfo)

A Short Message on Hacking

It is really exciting to be able to break ciphers and read other people's secrets. But you should keep in mind that just because you can do something doesn't mean you should. Reading other people's secret messages is an invasion of their privacy. This is true if they have encrypted their messages, or just left their files as plaintext and easily accessible. In some cases, invading their privacy may also be illegal. But legal or not, and encrypted or not, you should not read other people's emails, diaries, or files if you know they would not want you to.

Many computer hackers just like gaining the knowledge of how cryptography systems work, and what their flaws are. They will crack their own messages just for the fun of it, or help build newer and better ciphers. But others will use this knowledge to read other people's private information. They may tell themselves that they are not doing anything wrong because violating someone's privacy isn't the same as stealing something. They might think there is no harm if the person whose privacy they are invading never finds out. They might think that it isn't wrong just because they are too busy thinking about how clever they are.

These are very poor excuses. They are the excuses that unscrupulous hackers tell themselves so they don't feel guilty for their actions. If you choose to learn more about cryptography and computers beyond this book, you may become so skilled that one day you will be able to read someone's email account and not get caught. Please don't. There are many useful and helpful ways to use skills in cryptography and computer programming. Use your talents to make the world better, not worse.

Chapter 8 – Designing Programs with Flow Charts

The Caesar Cipher program is a bit more complicated than the really simple programs like Hello World, so let's take a moment to think about how it's put together. It may help to draw a flow chart. A flow chart is a picture that shows every possible action that can happen in our program, and in what order. For example, here is a flow chart for the Caesar Cipher program:

TODO: Completed flow chart.

A flow chart is a diagram that shows a series of steps as a number of boxes connected with arrows. Each box represents a step, and the arrows show how one step leads to other steps. You can trace through the flow chart by putting your finger on the "Start" box of the flow chart and following the arrows to other boxes until you get to the "End" box. You can only move from one box to another in the direction of the arrow. You can never go backwards (unless there is a second arrow going back, like in the " Copy mode, copy lastMessage to clipboard." box.)

Of course, we don't have to make a flow chart. We could just start writing code. But often, once we start programming, we will think of things that need to be added or changed that we hadn't considered before. We may end up having to change or delete a lot of code that we had already written, which would be a waste of effort. To avoid this, it's always best to think carefully, and plan how the program will work before we start writing it.

The following flow chart is provided as an example of what flow charts look like and how to make them. For now, since you're just using the source code from this book, you don't need to draw a flow chart before writing code. The program is already written, so you don't have to plan anything out. But when you make your own games, a flow chart can be very handy.

Creating the Flow Chart

Keep in mind, your flow charts don't always have to look exactly like this one. As long as you understand the flow chart you made, it will be helpful when you start coding. We'll begin with a flow chart that only has a "Start" and an "End" box, as shown in Figure 8-2:

TODO: Start and end box.

Our program starts by asking the player what mode they want to enter. So we will draw a box that the start box goes to and label it “Get mode from user.” This box will have arrows going to other boxes, one for each of the modes the user can enter: Encrypt a message, decrypt a message, copy

the last message to the clipboard, generate a random key, or quit. These boxes are added to the flow chart with arrows going to them from the “Get mode from user.” box.

TODO: Next box.

The encrypt and decrypt modes both do the same things. First they get the message to translate from the user, then they get the key to use form the user, and then they do the translation, display it on the screen, and store it in the lastMessage variable. We can add these boxes to our flow chart as well:

TOOD: Image

After this is done the program should go back to asking the user what mode they want to enter, so let’s draw an arrow going back to the “Get mode from user.” box.

TODO: Image

If the user chose the “Copy Mode” option from the “Get mode from user.” box, then the program will simply copy the contents of lastMessage to the clipboard and then goes back to the “Get mode from user.” box. The flow chart now looks like this:

TODO: Image

The “Generate random key.” box has a similar flow. It displays a random key on the screen, and then afterwards goes back to the “Get mode from user.” box.

TODO: Image

The last box the user could go to from the “Get mode from user.” box is if they wanted to quit the program. This box will just have an arrow going from itself to the “End” box, indicating that this is how the program will terminate.

TODO: Image

Now we have the completed flow chart for the Caesar Cipher program. If you were designing and writing this program yourself from scratch, this flow chart would help you remember everything you want the program to do. It can also give you insights into how you should write your code. For example, every time you see a path of arrows go back to an earlier box, this indicates a loop of some kind is needed in your program. Or if you see one box that has several arrows going from it to other boxes, this indicates an if-elif-else statement is needed.

Summary: The Importance of Planning Out the Program

It may seem like a lot of work to sketch out a flow chart about the program first. After all, people want to use encryption programs, not look at flowcharts! But it is much easier to make changes and notice problems by thinking about how the program works before writing the code for it.

If you jump in to write the code first, you may discover problems that require you to change the code you've already written. Every time you change your code, you are taking a chance that you create bugs by changing too little or too much. It is much better to know what you want to build before you build it.

Exercises

Chapter 9 – Using the Debugger

Bugs!

"On two occasions I have been asked, 'Pray, Mr. Babbage, if you put into the machine wrong figures, will the right answers come out?' I am not able rightly to apprehend the kind of confusion of ideas that could provoke such a question."

-Charles Babbage, 19th century English mathematician, philosopher, inventor and mechanical engineer who originated the concept of a programmable computer.

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

If you enter the wrong code, the computer will not give you the right program. A computer program will always do what you tell it to, but what you tell the program to do might not be the same as what you wanted the program to do. A bug is another name for an error or problem in a computer program. Bugs happen when the programmer has not carefully thought about what exactly the program is doing. There are three types of bugs that can happen with your program:

- Syntax Errors are a type of bug that comes from typos in your program. When the Python interpreter sees a syntax error, it is because your code is not written in proper Python language. A Python program with even a single syntax error will not run.
- Runtime Errors are bugs that happen while the program is running (that is, executing). The program will work up until it reaches the line of code with the error, and then the program terminates with an error message (this is called crashing). The Python interpreter will display something called a "traceback" and show the line where the problem happens.
- Semantic Errors are the trickiest bugs to fix. This bug does not crash the program, and the program may appear to work fine. However, it is not doing what the programmer intended for the program to do. For example, if the programmer wants the variable total to be the sum of the values in variables a, b, and c but writes total = a + b * c, then the value in total will be wrong. This won't cause the program to crash immediately, but may or may not cause some other code to crash later on because of the unexpected value in total.

Finding bugs in our program can be hard, if you even notice them at all! When running your program, you may discover that sometimes functions are not called when they are supposed to be, or maybe they are called too many times. You may code the condition for a while loop wrong, so that it loops the wrong number of times. (A loop in your program that never exits is a kind of bug

is called an infinite loop. In order to stop this program, you can press Ctrl-C in the interactive shell.) Any of these things could mistakenly happen in your code if you are not careful.

It can be hard to figure out how your code could be producing a bug because all the lines of code get executed very quickly and the values in variables change so often. A debugger is a program that lets you step through your code one line at a time (in the same order that Python executes them), and shows what values are stored in all of the variables. A debugger lets you look at how each line of code affects your program. This can be very helpful to figure out what exactly the program is doing.

A video tutorial on using the debugger that comes with IDLE can be found on this book's website at http://inventwithpython.com/videos/

How to Not Fix Bugs

When they find that their program is not working how they want it to, many beginner programmers will want to put print() function calls around their program to print out what values are in variables. Resist the temptation to do this. What ends up happening is that the programmer adds a print() call to their code, reruns the program, and looks at what the value is. This gives them a hunch as to what the problem is, so they add another print() call and rerun the program again. And then they add more print() calls and rerun the program over and over until they solve the problem.

This is an easy way to find bugs, but you waste a lot of time running the program over and over again. If you take the time now to learn how to use a debugger, you will be able to fix problems with your code much faster.

Starting the Debugger

In IDLE, go ahead and open the Caesar Cipher program that you made in chapter 6. In the interactive shell, click on File and then Open, and then select caesar.py (or whatever you named the file when you saved it).

After opening the caesar.py file, click on the Debug menu item at the top of the interactive shell, and then click Debugger to make the Debug Control window appear (Figure 7-1).

Now when you run the Caesar Cipher program (by pressing F5 or clicking Run, then Run Module in the file editor window's top menu), the debugger program will be activated. This is called running a program “under a debugger”. In the Debug Control window, check the Source and Globals checkboxes. Then run the program by pressing F5 in the file editor window (Figure 7-2).

When you run Python programs with the debugger activated, the program will stop before it executes the first line of code. If you click on the file editor window's title bar (and you have checked the Source checkbox in the Debug Control window), the first line of code is highlighted in gray. Also, the Debug Control window shows that you are on line 3, which is the import random, pyperclip, string line (line 1 is a comment and line 2 is a blank line, so the debugger skips those lines).

The debugger lets you execute one line or code at a time (this is called "stepping"). To execute a single instruction, click the Step or Over button in the Debug Window. Go ahead and click the Over button once. This will cause the Python interpreter to execute the import random, pyperclip, string instruction, and then stop before it executes the next instruction. The Debug Control window will change to show that you are now on line 5, the def main(): line.

Note: In the Windows version of IDLE, there seems to be a bug if you press Step instead of Over to execute an import statement that imports a module that does not come with Python (such as our pyperclip module). If IDLE does not respond to mouse clicks or keyboard presses, just click the X in the corner and have Windows shut down the program, and then restart it. You don't need to restart Windows.

Stepping

Stepping is the process of executing one instruction of the program at a time. Doing this lets you see what happens after running a single line of code, which can help you figure out where a bug first appears in your programs.

The Debug Control window will show you what line is about to be executed when you click the Step button in the Debug Control window. This window will also tell you what line number it is on and show you the line of code itself.

Clicking the Step button will execute the line of code, and jump into any function calls or import statements. Clicking the Over button will execute all of the code inside the function or module and move to the next line after the function call or import statement. Think of the Step button as "stepping into" the function call, whereas the Over button executes enough instructions to appear to "step over" the function call.

Click the Step button a few more times. This will execute the def statements at the beginning of the caesar.py program to define these functions. As you define these functions, they will appear in the Globals area of the Debug Control window.

The text next to the function names in the Global area will look something like "<function main at 0x012859B0>". The module names also have confusing looking text next to them, such as

"<module 'random' from 'C:\\Python31\\lib\\random.pyc'>". This is detailed information is useful to advanced Python programmers, but you don't need to know what it means to debug your programs. Just seeing that the functions and modules are there in the Global area will tell you if the function has been defined or the module has been imported. You can also ignore the __builtins__, __doc__, and __name__ lines in the Global area. (Those are variables that appear in every Python program.)

The Global area in the Debug Control window is where all the global variables are stored. Remember, global variables are the variables that are created outside of any functions (that is, in the global scope). There is also a Local area, which shows you the local scope variables and their values. The local area will only have variables in it when the program execution is inside of a function. Since we start in the global scope and stay there until we call the main() function, this area is blank.

The Python debugger (and almost all debuggers) only lets you step forward in your program. Once you have executed an instruction, you cannot step backwards and undo the instruction.

The Go and Quit Buttons

If you get tired of clicking the step button over and over again, and just want the program to run normally, click the Go button at the top of the Debug Control window. This will tell the program to run as if you didn't have the debugger turned on.

If you ever want to terminate the program while it is running, just click the Quit button at the top of the Debug Control window. The program will immediately exit. This can be handy if you want to stop the program and start debugging it from the beginning again.

Stepping Into, Over, and Out

Start the Caesar Cipher program with the debugger, and keep stepping (by clicking the Step button in the Debug Control window) until the debugger is at line 38 (the call to displayIntro() line). When you click Step again, the debugger will jump into this function call and appear on line 5 (the first line in the def-block of the displayIntro() function. The kind of stepping we have been doing is called stepping into, because it will step into function calls.

If you click Step a few more times, you will see the output of the print() function call appear in the interactive shell window one at a time. When you step over the last print() function call in the displayIntro() function, the debugger will jump back to the first line (line 40) after function call.

Click Step one more time to step into the chooseCave() function. Keep stepping through the code until you execute the function call input() call. The program will wait until you type a response

into the shell, just like when you run the program normally. If you try clicking the Step button now, nothing will happen because the program is waiting for a keyboard response.

Enter a response by clicking back on the interactive shell window and type which cave you want to enter. You have to click on the bottom line in the shell before typing. If you are typing but nothing appears on the screen (and the blinking cursor is not below the Which cave will you go into? (1 or 2) text), then you have not clicked on the last line of the shell window.

Once you press the Enter key to enter your response, the debugger will continue to step lines of code again. Instead of clicking Step, try clicking the Out button on the Debug Control window. This is called stepping out, because it will cause the debugger to step over as many lines as it needs to until it jumps out of the function that it was in. After it jumps out, the execution will be on the line after the line that called the function. For example, if you were inside the displayIntro() function on line 6, clicking Out would have the debugger keep stepping until the function was over and returned to the line after the call to displayIntro(). Stepping out can save you from having to click Step over and over again to jump out of the function.

If you are not inside a function (that is, you are in the global scope) and you click Out, the debugger will execute all the remaining lines in the program (exactly as if you clicked the Go button).

The last kind of stepping is done by the Over button in the Debug Control window, and it is for stepping over function calls. Stepping over means that the debugger will not step into function calls. Instead, the debugger executes all the code inside the function at once and only stop at the line after the function call. This is useful if you do not want to step through every single line inside the function. (Think of Stepping Over as the same as Stepping Into and then immediately Stepping Out.)

You now know what the five buttons at the top of the Debug Control window do. Here's a recap of what each button does:

Go - Executes the rest of the code as normal, or until it reaches a break point. (Break points are described later.)

Step - Step one line of code. If the line is a function call, the debugger will step into the function.

Over - Step one line of code. If the line is a function call, the debugger will not step into the function, but instead step over the call.

Out - Keeps stepping over lines of code until the debugger leaves the function it was in when Out was clicked. This steps out of the function.

Quit - Immediately terminates the program.

Find the Bug

Using the debugger is a good way to figure out what is causing bugs in your program. As an example, here is a small program that has a bug in it. The program comes up with a random addition problem for the user to solve. In the interactive shell window, click on File, then New Window to open a new file editor window. Type this program into that window, and save the program as buggy.py.

```
import random
number1 = random.randint(1, 10)
number2 = random.randint(1, 10)
print('What is ' + str(number1) + ' + ' + str(number2) + '?')
answer = input()
if answer == number1 + number2:
    print('Correct!')
else:
    print('Nope! The answer is ' + str(number1 + number2))
```

Type the program in exactly as it is above, even if you can already tell what the bug is. Then trying running the program by pressing F5. This is a simple arithmetic program that comes up with two random numbers and asks you to add them. Here's what it might look like when you run the program:

```
What is 5 + 1?
6
Nope! The answer is 6
```

That's not right! This program has a semantic bug in it. Even if the user types in the correct answer, the program says they are wrong.

You could look at the code and think hard about where it went wrong. That works sometimes. But you might figure out the cause of the bug quicker if you run the program under the debugger. At the top of the interactive shell window, click on Debug, then Debugger (if there is no check already by the Debugger menu item) to display the Debug Control window. In the Debug Control window, make sure the all four checkboxes (Stack, Source, Locals, and Globals) are checked. This makes the Debug Control window provide the most information. Then press F5 in the file editor window to run the program under the debugger.

The debugger starts at the import random line. Nothing special happens here, so just click Step to execute it. You should see the random module added to the bottom of the Debug Control window in the Globals area.

Click Step again to run line 2. A new file editor window will pop open showing the random.py file. Remember that the randint() function is inside the random module. When you stepped into the function, you stepped into the random module because that is where the randint function is. The functions that come with Python's modules almost never have bugs in their code, so you can just click Out to step out of the randint() function and back to your program. After you have stepped out, you can close the random module's window.

Line 3 is also a call to the randint() function. We don't need to step through this code, so just click Over to step over this function call. The randint() function's code is still executed, it is just executed all at once so that we don't have to step through it.

Line 4 is a print() call to show the player the random numbers. But since we are using the debugger, we know what numbers the program will print even before it prints them! Just look at the Globals area of the Debug Control window. You can see the number1 and number2 variables, and next to them are the integer values stored in those variables. When I ran the debugger, it looked like Figure 7-4.

Figure 7-4: number1 is set to 9 and number2 is set to 10.

The number1 variable has the value 9 and the number2 variable has the value 10. When you click Step, the program will display the string in the print() call with these values. (Of course, we use the str() function so that we can concatenate the string version of these integers.)

Clicking on Step on line 5 will cause the debugger to wait until the player enters a response. Go ahead and type in the correct answer (in my case, 19) into the interactive shell window. The debugger will resume and move down to line 6.

Line 6 is an if statement. The condition is that the value in answer must match the sum of number1 and number2. If the condition is True, then the debugger will move to line 7. If the condition is False, the debugger will move to line 9. Click Step one more time to find out where it goes.

The debugger is now on line 9! What happened? The condition in the if statement must have been False. Take a look at the values for number1, number2, and answer. Notice that number1 and number2 are integers, so their sum would have also been an integer. But answer is a string. That means that the answer == number1 + number2 condition would have evaluated to '19' == 19. A string value and an integer value will always not equal each other, so the condition would have evaluated to False.

That is the bug in the program. The bug is that we use answer when we should be using int(answer). Go ahead and change line 6 to use int(answer) == number1 + number2 instead of answer == number1 + number2, and run the program again.

```
What is 2 + 3?
5
Correct!
```

This time, the program worked correctly. Run it one more time and enter a wrong answer on purpose to make sure the program doesn't tell us we gave the correct answer. We have now debugged this program. Remember, the computer will run your programs exactly as you type them, even if what you type is not what you intend.

Break Points

Stepping through the code one line at a time might still be too slow. Often you will want the program to run at normal speed until it reaches a certain line. You can do this with break points. A break point is set on a line when you want the debugger to take control once execution reaches that line. So if you think there is a problem with your code on, say, line 17, just set a break point on line 17 (or maybe a few lines before that) and when execution reaches that line, the debugger will stop execution. Then you can step through a few lines to see what is happening. Then you can click Go to let the program execute until it reaches the end (or another break point).

To set a break point, right-click on the line that you want a break point on and select "Set Breakpoint" from the menu that appears. The line will be highlighted with yellow to indicate a break point is on that line. You can set break points on as many lines as you want. To remove the break point, click on the line and select "Clear Breakpoint" from the menu that appears.

Figure 7-5: The file editor with two break points set.

Example of Using Break Points

Let's try debugging a program with break points. Here is a program that simulates coin flips by calling random.randint(0, 1). Each time this function call returns the integer 1, we will consider that "heads" and increment a variable called heads. We will also increment a variable called flips to keep track of how many times we do this "coin flip".

The program will do "coin flips" one thousand times. This would take a person over an hour to do, but the computer can do it in one second! Type in the following code into the file editor and save it as coinFlips.py. You can also download this code from http://inventwithpython.com/coinFlips.py

coinFlips.py

This code can be downloaded from http://inventwithpython.com/coinFlips.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/diff or email the author at al@inventwithpython.com

```
import random
print('I will flip a coin 1000 times. Guess how many times it will come up
heads. (Press enter to begin)')
input()
flips = 0
heads = 0
while flips < 1000:
    if random.randint(0, 1) == 1:
        heads = heads + 1
    flips = flips + 1
    if flips == 900:
        print('900 flips and there have been ' + str(heads) + ' heads.')
    if flips == 100:
        print('At 100 tosses, heads has come up ' + str(heads) + ' times so
far.')
    if flips == 500:
        print('Half way done, and heads has come up ' + str(heads) + ' times.')
print()
print('Out of 1000 coin tosses, heads came up ' + str(heads) + ' times!')
print('Were you close?')
```

The program runs pretty fast. It probably spent more time waiting for the user to press the Enter key than it did doing the coin flips. Let's say we wanted to see it do coin flips one by one. On the interactive shell's window, click on Debug and then Debugger at the top menu to bring up the Debug Control window. Then press F5 to run the program.

The program starts in the debugger on line 1. Press Step three times in the Debug Control window to execute the first three lines (that is, lines 1, 2, and 3). You'll notice the buttons become disabled because the input() function has been called and the interactive shell window is waiting for the player to type something. Click on the interactive shell window and press Enter. (Be sure to click beneath the text in the shell window, otherwise IDLE might not receive your keystrokes.) After entering text for the input() call, the Step buttons will become enabled again.

You can click Step a few more times, but you'll find that it would take quite a while to get through the entire program. Instead, set a break point on lines 12, 14, and 16 (Figure 7-6).

Figure 7-6: Three break points set.

After setting the breakpoints, click Go in the Debug Control window. The program will run at its normal speed until it reaches flip 100. On that flip, the condition for the if statement on line 13 is True. This causes line 14 (where we have a break point set) to execute, which tells the debugger to stop the program and take over. Look at the Debug Control window in the Globals section to see what the value of flips and heads are.

Click Go again and the program will continue until it reaches the next break point on line 16. Again, see how the values in flips and heads have changed. You can click Go one more time to continue the execution until it reaches the next break point.

And if you click Go again, the execution will continue until the next break point is reached, which is on line 12. You probably noticed that the print() functions on lines 12, 14 and 16 are called in a different order than they appear in the source code. That is because they are called in the order that their if statement's condition becomes True. Using the debugger can help make it clear why this is.

Summary

Writing code is only part of the work for making a working program. The next part is making sure the code we wrote actually works. Debuggers let us step through the code one line at a time, while examining which lines execute (and in what order) and what values the variables contain. When this is too slow, we can set break points and click Go to let the program run normally until it reaches a break point.

Using the debugger is a great way to understand what exactly a program is doing. While this book provides explanations of all the programs in it, the debugger can help you find out more on your own.

Chapter 10 - The Transpositional Cipher

As you saw in the last chapter, the Caesar Cipher isn't that good. It doesn't take much for a computer to decrypt all twenty six possible keys. The transpositional cipher has many more possible keys to make a brute force attack more difficult. This chapter will also teach you how you can encrypt files on your computer, rather than short messages you type or copy/paste into the program. These files can have thousands or millions of words in them, but your computer will be able to encrypt and decrypt them in seconds.

The Caesar Cipher would encrypt all the letters in a message, but any numbers and punctuation would be left alone. Another benefit of the transpositional cipher is that it will encrypt everything, not just letters.

How the Transpositional Cipher Works

The transpositional cipher does not replace letters with other letters. Rather, it jumbles up the message's existing symbols into an order that makes the original message unreadable.

Let's encrypt the message "Common sense is not so common." Including the spaces and punctuation, this message has 30 characters. We will use the number 8 for a key.

The first step for encrypting with the transpositional cipher is to draw out a number of boxes in a row equal to the key. If the key were 10, then we'd draw 10 boxes in a row. If the key were 500, we'd draw 500 boxes. So for our example, we draw 8 boxes since we have a key of 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 (we'll mark the box with (s) to indicate a space.

C	o	m	m	o	n	(s)	s

Of course, we only have 8 boxes but there are 30 characters. So draw another row of 8 boxes under the first row. In fact, if you ever fill up a row and still have characters left, keep creating a new row:

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 have two boxes left over on the last row, so let's just shade those boxes in to remind us to ignore them. In order to Transpositional cipher encryption, read the letters from the top left box and go down. This gives us the characters "C", "e", "n", and "o".

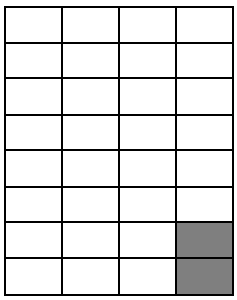
When you get to the bottom of the row, start at the top of the next column to the right. This gives us the characters "o", "n", "o", "m". Do this for all of the unshaded boxes. The final ciphertext will be "Cenoonommstmme oo snnio. s s c", which is sufficiently scrambled to keep someone from reading the original message.

1. Get the number of characters in the message and the key.
2. Draw a number of boxes equal to the key in one row.
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.
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.

To decrypt, we need the ciphertext and the key. Going from our previous example, our ciphertext will be "Cenoonommstmme oo snnio. s s c" and the key is 8. The first step is to calculate how many boxes we need 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 (just like the plaintext) and the key is 8. So calculate 30 / 8 which is 3.75. Round this number up to 4. So we want to draw boxes of 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 just round up to 6.0, not 7.0. The number only increases if there is anything besides 0 after the decimal point.

The second thing we need to calculate is how many shaded boxes to draw. To do this, we take the total number of boxes (32) and subtract the length of the ciphertext (30). 32 – 30 = 2, so the bottom 2 boxes on the rightmost column should be shaded in:



Then start filling in the boxes with one character in the ciphertext per box. Start at the top left and go right, just like we did when we were encrypting. 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	

You might see that the if you read the text going down the columns, the original plaintext has been restored: “Common sense is not so common.”

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 next column to the right after you reach the bottom of a column.

Note that if you use a different key, you will be drawing out the wrong number of rows, and even if you follow the other steps in the decryption process correctly, the plaintext will come out looking like random garbage. (If you ever want to frustrate someone, give them the wrong the key to a very large ciphertext and watch them get nothing but garbage after drawing out a pages of boxes.)

This explains how to perform the transpositional cipher on paper. It involves a lot of box drawing and careful writing, and can be easy to make mistakes. Now let's look a program that can implement this cipher for us.

Sample Run of Transpositional Cipher

This is what the transpositional cipher program looks like when you run it:

```
Transpositional Cipher
e  - Encrypt Message
ef - Encrypt File
d  - Decrypt Message
df - Decrypt File
k  - Generate Key
q  - Quit
> e

Enter your message:
> Anyone who has the power to make you believe absurdities has the power to
make you commit atrocities. -Voltaire
Enter an integer for the key:
> 30
Your encrypted text is:
Ak anetty hroyeono ceupi  otwbwiheeeolrs i .het avo-se V  motaalhbktesea u
ipryrodoewiuet ric eotsmo m himatas |

Transpositional Cipher
e  - Encrypt Message
ef - Encrypt File
d  - Decrypt Message
df - Decrypt File
```

```
k  - Generate Key
c  - Copy "Anyone who..." to clipboard
q  - Quit
> d

Enter your message:
> Ak anetty hroyeono ceupi  otwbwiheeeolrs i .het avo-se V  motaalhbktesea u
ipryrodoewiuet ric eotsmo m himatas
Enter an integer for the key:
> 20
Your decrypted text is:
Ato  wo a tkapduitmaktyc il.vVat roecs t yeeohrho aeuyet mhaa outese- ls rw
eoisnhnpwe tsmheioiro m eroibei eob|

Transpositional Cipher
e  - Encrypt Message
ef - Encrypt File
d  - Decrypt Message
df - Decrypt File
k  - Generate Key
c  - Copy "Atjnuocoit..." to clipboard
q  - Quit
> d

Enter your message:
> Ak inetny hjoyeuno seupt  oiwbwcheeeolrs i .hetavose   mtaahbkese u
pryodowiuet ric eotsmo m himatas
Enter an integer for the key:
> 30
Your decrypted text is:
Anyone who has the power to make you believe absurdities has the power to make
you commit atrocities. -Voltaire|

Transpositional Cipher
e  - Encrypt Message
ef - Encrypt File
d  - Decrypt Message
df - Decrypt File
k  - Generate Key
c  - Copy "Anyone who..." to clipboard
q  - Quit
> q
```

Notice that after the encrypted or decrypted text is printed to the string, there is a | character at the end of it. (The | character is called the “pipe” character.) This is because the Transpositional

Cipher may end up placing spaces at the end of the message, and you cannot see spaces printed on the screen at the end of the line. This is why the pipe character marked the end of the line.

This is important because if you do not have the When the program copies the encrypted or decrypted text to the clipboard, it will not include the pipe.

For example:

```
Hello| # There are no spaces after "Hello" and before the pipe.
Hello | # There is one space after "Hello" and before the pipe.
Hello  | # There are two spaces after "Hello" and before the pipe.
```

Encrypting and Decrypting Files

Notice that the Transpositional Cipher program has an ef and df mode to encrypt and decrypt files on your hard drive. You can write your own text files using Notepad (on Windows), TextMate or TextEdit (on Mac 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. The following text files can be downloaded from this book's website:

- http://becomeacodebreaker.com/candide.txt
- http://becomeacodebreaker.com/devilsdictionary.txt
- http://becomeacodebreaker.com/frankenstein.txt
- http://becomeacodebreaker.com/grimmsfairytales.txt
- http://becomeacodebreaker.com/siddhartha.txt
- http://becomeacodebreaker.com/thetimemachine.txt
- http://becomeacodebreaker.com/waroftheworlds.txt

These are text files of some books (now in the public domain, so it is perfectly legal to download them.) For example, download H.G. Wells' classic novel "The Time Machine" from http://becomeacodebreaker.com/thetimemachine.txt. Double click the file to open it in a text editor program. There are over 35,000 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.

Be sure to put the thetimemachine.txt file in the same directory as the transpositional.py file before running the program. When you run it, it will look something like this:

```
Transpositional Cipher
e  - Encrypt Message
ef - Encrypt File
```

```
d  - Decrypt Message
df - Decrypt File
k  - Generate Key
q  - Quit
> ef

Enter the source file.
Or enter CANCEL to quit.
> thetimemachine.txt
Enter the target file.
Or enter CANCEL to quit.
> thetimemachine_encrypted.txt
Enter an integer for the key:
> 1013
Encrypting file thetimemachine.txt...
Writing content to file thetimemachine_encrypted.txt is complete.

Transpositional Cipher
e  - Encrypt Message
ef - Encrypt File
d  - Decrypt Message
df - Decrypt File
k  - Generate Key
q  - Quit
> q
```

In the directory that thetimemachine.txt is, there will be a new file named thetimemachine_encrypted.txt that contains the content of thetimemachine.txt in encrypted form. If you double click the file to open it, it should look something like this:

```
P P,an e c V'is.ir pde.tbatden oasne h. ietthnihoaloeblent!nho ggich...

(the rest has been cut out for brevity)
```

If you run the Transpositional Cipher program again and decrypt the thetimemachine_encrypted.txt file with the key 1013, you will get a file that is identical to the original thetimemachine.txt file. To prevent the user from accidentally replacing files, the Transpositional Cipher requires a new file name before it writes a decrypted or encrypted file.

```
Transpositional Cipher
e  - Encrypt Message
ef - Encrypt File
d  - Decrypt Message
df - Decrypt File
```

```
k  - Generate Key
q  - Quit
> df

Enter the source file.
Or enter CANCEL to quit.
> thetimemachine_encrypted.txt
Enter the target file.
Or enter CANCEL to quit.
> thetimemachine_decrypted.txt
Enter an integer for the key:
> 1013
Decrypting file thetimemachine_encrypted.txt...
Writing content to file thetimemachine_decrypted.txt is complete.

Transpositional Cipher
e  - Encrypt Message
ef - Encrypt File
d  - Decrypt Message
df - Decrypt File
k  - Generate Key
q  - Quit
> q
```

If you open the thetimemachine_decrypted.txt file, you will find that it is the same as the original thetimemachine.txt file (unless you entered the wrong key).

To find other public domain texts to download, go to the Project Gutenberg website at http://www.gutenberg.org/.

Source Code of the Transpositional Cipher

The transpositional cipher program will be spread across two files. One file, transpositional.py, will have the main transpositional cipher code. The other file, codebreaker.py, will contain functions that we will use again in other cryptography programs. Instead of typing them again in each program, we can put this code in its own file and import it (just like we import the pyperclip or random modules.)

Type the following code into IDLE's file editor and save it as transpositional.py. You can also download this source code from http://becomeacodebreaker.com/chapter10.

```
1. # Transpositional Cipher http://becomeacodebreaker.com
2.
3. import codebreaker, sys, random, pyperclip, math
```

```
  4.
  5. def main():
  6.     if codebreaker.version < 1.0:
  7.         sys.exit('Must have at least codebreaker module version 1.0, not
%s' % codebreaker.version)
  8.
  9.     lastMessage = ''
 10.     mode = ''
 11.     while mode != 'q':
 12.         print()
 13.         print('Transpositional Cipher')
 14.         mode = codebreaker.getMode('e ef d df k c q ', lastMessage)
 15.         if mode == 'k':
 16.             # come up with a random key for the user
 17.             print(generateRandomKey())
 18.         elif mode == 'c':
 19.             # copy the last message to the clipboard
 20.             pyperclip.copy(lastMessage)
 21.         elif mode == 'e' or mode == 'd':
 22.             # encrypting and decrypting messages
 23.             message = codebreaker.getMessage()
 24.             key = getKey()
 25.             if mode == 'e':
 26.                 print('Your encrypted text is:')
 27.                 translated = getEncryptedMessage(message, key)
 28.             elif mode == 'd':
 29.                 print('Your decrypted text is:')
 30.                 translated = getDecryptedMessage(message, key)
 31.             print(translated + '|')
 32.             lastMessage = translated
 33.         elif mode == 'ef' or mode == 'df':
 34.             # encrypting and decrypting files
 35.             sourcefilename = codebreaker.getFilename('Enter the source
file.', 'exists')
 36.             if sourcefilename == None:
 37.                 continue
 38.             targetfilename = codebreaker.getFilename('Enter the target
file.', 'does not exist')
 39.             if targetfilename == None:
 40.                 continue
 41.
 42.             key = getKey()
 43.
 44.             sourcefo = open(sourcefilename, 'r')
 45.             sourcecontent = sourcefo.read()
 46.             sourcefo.close()
```

```
 47.
 48.             if mode == 'ef':
 49.                 print('Encrypting file %s...' % (sourcefilename))
 50.                 translated = getEncryptedMessage(sourcecontent, key)
 51.             elif mode == 'df':
 52.                 print('Decrypting file %s...' % (sourcefilename))
 53.                 translated = getDecryptedMessage(sourcecontent, key)
 54.
 55.             targetfo = open(targetfilename, 'w')
 56.             targetfo.write(translated)
 57.             targetfo.close()
 58.             print('Writing content to file %s is complete.' %
(targetfilename))
 59.
 60.
 61. def getKey():
 62.     # This function returns the integer the user typed in for the key.
 63.     while True:
 64.         print('Enter an integer for the key:')
 65.         key = input('> ')
 66.         if key.isdigit() and int(key) > 0:
 67.             return int(key)
 68.
 69.
 70. def generateRandomKey():
 71.     # This function returns a random integer for the key.
 72.     return random.randint(2, 100)
 73.
 74.
 75. def getEncryptedMessage(message, key):
 76.     # This function returns a string that is the encrypted version of the
 77.     # string in the message parameter, using the key of the key parameter.
 78.     ciphertext = [''] * key
 79.     for col in range(key):
 80.         pointer = col
 81.         while pointer < len(message):
 82.             ciphertext[col] += message[pointer]
 83.             pointer += key
 84.     return ''.join(ciphertext)
 85.
 86.
 87. def getDecryptedMessage(message, key):
 88.     # This function returns a string that is the decrypted version of the
 89.     # string in the message parameter, using the key of the key parameter.
 90.     numOfColumns = math.ceil(len(message) / key)
 91.     numOfRows = key
```

```
 92.     numOfShadedBoxes = numOfColumns * numOfRows - len(message)
 93.     plaintext = [''] * numOfColumns
 94.
 95.     col = 0
 96.     row = 0
 97.     for i in range(len(message)):
 98.         plaintext[col] += message[i]
 99.         col += 1
100.         if col == numOfColumns or (col == numOfColumns - 1 and row >=
numOfRows - numOfShadedBoxes):
101.             col = 0
102.             row += 1
103.     return ''.join(plaintext)
104.
105.
106. if __name__ == '__main__':
107.     main()
```

Here is the source code for the codebreaker.py file. You can also download this source code from http://becomeacodebreaker.com/chapter10.

```
 1. # Codebreaker Utility Module, http://becomeacodebreaker.com
 2. # version 1.0
 3.
 4. import os
 5.
 6. version = 1.0
 7.
 8. # Added in Version 1.0:
 9. def getMode(modeTypes, clipboardText):
10.     # This function returns 'e', 'd', 'k', 'c', 'q', 'ef', or 'df'
11.     # depending on what the user enters.
12.     if len(clipboardText) > 10:
13.         clipboardText = clipboardText[:10] + '...'
14.
15.     modeShortNames = ['e', 'ef', 'd', 'df', 'k', 'c', 'q']
16.     modeNames = [['e', 'Encrypt Message'],
17.                  ['ef', 'Encrypt File'],
18.                  ['d', 'Decrypt Message'],
19.                  ['df', 'Decrypt File'],
20.                  ['k', 'Generate Key'],
21.                  ['c', 'Copy "' + clipboardText + '" to clipboard'],
22.                  ['q', 'Quit']]
23.     mode = ''
24.     while mode not in modeShortNames:
25.         for modeName in modeNames:
```

```
26.             if modeName[0] + ' ' in modeTypes:
27.                 extraSpace = ''
28.                 if len(modeName[0]) == 1:
29.                     extraSpace = ' '
30.                 print('%s%s - %s' % (modeName[0], extraSpace, modeName[1]))
31.
32.         # get choice from user
33.         mode = input('> ').lower()
34.     print()
35.     return mode
36.
37.
38. def getMessage():
39.     # This function returns the string of the message the player typed in.
40.     print('Enter your message:')
41.     return input('> ')
42.
43.
44. def getFilename(directions, fileRequirement):
45.     while True:
46.         print(directions)
47.         print('Or enter CANCEL to quit.')
48.         filename = input('> ')
49.         print()
50.         if filename.upper() == 'CANCEL':
51.             return
52.         if fileRequirement == 'exists' and not os.path.exists(filename):
53.             print('Could not find the file %s' % (filename))
54.             continue
55.         elif fileRequirement == 'does not exist' and
os.path.exists(filename):
56.             print('That file already exists.')
57.             continue
58.         return filename
```

The next few sections will explain how the Transpositional Cipher program works, along with the codebreaker module. But first, here's an overview of what the functions in transpositional.py and codebreaker.py do:

The transpositional.py Functions

- **main()** – This is the main file in the Transpositional Cipher program, and calls all the other functions in the program.
- **getKey()** – This function asks the user to type in a Transpositional Cipher key and does input validation to make sure it is an integer larger than 0. The return value of this function is an integer of the key the user selected.

- **generateRandomKey()** – This function generates a random key between 2 and 1000 for the user. (Although it doesn't make sense to use a key that is larger than the number of characters in the message, but the user can just keep generating random keys until they find one smaller than the message length.) The return value of this function is this random integer.
- **getEncryptedMessage()** – This function takes the plaintext message and key as parameters and returns a string of encrypted text.
- **getDecryptedMessage()** – This function takes the ciphertext message and key as parameters and returns a string of decrypted text.

The codebreaker.py Functions

- **getMode()** – This function handle the user typing in what they want the program to do, and returns a string that corresponds to the mode the user selected. The return value of this function is the string 'e' or 'd' (for encrypting/decrypting), or 'ef' or 'df' (for encrypting/decrypting a file), or 'k' (for generating a random key), or 'c' (for copying the last message to the clipboard), or 'q' (for quitting the program.)
- **getMessage()** – This function handles the user typing in the message they want to encrypt or decrypt. The return value of this function is the string value that the user typed in.
- **getFilename()** – This function handles the user typing in the name of a file. A 'exists' string value can be passed to it to specify that the user needs to type in the name of a file that already exists. Or the string value 'does not exist' can be passed to specify that the user needs to type in the name of a file that doesn't already exist. (This is used to make sure the user doesn't overwrite and already existing file.) The return value of this function is either the string value of the filename the user typed in, or the special value None (which is explained later in this chapter).

The Start of the Transpositional Cipher Program

```
1. # Transpositional Cipher http://becomeacodebreaker.com
2.
3. import codebreaker, sys, random, math, pyperclip
```

This program import several different modules, including two modules that don't come with Python: codebreaker and pyperclip. The codebreaker module will contain functions that we'll use in several of the programs in this book. This saves us from retyping these functions into each program.

```
5. def main():
6.     if codebreaker.version < 1.0:
```

```
  7.         sys.exit('Must have at least codebreaker module version 1.0, not
%s' % codebreaker.version)
```

This is the main() function. Like in our previous programs, we will call the main() function as soon as the program begins to run, but after it has imported the modules and executed these def statements.

The codebreaker module will have a variable called version in it. The version variable has the floating point (that is, numbers with decimal points) data type. We will be updating the codebreaker module with new functions as we create new programs. Each time we update it, we will increase the number that is stored in the codebreaker module.

Think about it. What if we later add a function to the codebreaker module called foobar() (and update codebrekaer.version to the value 2.0) because some other program we write uses it? If we had an old 1.0 copy of codebreaker.py that didn't have foobar() in it, it wouldn't matter to our Transpositional Cipher program because it doesn't call codebreaker.foobar() anywhere.

But if our other program imported the old codebreaker 1.0 version, it would create an error because the 1.0 version doesn't have foobar() in it. Line 6 of our program checks that the version of codebreaker it has imported is at least 1.0. If it isn't, it calls sys.exit(). The sys.exit() function will print a string that is passed to it and then immediately terminate the program. You can also call sys.exit() without any arguments if you don't want your program to print anything before terminating.

```
 9.     lastMessage = ''
10.     mode = ''
11.     while mode != 'q':
12.         print()
13.         print('Transpositional Cipher')
14.         mode = codebreaker.getMode('e ef d df k c q ', lastMessage)
```

The next part of the main() function sets up the lastMessage and mode variables with blank strings. The lastMessage variable will store the last message that was encrypted or decrypted in case the user wants to copy it to the clipboard. The mode variable is set to blank so that the condition on line 11's while loop will be True.

This while loop, like the one in the Caesar Cipher program's main() function, will constantly fetch what mode the user wants and then which message to encrypt or decrypt. However, in this program, the getMode() function has been moved to the codebreaker module (because it is a function that can be useful in many other cipher programs besides this one.)

The two arguments we pass to codebreaker.getMode() is a string 'e ef d df k c q ' and the variable lastMessage. The first argument will tell what modes to present to the user on the screen. It is needed because some cipher programs that use the codebreaker module might support different modes. Note that there is a space at the end after the “q”.

The second argument that is passed to getMode() is the lastMessage variable, which is used so that the getMode() function can print part of it to the screen to let the user see what will be copied to the clipboard if he chooses mode 'c'.

```
15.         if mode == 'k':
16.             # come up with a random key for the user
17.             print(generateRandomKey())
18.         elif mode == 'c':
19.             # copy the last message to the clipboard
20.             pyperclip.copy(lastMessage)
```

Lines 15 to 20 handle the case where the user enters 'k' or 'c' for the mode. It is very similar to the code in the previous Caesar Cipher program. Mode 'k' will print a random key that is returned from the generateRandomKey() function. (Notice that generateRandomKey() is not in the codebreaker module because it does not have codebreaker. in front of it.)

The 'c' case will copy the last message that was encrypted or decrypted to the clipboard. This message will always be stored in the lastMessage variable.

```
21.         elif mode == 'e' or mode == 'd':
22.             # encrypting and decrypting messages
23.             message = codebreaker.getMessage()
24.             key = getKey()
```

The above code handles the encryption and decryption modes. The code for both encrypting and decrypting is fairly similar, so we handle both in the same elif block. The code in the elif block executes if getMode() returned either the string 'e' or the string 'd'.

In both cases, we want to call codebreaker.getMessage() to get the message to encrypt or decrypt, followed by calling getKey() to get the encryption or decryption key. Note that codebreaker.getMessage() is in the codebreaker module, while getKey() is not (we define that function on line 61.) This is because getting the message from the user will be the same in all our cryptography programs, while each cipher has its own types of keys so each program will need its own getKey() function. For example, the Caesar Cipher program uses integers from 0 to 25 as keys, but the Transpositional Cipher uses any integer larger than 0. These differences will require different code to handle the different input validation.

```
25.             if mode == 'e':
26.                 print('Your encrypted text is:')
27.                 translated = getEncryptedMessage(message, key)
28.             elif mode == 'd':
29.                 print('Your decrypted text is:')
30.                 translated = getDecryptedMessage(message, key)
31.             print(translated + '|')
32.             lastMessage = translated
```

Our Transpositional Cipher program won't have one function for both encrypting and decrypting. (The code to encrypt is very different from the code to decrypt.) Instead we have getEncryptedMessage() to encrypt a message and getDecryptedMessage() to decrypt a message. Both functions have two parameters: one for the message to be translated and a second for the key to do the translation.

Either way, we will put the encrypted/decrypted string in the translated variable and print it on the screen on line 31 (with a pipe after it to show any spaces at the end of the translated string.) We save a copy of the string in lastMessage so the user can copy it to the clipboard later.

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 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". 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 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. The None value is written without quotes and with a capital "N" and lowercase "one".

The continue and break Statements

The continue statement will make program execution jump back to the beginning of a loop and reevaluate the condition. For example, consider the following program:

```
counter = 0
while counter < 6:
```

```
    counter = counter + 1
    if counter == 3:
        continue
    print(counter)
```

This simple program puts 0 in the counter variable, then enters a loop. The code in the loop increments the counter variable by 1, and then prints the value in the counter variable. You think this would print out 1, then 2, 3, 4, 5, 6. But if you run this program, it looks like this:

```
1
2
4
5
6
```

The reason is because the if statement checks when counter is equal to 3, and if so, it runs the continue statement. When the program executes the continue statement, it skips the rest of the code in the loop's block and jumps back to the while statement to recheck the condition. If the condition is True execution enters the loop and if the condition is False execution skips past the block. The continue statement is very handy when you want to jump back to the beginning of a loop.

The break statement jumps out of the loop without rechecking the condition. For example, consider this program (which is similar to the previous one):

```
counter = 0
while counter < 6:
    counter = counter + 1
    if counter == 3:
        break
    print(counter)
```

This program has a break statement instead of a continue statement. When you run this program, its output looks like this:

```
1
2
```

The reason the program only prints 1 and 2 is because when counter is equal to three, the break statement makes the program execution jumps out of the while loop's block (whether the loop's condition is True or False).

You can view an online trace of these small programs at http://becomeacodebreaker.com/moreinfo.

```
 33.         elif mode == 'ef' or mode == 'df':
 34.             # encrypting and decrypting files
 35.             sourcefilename = codebreaker.getFilename('Enter the source
file.', 'exists')
 36.             if sourcefilename == None:
 37.                 continue
 38.             targetfilename = codebreaker.getFilename('Enter the target
file.', 'does not exist')
 39.             if targetfilename == None:
 40.                 continue
```

Instead of encrypting or decrypting a message that we type in, we could specify the filename of a file we want to encrypt. The codebreaker.getFilename() handles letting the user type in the filename and doing proper input validation. We call codebreaker.getFilename() once to get the source file and once to get the target file. The source file is the file that you want to encrypt or decrypt. The user needs to type the filename of a file that already exists for the source file, which is why 'exists' is passed to codebreaker.getFilename(). The target file is the file to write the encrypted or decrypted results to. To prevent the user from overwriting a file that already exists (which would erase the original file), we pass 'does not exist' to codebreaker.getFilename() so that the input validation code makes sure they type a filename that does not already exist.

If the codebreaker.getFilename() calls ever return the value None, that indicates the user wants to cancel entering a filename (maybe they can't remember the name or have changed their mind about encrypting it). In this case, we execute a continue statement so that the program jumps back to line 11 and asks the player what to do again.

```
 42.             key = getKey()
```

Just like when we are encrypting or decrypting a message, we need to ask the user for which key to use to encrypt or decrypt a file.

Reading From Files

Up until now, any input we want to give our programs would have to be typed in by the user. But if we want to deal with large amounts of text, typing is too slow. Python programs can open and read files off of the hard drive. There are three steps to reading the contents of a file: opening the file, reading the file, and then closing the file.

The open() Function

The open() function has two string parameters. The first string parameter is the name of the file. 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\\thetimemachine.txt' (on Windows) and '/usr/foobar/thetimemachine.txt' (on Mac OS X and Linux) are absolute filenames. (Remember that the \ backslash must be escaped with another backslash before it.)

The second string parameter is the string 'r', which stands for "read". This tells the open() function that you want to read this file, not write to it. This also ensures that you don't accidentally overwrite the file and erase the original contents.

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 (change the directory in the open() call to whichever directory your text file is in):

```
>>> fo = open('c:\\spam.txt', 'r')
>>> content = fo.read()
>>> print(content)
Hello world!
>>>
```

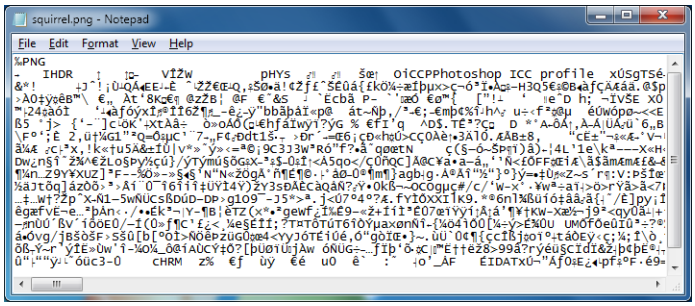
If your text file has multiple lines, the string returned by read() will have \n newline characters in it at the end of each line. When you try to print a string with newline characters, the string will print across several lines:

```
>>> 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.

Note that you will only be able to read text files, but not "binary" files (which include images, movies, MP3s, and other types of files.) You can tell if a file is a binary one by opening it in a

plain text editor such as Notepad or TextMate. If the file has lots of crazy looking characters in it, it is a binary file. If you call the read() method on a binary file that you've opened, you will get a "UnicodeDecodeError" message.



We will not be reading binary files in this book, but if you would like to learn more about text and binary files, go to http://becomeacodebreaker.com/moreinfo.

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()
>>>
```

If you forget to call this, Python will automatically close any open files when the program terminates. But 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.

```
44.             sourcefo = open(sourcefilename, 'r')
45.             sourcecontent = sourcefo.read()
46.             sourcefo.close()
```

Our codebreaker.getFilename() function returned the filename as a string, which we stored in the sourcefilename variable. We want to open this file and store the file object from open() in the sourcefo variable. We then call the read() method on this file object and store the string of the file's content in the sourcecontent variable. Then we close the file object.

The sourcecontent variable now contains the string we want to encrypt or decrypt, whether it is just a few words long or several million words long.

```
48.             if mode == 'ef':
49.                 print('Encrypting file %s...' % (sourcefilename))
50.                 translated = getEncryptedMessage(sourcecontent, key)
```

```
51.             elif mode == 'df':
52.                 print('Decrypting file %s...' % (sourcefilename))
53.                 translated = getDecryptedMessage(sourcecontent, key)
```

Like lines 25 to 30, we want a set of if-elif statements to call either the getEncryptedMessage() or getDecryptedMessage() functions depending on what mode the user requested. Either way, the translated string is stored in the translated variable.

Writing To Files

We read the original file and now will write the translated form to a different file. The file object returned by open() also 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')
>>>
```

Note that if you use a filename of a file that already exists, open() will delete this file and and any strings you write to it will overwrite any content that used to be in the file. We have our codebreaker.getFilename() function do input validation to make sure we don't use an existing filename to prevent accidentally overwriting it with the Transpositional Cipher program.

Along with "read" and "write", there is also an "append" mode. In append mode, you do not overwrite the existing file, but instead any strings you write to the file will be appended to the end of any content that is already in the file. To open a file in append mode, pass the string 'a' as the second argument to open(), like this:

```
>>> fo = open('filename.txt', 'a')
>>>
```

The write() Function

Values of the file object data type have a write() method. It is only valid to call the write() method if the file object was 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. Remember that if you open a file in write mode, it will overwrite any text that was in it. If you wish to change the content

```
 55.             targetfo = open(targetfilename, 'w')
 56.             targetfo.write(translated)
 57.             targetfo.close()
 58.             print('Writing content to file %s is complete.' %
(targetfilename))
```

Now that we have the encrypted/decrypted string in the translated variable, we want to open the target file in "write" mode by passing 'w' to the open() function. On the file object that it returns, we will call the write() method and pass the translated variable to it. Finally, we will close the file object by calling the file object's close() method. The print() call on line 58 will tell the user that the encryption/decryption writing is complete.

Line 58 is the end of the while loop that began on line 11. After executing whatever code is associated with the mode that the user selected, the program execution jumps back to line 11 and re-enters the loop as long as the user had not selected the 'q' quit mode. The program then asks the user if they'd like to encrypt or decrypt anything else.

The rest of transpositional.py implements the various functions that main() calls. Let's look at these functions in detail now.

```
 61. def getKey():
 62.     # This function returns the integer the user typed in for the key.
 63.     while True:
 64.         print('Enter an integer for the key:')
 65.         key = input('> ')
 66.         if key.isdigit() and int(key) > 0:
 67.             return int(key)
```

The getKey() function is similar to the getKey() function we wrote in the Caesar Cipher program. There is an infinite loop on line 63 (we can tell that it is an infinite loop because the condition is merely the True value, meaning that execution will always enter the loop.)

Inside the loop, we ask the user to enter a key to use on line 65. The if statement on line 66 checks that what the user entered is a number that is larger than 0 (because 0 is not a valid key for the Transpositional Cipher.) Remember that short circuit evaluation will prevent the int() function call from throwing an error if the string in key is not a number.

If the key the user entered passes our input validation check on line 66, then we return the integer form of the key as the return value for getKey().

```
 70. def generateRandomKey():
 71.     # This function returns a random integer for the key.
```

```
 72.     return random.randint(2, 100)
```

The generateRandomKey() for the Transpositional Cipher simply returns a random integer between 2 and 100.

Lists

Part of our implementation of the Transpositional Cipher involves a new data type called a list. A list value can contain several other values in it. In Python code, a list starts and ends with the square bracket characters [and]. This is just like how string values begin and end with double quote or single quote characters.

Try typing this into the shell: ['apples', 'oranges', 'HELLO WORLD']. This is a list value that contains three string values. Just like any other value, you can store this list in a variable. Try typing spam = ['apples', 'oranges', 'HELLO WORLD'], and then type spam to view the contents of spam.

```
>>> spam = ['apples', 'oranges', 'HELLO WORLD']
>>> spam
['apples', 'oranges', 'HELLO WORLD']
>>>
```

Lists are a good way to store several different values into one variable. The individual values inside of a list are also called items. Try typing: animals = ['aardvark', 'anteater', 'antelope', 'albert'] to store various strings into the variable animals. The square brackets can also be used to get an item from a list. Try typing animals[0], or animals[1], or animals[2], or animals[3] into the shell to see what they evaluate to.

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

The number between the square brackets is the index. In Python, the first index is the number 0 instead of the number 1. So the first item in the list is at index 0, the second item is at index 1, the

third item is at index 2, and so on. 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 or thousands (or millions or billions) of different strings that you want stored in a list. Using the square brackets, you can treat items in the list just like any other value. Try typing animals[0] + animals[2] into the shell:

```
>>> animals[0] + animals[2]
'aardvarkantelope'
>>>
```

Because animals[0] evaluates to the string 'aardvark' and animals[2] evaluates to the string 'antelope', then the expression animals[0] + animals[2] is the same as 'aardvark' + 'antelope'. This string concatenation evaluates to 'aardvarkantelope'.

What happens if we enter an index that is larger than the list's largest index? Try typing animals[4] or animals[99] into the shell:

```
>>> animals = ['aardvark', 'anteater', 'antelope', 'albert']
>>> animals[4]
Traceback (most recent call last):
File "", line 1, in
animals[4]
IndexError: list index out of range
>>> animals[99]
Traceback (most recent call last):
File "", line 1, in
animals[99]
IndexError: list index out of range
>>>
```

If you try accessing an index that is too large, you will get an index error.

Changing the Values of List Items with Index Assignment

You can also use the square brackets to change the value of an item in a list. Try typing animals[1] = 'ANTEATER', then type animals to view the list.

```
>>> animals = ['aardvark', 'anteater', 'antelope', 'albert']
>>> animals[1] = 'ANTEATER'
>>> animals
['aardvark', 'ANTEATER', 'antelope', 'albert']
>>>
```

The second item in the animals list has now been overwritten with a new string.

for Loops

Instead of looping through a block of code while a condition is True like a while loop does, a for loop will loop through a block of code once for each item in a list. On each iteration through the loop, a variable is assigned the value of an item in the list.

Open the file editor and type in the following short program, and then run it:

```
1. for i in ['aardvark', 'anteater', 'antelope', 'albert']:
2.     print(i)
```

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

```
aardvark
anteater
antelope
albert
```

What the for loop does is on each iteration through the block of code, it assigns the variable i the to a value in the list (starting at the first item and moving through the items in order.) The loop in the above program essentially does the exact same thing as this code:

```
i = 'aardvark'
print(i)
i = 'anteater'
print(i)
i = 'antelope'
print(i)
i = 'albert'
print(i)
```

Or, if you would rather use a while loop, the while loop does the exact same thing as the for loop:

```
index = 0
myList = ['aardvark', 'anteater', 'antelope', 'albert']
```

```
while index < len(myList):
       i = myList[index]
       print(i)
       index = index + 1
```

As you can see, if we want to run a block of code once for each item in a list, then a for loop is much shorter to type than a while loop.

The range() Function

The range() function is often used with for loops. Often you will want a block of code executed a specific number of times. Using a while loop to do this would look like this:

```
i = 0
while i < someNumber:
    doStuff(i)
    i = i + 1
```

The range() function returns a list of integers from 0 to the integer argument you pass to range. For example, if you call range(10), the call evaluates to a list value of ten integers: [0, 1, 2, 3, 4, 5, 6, 7, 8, 9]. This can be useful if you want to use a for loop to execute a block of code ten times:

```
for i in range(10):
    doStuff(i)
```

What's more, the value i in this loop will take start at 0 and increase by 1 on each iteration. Open the file editor and type in the following short program, then run it:

```
for i in range(5):
    print('Iteration #%s' % (i))
```

Just imagine how range(5) evaluates to the list value [0, 1, 2, 3, 4]. When we run this program, the output will be:

```
Iteration #0
Iteration #1
Iteration #2
Iteration #3
Iteration #4
```

You can specify two integer arguments to the range function to specify a starting integer besides 0. So while range(10) will return a list of integers 0 to 9, calling range(4, 10) will return a list of

integers 4 to 9. The returned list always includes the first argument (in the previous case, 4) but does not include the second argument (in the previous case, 10). Think of range(8) as always having a 0 for the first argument of a two-argument form: range(0, 8).

You can also specify three integers. The first is the starting integer, the second is the ending integer, and the third integer is how much the integers in the list should increment. For example, range(0, 10, 1) would return the list [0, 1, 2, 3, 4, 5, 6, 7, 8, 9]. But the call range(0, 10, 2) would increment the integers by 2 instead of 1: [0, 2, 4, 6, 8]. The call to range(4, 10, 3) would evaluate to the list value [4, 7]. Just like range(8) is really the same as range(0, 8), you can think of range(0, 8) as being the same as range(0, 8, 1).

Here is a for loop that uses the three-argument form:

```
for i in range(4, 20, 3):
    doStuff(i)
```

The while loop equivalent of this for loop would look like this:

```
i = 4
while i < 20:
    doStuff(i)
    i = i + 3
```

There is one thing you should know though. In Python 2, the range() function returns a list value. But in Python 3 (the version this book covers) the range() function technically does not return a list value. The value it returns is a "range object", but the for statement treats a range object just like a list value. Most of the time that you call a range() function it will be for a for loop, so this "range object" difference doesn't matter.

But if you ever need a list of integers value, pass the range object to the list() function. Just like the int() and str() functions return the integer and string data type value of the argument, list() returns the list data type value of the argument. So if you want the list value [0, 1, 2, 3], call list(range(4)). But remember, the for statement will always treat the range object as the same as a list value. If you would like to know more about why this "range object" stuff exists, you can read more at http://becomeacodebreaker.com/moreinfo.

The Encryption Algorithm

The Transpositional Cipher seems simple to implement with a pencil and paper. We draw boxes and write out the message across the rows, and then read the ciphertext message down the columns. But how do we translate this 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 (note that there are 8 columns, and the (s) marks where a space character goes):

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	.		

Let's replace each of the letters in the boxes with its index in the string. Remember that indexes begin with 0, not 1.

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

There is an exception for the 6^{th} and 7^{th} 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.

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 until the index is larger than 30 (the message length), at which point we then move on 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. (Our list will have 8 blank strings since we are using the key 8 in our example.) Let's look at the code.

```
75. def getEncryptedMessage(message, key):
76.     # This function returns a string that is the encrypted version of the
77.     # string in the message parameter, using the key of the key parameter.
78.     ciphertext = [''] * key
```

Our getEncryptedMessage() function will be given two parameters: message (which contains the string value to be encrypted) and key (which contains the integer key to be used for encrypting).

The ciphertext variable will contain the list of blank strings that we will add the columns of letters to. The ['] * key uses "list replication" to create a list of a number of blank strings equal to the integer in key.

```
79.     for col in range(key):
80.         pointer = col
81.         while pointer < len(message):
82.             ciphertext[col] += message[pointer]
83.             pointer += key
84.     return ''.join(ciphertext)
```

The for loop on line 79 starts a variable named col at 0, and on each iteration increments it by 1 until it reaches 1 less than key. (This is because the range() function goes from 0 and up to, but not including, the value we passed it, in this case, key.) When col is set to 0, we will fill the 0^{th} column (which is, in the case of our program, the list at ciphertext[0].) The code inside the for loop handles this filling up.

Then on the next iteration, the value in col will be set to 1 and we will fill up the string at ciphertext[1]. The iteration after that, col will be set to 2, and so on.

Each iteration through the for loop starts at line 80. We set a variable called pointer to col to begin with. Then, we enter a new loop: the while loop on line 81. As long as the integer in pointer is less than the length of message (that is, the number of characters in the string in message), the loop will execute lines 82 and 83.

Line 82 does the actual "filling up". It takes the character at the index pointer from message and concatenates it to the end of the string at ciphertext[col]. Line 83 then increases the pointer variable by the value in key. This keeps happening until an execution of line 83 increases the pointer variable to be equal or larger than the length of the message string, which will make the while loop's condition False.

This loop-within-a-loop can be hard to understand. Let's use our example of a key value of 8 and a message value of "Common sense is not so common." (This means that len(message) will

always return the integer value 30, since the string in message has 30 characters counting the spaces and period.)

The for loop will start col at 0, which is also where line 80 will start the pointer variable. Then message[pointer] (which is message[0] since pointer is set to 0) will be added to the string at ciphertext[0]. The variable pointer is then incremented by 8 on line 83 (since key is set to 8). This happens over and over again.

In order, the characters at message[0], message[8], message[16], and message[24] are added to ciphertext[0]. After that, when we add 8 to pointer (which was last set to 24), pointer will be equal to 32, which is larger than len(message). This means the execution exits the while loop and then we go on the next iteration of the for loop, which sets col to 1.

When col is 1, the pointer variable is first set to 1. The while loop will then add the following to ciphertext[1]: message[1], message[9], message[17], and message[25]. This follows the same pattern as the boxes of numbers: 0, 8, 16, 24 for the first column, 1, 9, 17, 25 for the second, then 2, 10, 18, 26 for the third column, and so on.

When the for loop is finally finished, the ciphertext list will have a value similar to our box of letters we drew out earlier: ['Ceno', 'onom', 'mstm', 'me o', 'o sn', 'nio.', ' s ', 's c'] The join() string method call on line 84 can join this list of strings into a single string value: 'Cenoonommstmme oo snnio. s s c' This is the encrypted ciphertext version of our message, and so the getEncryptedMessage() function returns this string as the function's return value.

The Decryption Algorithm

The first few steps of the decryption process translate easily into code. First we need to calculate the number of columns, which is the number of characters in the message divided by the key, and then rounded up. The math.ceil() function in the math module takes a number value and returns the rounded up number, so we will pass the expression len(message) / key to it. The number of rows is simply the same as the key.

```
87. def getDecryptedMessage(message, key):
88.     # This function returns a string that is the decrypted version of the
89.     # string in the message parameter, using the key of the key parameter.
90.     numOfColumns = math.ceil(len(message) / key)
91.     numOfRows = key
```

Now we need to the function to calculate how many shaded boxes there will be. Here's a picture of our "common sense" decryption example that was shown earlier, but now with the rows and columns numbered:

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

Notice that in the above example, we have 4 columns and 8 rows, which means we have 32 boxes. 2 of the boxes are shaded in, because the message only has 30 characters. That means if we want to calculate the number of shaded boxes there will be, we just multiply the rows and columns, and then subtract the length of the message. This is what we do on line 92.

```
92.     numOfShadedBoxes = numOfColumns * numOfRows - len(message)
```

Our program won't be drawing out boxes and then writing numbers in them, but it will imitate this behavior with a list of strings stored in a variable named plaintext.

```
93.     plaintext = [''] * numOfColumns
```

The number of strings in this list will be the same as the number of columns we calculated. All of the strings will start as blank strings. Line 93 uses list replication, where the multiplication operator creates a list made up of repeats of the list in the expression. For example, ['hello', 42] * 3 would evaluate to the list, ['hello', 42, 'hello', 42, 'hello', 42].

Each string will represent a column in our box-drawing example. The string at plaintext[0] will represent the first column, column 0. The string at plaintext[1] will represent column 1, and so on. As we go through the letters of the ciphertext message, we will add them to them to the next column's string, until we get past the last column where we will go around back to the first column.

For example, with our "common sense" example, the ciphertext is 'Cenoonommstmme oo snnio. s s c'. The key is 8, which means numOfColumns will be 4 (message length divided by key, rounded up), and plaintext will be set to [''] * 4, which evaluates to ['', '', '', ''].

Below shows the manual box-drawing steps of the decryption process and also what the value of plaintext will be at that same step in the programming decryption process:

	0	1	2	3

0	C			
1				
2				
3				
4				
5				
6				
7				

plaintext = ['C', '', '', '']

	0	1	2	3
0	C	e		
1				
2				
3				
4				
5				
6				
7				

plaintext = ['C', 'e', '', '']

	0	1	2	3
0	C	e	n	o
1	o			
2				
3				
4				
5				
6				
7				

plaintext = ['Co', 'e', 'n', 'o']

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

7	s	(s)	c	

plaintext = ['Common s', 'ense is ', 'not so c', 'ommon.']

```
 95.     col = 0
 96.     row = 0
 97.     for i in range(len(message)):
 98.         plaintext[col] += message[i]
 99.         col += 1
100.         if col == numOfColumns or (col == numOfColumns - 1 and row >=
numOfRows - numOfShadedBoxes):
101.             col = 0
102.             row += 1
103.     return ''.join(plaintext)
```

Notice that if we tried to decrypt the message with the wrong key, the wrong message would be the result. For example, using the key 7 instead of 8, the boxes would 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	

And the plaintext would come out as 'Cnt so eomon.snmmon ome iscos'. LEFT OFF

```
112. if __name__ == '__main__':
113.     main()
```

The codebreaker Module, Version 1

```
1. # Codebreaker Utility Module, http://becomeacodebreaker.com
2. # version 1.0
3.
4. import os
5.
6. version = 1.0
```

```
9. def getMode(modeTypes, clipboardText):
10.     # This function returns 'e', 'd', 'k', 'c', 'q', 'ef', or 'df'
11.     # depending on what the user enters.
12.     if len(clipboardText) > 10:
13.         clipboardText = clipboardText[:10] + '...'
```

```
15.     modeShortNames = ['e', 'ef', 'd', 'df', 'k', 'c', 'q']
16.     modeNames = [['e', 'Encrypt Message'],
17.                  ['ef', 'Encrypt File'],
18.                  ['d', 'Decrypt Message'],
19.                  ['df', 'Decrypt File'],
20.                  ['k', 'Generate Key'],
21.                  ['c', 'Copy "' + clipboardText + '" to clipboard'],
22.                  ['q', 'Quit']]
23.     mode = ''
24.     while mode not in modeShortNames:
25.         for modeName in modeNames:
26.             if modeName[0] + ' ' in modeTypes:
27.                 extraSpace = ''
28.                 if len(modeName[0]) == 1:
29.                     extraSpace = ' '
```

```
30.                 print('%s%s - %s' % (modeName[0], extraSpace, modeName[1]))
```

```
32.         # get choice from user
33.         mode = input('> ').lower()
34.     print()
35.     return mode
```

```
38. def getMessage():
39.     # This function returns the string of the message the player typed in.
40.     print('Enter your message:')
41.     return input('> ')
```

```
44. def getFilename(directions, fileRequirement):
45.     while True:
46.         print(directions)
47.         print('Or enter CANCEL to quit.')
48.         filename = input('> ')
49.         print()
50.         if filename.upper() == 'CANCEL':
51.             return
52.         if fileRequirement == 'exists' and not os.path.exists(filename):
53.             print('Could not find the file %s' % (filename))
54.             continue
55.         elif fileRequirement == 'does not exist' and
os.path.exists(filename):
56.             print('That file already exists.')
57.             continue
58.         return filename
```

Transpositional Cipher Unit Test

It is easy to try out our cipher program after we write it to see if it works. But sometimes bugs can be hard to find because they only appear under certain rare circumstances. For example, what if

(for whatever reason) our crypto program worked correctly for keys less than 100, but was broken if the user tried a key above 100? Or what if the program could encrypt some messages but couldn't encrypt others? You may not find these bugs for a long time, and well after you've given them to other people to use.

It would take a lot of work to keep checking several different messages and keys by typing them in one at a time. But you can actually write a program to test your programs for you. This way, if you do find a bug with your cryptography program and fix it, you can re-test all the different keys and messages just by running a program.

These testing programs are called unit tests. We can write a unit test for our Transpositional cipher program. It simply needs to try to encrypt different messages with different keys, and then decrypt them. If the decrypted text matches the original text we encrypted, then we can be more certain the transpositional cipher program is free of bugs.

Here is the source code for our unit test program. Notice that it imports our transpositional.py file so it can call its getEncryptedMessage() and getDecryptedMessage() functions. This unit test won't test the other parts of our transpositional program, but the encryption and decryption functions are the most complicated part of the program and the most likely place where bugs will be.

Chapter 10 - Symbols, Encodings, and Data

When people want to store information to remember later, such as a grocery list, they write it down on paper with a pencil. Or if they don't have paper and pencil, maybe they write it on a napkin with a marker. There are many ways a person can store information by writing it down, but it always involves marking letters on something.

There are several different ways a computer can store information to remember later. The computer doesn't mark letters on paper though, instead it only makes a mark one way or another. One type will represent the number 0 and the other type will represent the number 1. The hard disk drive has billions of tiny bits that are magnetized one way to represent 1 and another to represent 0. DVDs have billions of tiny bits that are shaped one way to represent 0 and another to represent 1. Thumb drives have billions of tiny electric fields that are charged one way to present 1 and another to represent 0. This lets these devices store billions of bytes of data. Using just 1s and 0s, the computer can represent any number, string of text, or any digitized data.

Binary Numbers

A computer can use the numerals 0 and 1 to represent any number, text, songs, computer games, and data. First we will learn how you can represent any number with just 0 and 1.

The first ten integers starting from 0 are 0, 1, 2, 3, 4, 5, 6, 7, 8, and 9. When you get to the integer after 9, we do not have a single numeral that represents "ten". Instead we use the "one" and "zero" characters: 10. This is the decimal (also called "base ten") numeral system, and it is the one we are used to.

The binary (also called "base two") numeral system only has 0 and 1 for numerals. The number zero is 0 in both decimal and binary. The next number after zero is one, and is written 1 in both decimal and binary also. Just like how the decimal system runs out of numerals at ten and is forced to reuse numerals 10, the binary system runs out of numerals at two, and so it writes that number as 10. The binary number three is written as 11.

Here is a table that shows how the equivalent numbers are written in decimal and binary:

Number	Decimal (base 10)	Binary (base 2)
Zero	0	0
One	1	1
Two	2	10
Three	3	11
Four	4	100
Five	5	101
Six	6	110
Seven	7	111
Eight	8	1000
Nine	9	1001
Ten	10	1010
Eleven	11	1011
Twelve	12	1100
Thirteen	13	1101
Fourteen	14	1110
Fifteen	15	1111
Sixteen	16	10000
Seventeen	17	10001
Eighteen	18	10010
Nineteen	19	10011
Twenty	20	10100

Convert Between Decimal to Binary

Converting a decimal number to binary or a binary number to decimal is easy to learn how to do. If you are interested, please see http://becomeacodebreaker.com/moreinfo.

However, you can havy Python do it for you. Simply call the bin() function and pass the integer (which will be a decimal) to it. The binary form will be returned as a string with the '0b' prefix in front of it (to tell you that it is a binary number).

```
>>> bin(42)
'0b101010'
>>> bin(0)
'0b0'
>>> bin(1000000)
'0b11110100001001000000'
>>>
```

To convert a string with the binary form of a number to decimal, call the int() function with the integer 2 as a second argument. This will tell the int() function that you are passing it a base-2 (binary) number. The string you pass may or may not have the '0b' prefix in front.

```
>>> int('0b101010', 2)
42
>>> int('101010', 2)
42
>>> int('11110100001001000000', 2)
1000000
>>>
```

So the int() function can convert not only decimal numbers in a string to integer values, but also binary numbers in a string to integer values.

Bits and Bytes

Each binary 1 or 0 that the computer stores is a bit, which is a short form of binary digit. A byte is made of eight bits. A gigabyte is about a billion bytes (actually a gigabyte is a little bit more than one billion: 1,073,741,824 or 2^30 bytes). So a hard disk drive that can hold 100 gigabytes can store about 800 billion bits.

Hexadecimal Numbers

You can create a numeral system with any base. But base-10 is what we humans are used to dealing with, and base-2 is the base that computers deal with (though it gets converted to base-10 when it is displayed to us). Another base that is often used in programming is hexadecimal, or base-16. While the base-2 binary numeral system has 8 fewer numbers than the base-10 decimal numeral system, the base-16 hexadecimal system has 6 more numbers than base-10.

Remember how in decimal, there were no more numbers after 9, so we had to use two numerals for the number ten? In hexadecimal, there is a single numeral for ten: A. There is also a single numeral for the numbers eleven through fifteen: B, C, D, E, and F. So the numerals for hexadecimal go from 0 to F, instead of just 0 to 9.

Here is a table that shows how the equivalent numbers are written in decimal, binary, and hexadecimal:

Number	Decimal (base 10)	Binary (base 2)	Hexadecimal (base 16)
Zero	0	0	0
One	1	1	1

Two	2	10	2
Three	3	11	3
Four	4	100	4
Five	5	101	5
Six	6	110	6
Seven	7	111	7
Eight	8	1000	8
Nine	9	1001	9
Ten	10	1010	A
Eleven	11	1011	B
Twelve	12	1100	C
Thirteen	13	1101	D
Fourteen	14	1110	E
Fifteen	15	1111	F
Sixteen	16	10000	10
Seventeen	17	10001	11
Eighteen	18	10010	12
Nineteen	19	10011	13
Twenty	20	10100	14

In Python, hexadecimal numbers are strings that begin with the prefix, “0x”. (Just like binary numbers begin with the prefix “0b”.) You can use the hex() function to convert decimal integer values to hexadecimal string values:

```
>>> hex(2)
'0x2'
>>> hex(42)
'0x2a'
>>> hex(100)
'0x64'
>>> hex(1000000)
'0xf4240'
>>>
```

To convert from hexadecimal string values to decimal integer values, use the int() function but pass 16 as the second argument:

```
>>> int('0x2', 16)
2
>>> int('0X2A', 16)
42
>>> int('0x64', 16)
100
```

```
>>> int('0xf4240', 16)
1000000
>>>
```

The reason hexadecimal is useful in programming is because binary numbers are usually very long, but one hexadecimal digit can represent four binary digits perfectly. So if we had a binary number like this:

```
110011110110001101011100
```

That same number could be represented as a much shorter hexadecimal number like this:

```
CF635C
```

You can convert binary to hexadecimal by calling hex() on the return value of int():

```
>>> hex(int('11001111011000110101110010000111', 2))
'0xcf635c'
>>>
```

You can convert hexadecimal to binary by calling bin() on the return value of int():

```
>>> bin(int('0xcf635c', 16))
'0b110011110110001101011100'
>>>
```

A single byte is eight bits, which means a single byte can be represented with two hexadecimal digits.

Computers Store Binary Numbers

Whether on a hard drive, thumb drive, DVD, or in memory, computers only store 1s and 0s. So all of your text files, MP3s, games, and photos are really stored in binary.

Encoding: Digitizing Information into Binary

Storing a Picture as Bytes

Storing Text as Bytes

The ASCII Encoding

The chr() and ord() Functions

Hexadecimal Numbers

Hex Editors

Binary Data vs. Text Data

Converting Binary to Text with Base64 Encoding

Opening Files in Binary Mode

Chapter 12 – Detecting English Programmatically

A message encrypted with the transpositional cipher can have thousands or millions of possible keys. Your computer can still easily try out this many keys, but you would then have to look through thousands or millions of decryptions to find the one correct plaintext. This is a big problem for the brute force method of cracking the transpositional 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 millions 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 is 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-386. He will try to eat you.

ai-pey  e. xrx ne augur iirl3 Rtiyt fhubE6d hrei 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 words is easy. There is already a Python function named split() that will do this for us. It just sees when each word begins or ends by looking for the spaces. Once we have the individual words, we can test to see if it 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. So we just need to write a function that checks if the words in the string exist somewhere in that file.

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 decryptions that might have non-English words in them, such as "RX-386" in our above English sentence. (Or maybe the plaintext is in a different language besides English. But we'll just assume it is for now.) And garbage text might just happen to have an English word or two in it by coincidence. 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 full proof. But if most of the words in the passed string 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.

So our function will have to split up a string into words, check if each word is in a file full of thousands of English words, and if the number of English words is more than the number of non-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.

The append() List Method

The most common list method you will use is append(). This method will add the value you pass as an argument 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]
>>>
```

List Concatenation

You can join lists together into one list with the + operator, just like you can join strings. When joining lists, this is known as list concatenation. Try typing [1, 2, 3, 4] + ['apples', 'oranges'] + ['Alice', 'Bob'] into the shell:

```
>>> [1, 2, 3, 4] + ['apples', 'oranges'] + ['Alice', 'Bob']
[1, 2, 3, 4, 'apples', 'oranges', 'Alice', 'Bob']
>>>
```

Notice that lists do not have to store values of the same data types. The example above has a list with both integers and strings in it. Remember, when you do list concatenation, you must add together two list values. ['apples'] + ['oranges'] will evaluate to ['apples', 'oranges']. But ['apples'] + 'oranges' will result in an error because you are adding a list value and string value instead of two list values. If you want to add non-list values to a list, use the append() method (which is described later).

The in Operator

The in operator makes it easy to see if a value is inside a list or not. Expressions that use the in operator return a Boolean value: True if the value is in the list and False if the value is not in the list. Try typing 'antelope' in animals into the shell:

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

The expression 'antelope' in animals returns True because the string 'antelope' can be found in the list, animals. (It is located at index 2.)

But if we type the expression 'ant' in animals, this will return False because the string 'ant' does not exist in the list. We can try the expression 'ant' in ['beetle', 'wasp', 'ant'], and see that it will return True.

```
>>> animals = ['aardvark', 'anteater', 'antelope', 'albert']
>>> 'antelope' in animals
True
>>> 'ant' in animals
False
>>> 'ant' in ['beetle', 'wasp', 'ant']
True
```

```
>>>
```

The in operator also works for strings as well as lists. You can check if one string exists in another the same way you can check if a value exists in a list. Try typing 'hello' in 'Alice said hello to Bob.' into the shell. This expression will evaluate to True.

```
>>> 'hello' in 'Alice said hello to Bob.'
True
>>>
```

Removing Items from Lists with del Statements

You can remove items from a list with a del statement. ("del" is short for "delete.") Try creating a list of numbers by typing: spam = [2, 4, 6, 8, 10] and then del spam[1]. Type spam to view the list's contents:

```
>>> spam = [2, 4, 6, 8, 10]
>>> del spam[1]
>>> spam
[2, 6, 8, 10]
>>>
```

Notice that when you deleted the item at index 1, the item that used to be at index 2 became the new value at index 1. The item that used to be at index 3 moved to be the new value at index 2. Everything above the item that we deleted moved down one index. We can type del spam[1] again and again to keep deleting items from the list:

```
>>> spam = [2, 4, 6, 8, 10]
>>> del spam[1]
>>> spam
[2, 6, 8, 10]
>>> del spam[1]
>>> spam
[2, 8, 10]
>>> del spam[1]
>>> spam
[2, 10]
>>>
```

Just remember that del is a statement, not a function or an operator. It does not evaluate to any return value.

Lists of Lists

Lists are a data type that can contain other values as items in the list. But these items can also be other lists. Let's say you have a list of groceries, a list of chores, and a list of your favorite pies. You can put all three of these lists into another list. Try typing this into the shell:

```
>>> groceries = ['eggs', 'milk', 'soup', 'apples', 'bread']
>>> chores = ['clean', 'mow the lawn', 'go grocery shopping']
>>> favoritePies = ['apple', 'frumbleberry']
>>> listOfLists = [groceries, chores, favoritePies]
>>> listOfLists
[['eggs', 'milk', 'soup', 'apples', 'bread'], ['clean', 'mow the lawn', 'go
grocery shopping'], ['apple', 'frumbleberry']]
>>>
```

You could also type the following and get the same values for all four variables:

```
>>> listOfLists = [['eggs', 'milk', 'soup', 'apples', 'bread'], ['clean', 'mow
the lawn', 'go grocery shopping'], ['apple', 'frumbleberry']]
>>> groceries = listOfLists[0]
>>> chores = listOfLists[1]
>>> favoritePies = listOfLists[2]
>>> groceries
['eggs', 'milk', 'soup', 'apples', 'bread']
>>> chores
['clean', 'mow the lawn', 'go grocery shopping']
>>> favoritePies
['apple', 'frumbleberry']
>>>
```

To get an item inside the list of lists, you would use two sets of square brackets like this: listOfLists[1][2] which would evaluate to the string 'go grocery shopping'. This is because listOfLists[1] evaluates to the list ['clean', 'mow the lawn', 'go grocery shopping'][2]. That finally evaluates to 'go grocery shopping'.

Here is another example of a list of lists, along with some of the indexes that point to the items in the list of lists named x. The red arrows point to indexes of the inner lists themselves. The image is also flipped on its side to make it easier to read:

TODO – split() and append() ?

Source Code for the Codebreaker Module, Version 2.0

We are going to make some additions to the codebreaker.py module. First, at the top of the file, change the version to 2.0 and add a new variable called englishWords to line 7:

```
6. version = 2.0
7. englishWords = None
```

The englishWords variable will store a list of English words. But when the codebreaker module is first run, we simply assign the value None to it. Starting on line 60, add the following lines:

```
 60. # Added in Version 2.0:
 61. def loadDictionary(filename):
 62.     # This function fills the global englishWords variable with all the
words
 63.     # from the dictionary file. This function only returns None.
 64.     global englishWords # in this function, englishWords is the global
variable
 65.     fp = open(filename)
 66.
 67.     englishWords = fp.readlines()
 68.     # englishWords is a list of strings, one string per line in the file
 69.
 70.     # remove the newline character at the end of each string in
englishWords
 71.     for i in range(len(englishWords)):
 72.         englishWords[i] = englishWords[i].strip()
 73.     fp.close()
 74.
 75.
 76. def isEnglishWord(word):
 77.     # This function returns True if "word" is in the dictionary, otherwise
 78.     # it returns False.
 79.
 80.     global englishWords # in this function, englishWords is the global
variable
 81.
 82.     # If the dictionary hasn't been loaded yet, do it now.
 83.     if englishWords == None:
 84.         loadDictionary('dictionary.txt')
 85.
 86.     # strip out non-letter characters
 87.     word = word.lower()
 88.     lettersOnlyWord = ''
 89.     for letter in word:
 90.         if letter in string.ascii_lowercase:
```

```
 91.             lettersOnlyWord += letter
 92.
 93.     return word in englishWords
 94.
 95.
 96. def getEnglishCount(message):
 97.     # Returns the amount of words in message that appear in the english
dictionary.
 98.     message = message.lower()
 99.
100.     # remove all non-letters and non-whitespace from message
101.     justLetters = ''
102.     for i in range(len(message)):
103.         if message[i] in string.ascii_lowercase or message[i] in (' ',
'\t', '\n'):
104.             justLetters += message[i]
105.
106.     # split up the message into words
107.     words = justLetters.split()
108.
109.     # Go through each word and see how many are english words.
110.     matches = 0
111.     for word in words:
112.         if isEnglishWord(word):
113.             matches += 1
114.     return matches / len(words)
115.
116.
117. def isEnglish(message, thresholdPercent):
118.     # Returns True if the percentage of words in the message parameter
119.     # is equal or greater than the thresholdParameter.
120.     # Otherwise, returns False.
121.     # thresholdParameter should be between 0 and 100.
122.     if englishWords == None:
123.         loadDictionary('dictionary.txt')
124.     return (100 * getEnglishCount(message)) >= thresholdPercent
```

Reading in the Dictionary File

```
 61. def loadDictionary(filename):
 62.     # This function fills the global englishWords variable with all the
words
 63.     # from the dictionary file. This function only returns None.
 64.     global englishWords # in this function, englishWords is the global
variable
```

```
 65.     fp = open(filename)
 66.
 67.     englishWords = fp.readlines()
 68.     # englishWords is a list of strings, one string per line in the file
```

The loadDictionary() function takes a parameter that is the filename of our dictionary file. The dictionary file is simply a plain text file that has one English word per line. (You can look at this dictionary file in any text editor program.)

The codebreaker module will have a global variable named englishWords which will be a list of several thousands of strings, one string for each of the words in the dictionary file. We can't just type in this list manually into codebreaker.py, because it would be too big. Instead, line 65 opens the dictionary file, line 66 returns a list of strings where each string is a single line in the file. We are done reading the dictionary file, so line 67 closes the file by calling the close() method on the file object. (Remember to call close() on the file object, not the string of the filename.)

```
 70.     # remove the newline character at the end of each string in
englishWords
 71.     for i in range(len(englishWords)):
 72.         englishWords[i] = englishWords[i].strip()
 73.     fp.close()
```

The split() Method

The split() method, which is a method for the string data type (just like the lower() and upper() methods). The split() 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 nine pies'.split()
['My', 'very', 'energetic', 'mother', 'just', 'served', 'us', 'nine', 'pies']
>>>
```

The result is a list of nine 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). Once we've called split(), the words list will contain all the possible secret words that can be chosen by the computer for our Hangman game. You can also add your own words to the string, or remove any you don't want to be in the game. Just make sure that the words are separated by spaces.

You can pass an optional argument to the split() method to tell it to split on a different string other than just a space. For example:

```
>>> 'helloXXXworldXXXhowXXXareXXyou?'.split('XXX')
['hello', 'world', 'how', 'areXXyou?']
>>>
```

TODO – slicing, and comparing lists to strings.

Chapter 13 - Breaking the Transpositional Cipher

To break the transpositional cipher, we will use a brute force approach. We will use the English detection code we developed in the last chapter to have the program realize when it has found the correct key.

Source Code of the Transpositional Cipher Breaker

Chapter 14 - The Simple Substitution Cipher

The transpositional cipher had a larger number of possible keys, but a computer can still easily go through all of them. We'll need a cipher that has so many possible keys, that no computer can possibly brute force through them all.

The simple substitution cipher is effectively invulnerable to a brute force attack. Even if your computer could try out a trillion keys every second, it would still take twelve million years to for it to try out every key.

To implement the Caesar Cipher, choose a random letter to encrypt each letter of the alphabet to. Use each letter once and only once.

There are 403,291,461,126,605,635,584,000,000 possible keys for the simple substitution cipher. To see how this number was calculated, see http://becomeacodebreaker.com/moreinfo)

Chapter 15 - Breaking the Simple Substitution Cipher

The simple substitution cipher is immune to any brute force attack we could do on it. We're going to have to employ a more intelligent attack if we want to crack a simple substitution ciphertext.

Let's examine one word from a simple substitution ciphertext: HGHHU. What can we learn from this one word of ciphertext?

First, we know that just about every word in the English language has a vowel in it: a, e, i, o, u (and sometimes y). So at least one of the letters H, G, or U must be the ciphertext for a vowel letter.

Second, we also know that whatever the original word is, it must be five letters long, because the simple substitution cipher replaces one letter with one and only one letter.

In fact, if we think about it, the first, third, and fourth letters of the original word must all be the same. And whatever it is, it must be different from the second and fifth letter in the word. (Otherwise the ciphertext word would be HHHHU or HGHHH.)

So the original word must be five letters long, use exactly three different letters, and at least one of those letters is a vowel. What words in the English language fit this pattern?

"Daddy" is one. It is five letters long using three different letters in that same pattern. "Mommy" works too. And also the name "Bobby" and the words "puppy", "nanny", and "lilly". If we had a lot of time on our hands, we could go through the entire dictionary and find all the words that fit this pattern.

But that would take a while. Instead we could have the computer go through the dictionary and find this out for us. Let's call the pattern that the words "daddy", "puppy", and others is "ABAAC". Every time there is a new letter in the ciphertext word, we use the next letter of the alphabet in the pattern. So "emergency" is "ABACDAEFG", "kooky" is "ABBAC", and "assistant" is "ABBCBDAED".

If we find the pattern for every word in the dictionary and sort them in a list, it will be easy to find all the possible words for a given ciphertext word.

Chapter 16 – A Simple Substitution Breaker Tool

Chapter 17 - The Vigenere Cipher

The Vigenere cipher is a stronger cipher than the ones we've seen before. There are too many possible keys to brute force, even with English detection. It cannot be broken with the word pattern attack that worked on the simple substitution cipher. It was first described in 1553, and remained unbroken until Charles Babbage broke it in the 19th century. It was called "le chiffre indéchiffrable", French for "the indecipherable cipher".

The Vigenere cipher is similar to the Caesar Cipher, except with multiple keys. Because it uses more than one set of substitution, it is also called a polyalpabetic substitution cipher. Remember that the Caesar Cipher had a key from 0 to 25. For the Vigenere cipher, instead of using a numeric key, we will use a letter key. The letter "A" will be used for key 0. The letter "B" will be used for key 1, and so on up to "Z" for the key 25.

And instead of just one key, we will have multiple subkeys. If we use a Vigenere key of "PIZZA", then the first subkey is "P", the second subkey is "I", the third and fourth subkeys are both "Z" and the fifth subkey is "A". We will use the first subkey to encrypt the first letter of the plaintext, and the second subkey to encrypt the second letter, and so on. When we get to the sixth letter of the plaintext, we will go back to using the first subkey.

The following shows which subkey will be used to encrypt the message, "Common sense is not so common." with the Vigenere key, "PIZZA".

```
COMMONSENSEISNOTSOCOMMON
PIZZAPIZZAPIZZAPIZZAPIZZ
```

To encrypt the first "C" with the subkey P, encrypt it with the Caesar Cipher using numeric key 15 (the number for the letter "P") which creates the ciphertext R. Do this for each of the letters of the plaintext. The following table shows this process:

Plaintext Letter	Subkey	Numeric Subkey	Ciphertext Letter
C	P	15	R
O	I	8	W
M	Z	25	L
M	Z	25	L
O	A	0	O
N	P	15	C
S	I	8	A
E	Z	25	D
N	Z	25	M
S	A	0	S
E	P	15	T
I	I	8	Q
S	Z	25	R
N	Z	25	M
O	A	0	O
T	P	15	I
S	I	8	A
O	Z	25	N
C	Z	25	B
O	A	0	O
M	P	15	B
M	I	8	U
O	Z	25	N
N	Z	25	M

So using the Vigenere cipher with the key “pizza”, the plaintext “Common sense is not so common.” becomes the ciphertext “Rwlloc admst qr moi an bobunm.”

Using the Vigenere cipher is just like using the Caesar Cipher, except we use multiple keys instead of just one key. And the more letters in the Vigenere key, the stronger the encrypted message will be against a brute force attack. The choice of “pizza” is a poor one for a Vigenere key, because it only has five letters. A key with only five letters has 11,881,376 possible combinations. (26^5 = 26 x 26 x 26 x 26 x 26 = 11,881,376) Eleven million keys is far too many for a human to try out, but a computer could try them all in a few hours. It would first try to decrypt the message with the key “AAAAA” and check if the resulting decryption was in English. Then it could try “AAAAB”, then “AAAAC”, until it got to “PIZZA”.

The good news is that for every additional letter the key has, the number of possible keys multiplies by 26. Once there are quadrillions of possible keys, it would take a computer years to break. Here is a table that shows how many possible keys there are for each length:

Key Length	Equation	Possible Keys
1	26	26
2	26 x 26	676
3	676 x 26	17,576
4	17,576 x 26	456,976
5	456,976 x 26	11,881,376
6	11,881,376 x 26	308,915,776
7	308,915,776 x 26	8,031,810,176
8	8,031,810,176 x 26	208,827,064,576
9	208,827,064,576 x 26	5,429,503,678,976
10	5,429,503,678,976 x 26	141,167,095,653,376
11	141,167,095,653,376 x 26	3,670,344,486,987,776
12	3,670,344,486,987,776 x 26	95,428,956,661,682,176
13	95,428,956,661,682,176 x 26	2,481,152,873,203,736,576
14	2,481,152,873,203,736,576 x 26	64,509,974,703,297,150,976

Once we get to keys that are twelve letters long, then it becomes impossible for most computers to crack in a reasonable amount of time. But remember, someone could also buy several computers that try different keys to share the work. And computers also get twice as fast about every two years. This is called Moore's Law, and it has held true every two years since 1958.

So today in 2011 (when this book was written) a computer might not be powerful enough to break a ciphertext encrypted with a 12 letter Vigenere key. But in 2013 computers will be twice as fast as they are now. And in 2015 they will be four times as fast. In 2017 then will be eight times as fast as today. So in 2031, the average computer will be over a thousand times as power as computers today (if Moore's Law still remains true.) Depending on how long you want your messages to remain secret, you should consider using a longer key.

A Vigenere key does not have to be a word like "pizza". It can be any combination of letters, such as "duriwknmfick". In fact, it is much better to not use a word that can be found in the dictionary. The word "radiologists" is a twelve letter key that is easier to remember than "duriwknmfick" even though they have the same number of letters in them. But a cryptanalyst might anticipate that the cryptographer is being lazy by using an easy to remember word for the Vigenere key. There are 95,428,956,661,682,176 possible 12-letter keys, but there are only about 45,000 12-letter words in the dictionary. If you are using a 12-letter English word, it would be easier to brute force that ciphertext than it would be brute force a 4-letter random key.

Of course, the cryptographer is helped by the fact that the cryptanalyst does not know how many letters long the Vigenere key is. But the cryptanalyst could try all 1-letter keys, then all 2-letter keys, and so on. In fact, trying every key that is 1, 2, 3, 4 and 5-letter key is much easier than trying every 6-letter key. If you do the math, 26 + 676 + 17576 + 456,976 + 11,881,376 = 12,356,630. But 12,356,630 is still much less than the 308,915,776 possible 6-letter keys.

We will explain these attacks in more detail in chapters 16 and 17. But for now, let's look at the program that can implement the Vigenere cipher.

Source Code of Vigenere Cipher

The Unbreakable One-Time Pad Cipher

There is one cipher that is impossible to crack, no matter how powerful your computer is or how much time you have to crack it. We won't have to write a new program to use it either. Our Vigenere program can implement this cipher without any changes. The catch is that it is so inconvenient to use on a regular basis that it is often only used for the most top secret messages.

The one-time pad cipher is an unbreakable cipher. It is a Vigenere cipher where the key (also called a pad) is as long as the message that is encrypted, and the key is never used again for any other message. By following these two rules, your encrypted message will be invulnerable to any cryptanalyst's attack.

To see why the one-time pad (OTP) cipher is unbreakable, let's think about why the regular Vigenere cipher is vulnerable to breaking. Or Vigenere cipher breaking program works by doing frequency analysis. But if the key is the same length as the message, then every possible ciphertext letter is equally probably from the same plaintext letter. Here is an example.

Say that we want to encrypt the message, "If you want to survive out here, you've got to know where your towel is." If we remove the spaces and punctuation, this message has 55 letters. So to encrypt it with a one-time pad, we need a key that is also 55 letters long. Let's use the key kcqyzhepxautiqekxejmoretzhztrwwqdylbttvejmedbsanybpxqik. Encrypting the string looks like this:

Plaintext	ifyouwanttosurviveouthereyouvegottoknowwhereyourtowelis
Key	kcqyzhepxautiqekxejmoretzhztrwwqdylbttvejmedbsanybpxqik
Ciphertext	shomtdecqtilchzssixghyikdfnnmacewrzlghraqqvhzguerplbbqc

Now imagine a cryptanalyst got a hold of the ciphertext. How could they attack the cipher? Brute forcing through the keys would not work, because there are too many even for a computer. The number of keys is 26^(number of letters in the message), so if the message has 55 letters, there would be a total of 666,091,878,431,395,624,153,823,182,526,730,590,376,250,379,528,249,805,353,030,484,209,594,192,101,376 possible keys. If the world had a trillion times as many computers that were each a trillion times more powerful and all working to try each key, it would still take longer than the universe has existed to go through a fraction of these keys.

But it turns out that even if we had a computer that was powerful enough to try all the keys, it still would not break the one-time pad cipher. This is because for any ciphertext, all possible plaintext messages are equally likely.

Consider the ciphertext “Sh omt decq ti lchzssi xgh yikd, fnn'ma cew rz lghr aqqvh zgue rplbb qc.” We know that if the key used was “kcqyzhepxautiqekxejmoretzhztrwwqdylbttvejmedbsanybpxqik” then the plaintext must be “If you want to survive out here, you've got to know where your towel is.” But what if we tried the key “kcvfpcwlnlqjlhakorxtevozshztrwvetjggptfpwdtagzulohyxyip”. As far as the cryptanalyst goes, this is as likely to be the key as any other random 55 letter long key. Then the ciphertext would decrypt to, “If the bird is crazier and dull, you've had it from lunch that dined in.”

This message makes no sense, but it is valid English. The cryptanalyst has no way of telling if this was the original message or not. In fact, any plaintext message that is 55 letters long and matches the space and punctuation of the ciphertext is just as likely to be the original plaintext. There is no way to tell which plaintext was the original one the cryptographer encrypted without knowing what the encryption key.

The one-time pad cipher is mathematically impossible to break. However, the fact that the key must be the same length as the plaintext makes it very inconvenient to use. But if you need absolute secrecy, then no cipher can beat the one-time pad.

Chapter 18 - Frequency Analysis

There are 26 letters in the English alphabet, but some letters are used more often than others. For example, if you look at the letters in this book you will find that the letters “E”, “T”, “A” and “O” occur very frequently in English words. But the letters “J”, “X”, “Q”, and “Z” are rarely found in English text. We can use this fact to help crack encrypted messages. This technique is called frequency analysis.

Think about the transpositional cipher. It contains all the original letters of the original English plaintext except in a different order. But the frequency of each letter in the ciphertext remains the same: “E”, “T”, and “A” should occur much more often than “Q” and “Z”. The Caesar and simple substitution ciphers that have their letters replaced, but you can still count the frequency of the letters. There should be letters that commonly occur in the ciphertext. These letters are good candidates for being the “E”, “T”, or “A” letters.

Chapter 19 - Breaking the Vigenere Cipher

We will try two different methods to break the Vigenere cipher. The first is a brute force attack that uses English recognition. This method will only work if the user chose a very weak key. The second is a more sophisticated method, which was invented by the mathematician Charles Babbage in the 19th century.

The First Method: The Dictionary Attack

The Second Method: The Babbage Attack