6

HACKING THE CAESAR CIPHER WITH BRUTE-FORCE

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

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

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

TOPICS COVERED IN THIS CHAPTER

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

Source Code for the Caesar Cipher Hacker Program

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

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

caesarHacker.py

```python
# Caesar Cipher Hacker
# https://www.nostarch.com/crackingcodes/ (BSD Licensed)

message = 'guv6Jv6Jz!J6rp5r7Jzr66ntrM'
SYMBOLS = 'ABCDEFGHIJKLMNOPQRSTUVWXYZabcdefghijklmnopqrstuvwxyz1234567890 !?.'

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

    # The rest of the program is almost the same as the Caesar program:

```

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

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

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

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

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

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

Sample Run of the Caesar Cipher Hacker Program

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

```
Key #0: guv6Jv6Jz!J6rp5r7Jzr66ntrM
Key #1: ftu5Iu5Iy I5qo4q6Iyq55msqL
Key #2: est4Ht4Hx0H4pn3p5Hxp44lrpK
Key #3: drs3Gs3Gw9G3om2o4Gwo33kqoJ
Key #4: cqr2Fr2Fv8F2nl1n3Fvn22jpnI
--snip--
Key #11: Vjku?ku?o1?ugetgv?oguucigB
Key #12: Uijt!jt!nz!tfdsfu!nfttbhfA
Key #13: This is my secret message.
Key #14: Sghr0hr0lx0rdbqds0ldrrZfd?
Key #15: Rfgq9gq9kw9qcapcr9kcqqYec!
--snip--
Key #61: lz1 O1 O5CO wuOw!O5w  sywR
Key #62: kyzONzON4BNOvt9v N4vOOrxvQ
Key #63: jxy9My9M3AM9us8uOM3u99qwuP
Key #64: iwx8Lx8L2.L8tr7t9L2t88pvtO
Key #65: hvw7Kw7K1?K7sq6s8K1s77ousN
```

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

Setting Up Variables

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

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

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

Looping with the range() Function

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

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

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

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

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

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

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

```
2
3
4
5
```

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

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

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

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

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

Decrypting the Message

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

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

It's important that we reset `translated` to a blank string at the beginning of this `for` loop; otherwise, the text that was decrypted with the current key will be added to the decrypted text in `translated` from the last iteration in the loop.

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

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

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

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

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

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

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

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

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

Using String Formatting to Display the Key and Decrypted Messages

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

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

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

Enter the following into the interactive shell:

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

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

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

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

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

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

Summary

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

PRACTICE QUESTION

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

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

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

z.GM?.cEQc. 70c.7KcKMKHA9AGFK,
?MFYp2pPJJUpZSIJWpRdpMFY,
ZqH8sl5HtqHTH4s3lyvH5zH5spH4t pHzqHlH3l5K

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