4
THE REVERSE CIPHER

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

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

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

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

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

TOPICS COVERED IN THIS CHAPTER

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

Source Code for the Reverse Cipher Program

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

reverseCipher.py

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

Sample Run of the Reverse Cipher Program

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

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

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

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

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

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

Setting Up Comments and Variables

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

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

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

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

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

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

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

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

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

Finding the Length of a String

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

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

The expression evaluated and stored in the variable is `len(message) - 1`. The first part of this expression, `len(message)`, is a function call to the `len()` function, which accepts a string argument, just like `print()`, and returns an integer value of how many characters are in the string (that is, the *length* of the string). In this case, we pass the message variable to len(), so len(message) returns how many characters are in the string value stored in message.

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

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

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

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

Introducing the while Loop

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

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

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

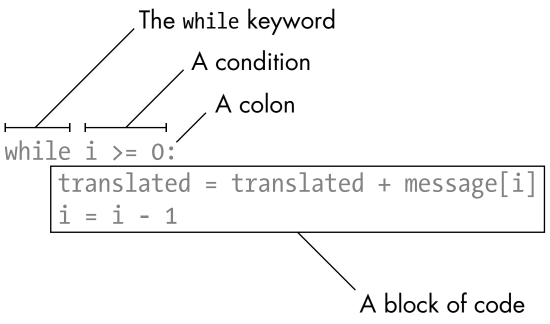

Figure 4-1: The parts of a while loop

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

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

The Boolean Data Type

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

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

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

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

Comparison Operators

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

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

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

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

Table 4-1: Comparison Operators

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

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

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

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

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

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

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

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

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

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

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

String comparisons work similarly:

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

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

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

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

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

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

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

Blocks

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

A block begins when a line is indented by four spaces. Any following line that is also indented by at least four spaces is part of the block. When a line is indented with another four spaces (for a total of eight spaces in front of the line), a new block begins inside the first block. A block ends when there is a line of code with the same indentation as before the block started.

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

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

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

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

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

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

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

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

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

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

The while Loop Statement

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

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

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

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

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

"Growing" a String

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

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

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

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

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

Let's think about the behavior of this loop to understand how many times it runs the code in the block. The variable `i` starts with the value of the last index of `message`, and the `translated` variable starts as a blank string. Then inside the loop, the value of message[i] (which is the last character in the message string, because i will have the value of the last index) is added to the end of the translated string.

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

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

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

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

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

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

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

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

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

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

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

Improving the Program with an input() Prompt

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

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

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

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

Summary

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

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

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

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

PRACTICE QUESTIONS

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

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

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

2. What does this code print?

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

3. How about this code?

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

4. And this?

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