5

THE CAESAR CIPHER

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

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

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

TOPICS COVERED IN THIS CHAPTER

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

Source Code for the Caesar Cipher Program

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

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

caesarCipher.py

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

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

Sample Run of the Caesar Cipher Program

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

```
guv6Jv6Jz!J6rp5r7Jzr66ntrM
```

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

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

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

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

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

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

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

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

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

Importing Modules and Setting Up Variables

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

Line 4 contains an `import` statement:

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

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

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

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

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

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

Constants and Variables

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

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

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

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

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

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

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

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

The for Loop Statement

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

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

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

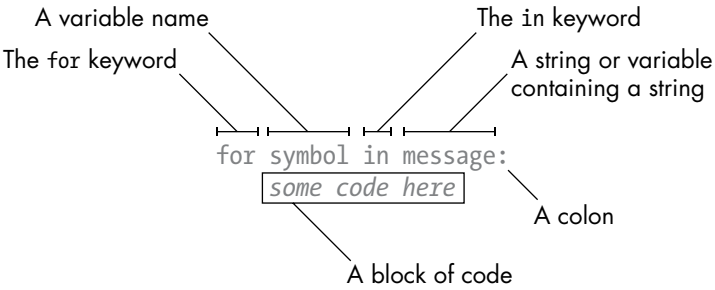

Figure 5-1: The six parts of a for loop statement

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

An Example for Loop

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

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

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

A while Loop Equivalent of a for Loop

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

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

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

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

The if Statement

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

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

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

An Example if Statement

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

checkPw.py

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

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

The else Statement

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

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

checkPw.py

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

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

The elif Statement

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

checkPw.py

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

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

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

The in and not in Operators

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

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

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

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

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

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

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

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

The find() String Method

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

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

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

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

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

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

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

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

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

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

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

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

Encrypting and Decrypting Symbols

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

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

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

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

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

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

caesarCipher.py

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

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

Handling Wraparound

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

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

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

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

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

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

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

Handling Symbols Outside of the Symbol Set

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

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

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

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

Displaying and Copying the Translated String

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

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

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

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

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

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

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

Encrypting Other Symbols

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

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

For example, you could expand line 16 to be:

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

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

Summary

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

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

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

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

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

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

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

PRACTICE QUESTIONS

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

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

   a.

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

(continued)

b.

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

c.

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

d.

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

e.

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