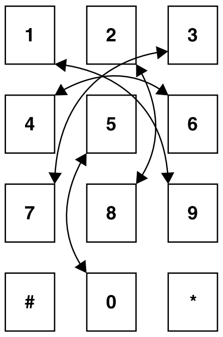
4 Jump the Five: Working with dictionaries

"When I get up, nothing gets me down."

—D. L. Roth



In an episode of the television show *The Wire*, drug dealers assume the police are intercepting their text messages. Whenever a phone number needs to be texted in the course of a criminal conspiracy, the dealers will obfuscate the number. They use an algorithm we'll call "Jump the Five" because each number is changed to its mate on the opposite of a US telephone pad if you jump over the 5. In this exercise, we'll discuss how to encrypt messages using this algorithm, and then we'll see how to use it to decrypt the encrypted messages, you feel me?

If we start with the 1 button and jump across the 5, we get to 9. The 6 jumps the 5 to become 4, and so forth. The numbers 5 and 0 will swap with each other.

In this exercise, we're going to write a Python program called jump.py that will take in some text as a positional argument. Each number in the text will be encoded using this algorithm. All non-number text will pass through unchanged. Here are a couple of examples:

```
$ ./jump.py 867-5309
243-0751
$ ./jump.py 'Call 1-800-329-8044 today!'
Call 9-255-781-2566 today!
```

You will need some way to inspect each character in the input text to identify the numbers—you will learn how to use a `for` loop for this. Then you'll see how a `for` loop can be rewritten as a "list comprehension." You'll need some way to associate a number like 1 with the number 9, and so on for all the numbers—you'll learn about a data structure in Python called a *dictionary* that will allow you to do exactly that.

In this chapter, you will learn to

- Create a dictionary
- Use a `for` loop and a list comprehension to process text, character by character
- Check if items exist in a dictionary
- Retrieve values from a dictionary
- Print a new string with the numbers substituted with their encoded values

Before we start writing, you need to learn about Python's dictionaries.

4.1 Dictionaries

A Python dictionary allows us to relate some *thing* (a "key") to some other *thing* (a "value"). An actual dictionary does this. If we look up a word like "quirky" in a dictionary (www.merriam-webster.com/dictionary/quirky), we can find a definition, as in figure 4.1. We can think of the word itself as the "key" and the definition as the "value."

quirky ⇨ unusual, esp. in an interesting or appealing way

Figure 4.1 You can find the definition of a word by looking it up in a dictionary.

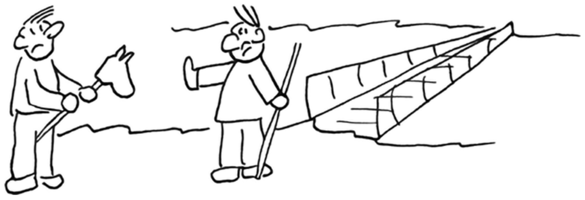
Dictionaries actually provide quite a bit more information about words, such as pronunciation, part of speech, derived words, history, synonyms, alternate spellings, etymology, first known use, and so on. (I really love dictionaries.) Each of those attributes has a value, so we could also think of the dictionary entry for a word as itself being another "dictionary" (see figure 4.2).

definition	⇨	**unusual, esp. in an interesting or appealing way**
pronunciation	⇨	**ˈkwər-kē**
part of speech	⇨	**adjective**

Figure 4.2 The entry for "quirky" can contain much more than a single definition.

Let's see how we can use Python's dictionaries to go beyond word definitions.

4.1.1 Creating a dictionary



In the film *Monty Python and the Holy Grail*, King Arthur and his knights must cross The Bridge of Death. Anyone who wishes to cross must correctly answer three questions from the Keeper. Those who fail are cast into the Gorge of Eternal Peril.

Let us ride to CAMELOT.... No, sorry, let us create and use a dictionary to keep track of the questions and answers as key/value pairs. Once again, I want you to fire up your `python3` or IPython REPL or Jupyter Notebook and type these out for yourself.

Lancelot goes first. We can use the `dict()` function to create an empty dictionary for his answers.

```
>>> answers = dict()
```

Or we can use empty curly brackets (both methods are equivalent):

```
>>> answers = {}
```

The Keeper's first question is, "What is your name?" Lancelot answers, "My name is Sir Lancelot of Camelot." We can add the key "name" to the `answers` dictionary by using square brackets (`[]`—not curlies!) and the literal string `'name'`:

```
>>> answers['name'] = 'Sir Lancelot'
```

If you type `answers` and press Enter in the REPL, Python will show you a structure in curlies (see figure 4.3) to indicate that this is a `dict`:

```
>>> answers
{'name': 'Sir Lancelot'}
```

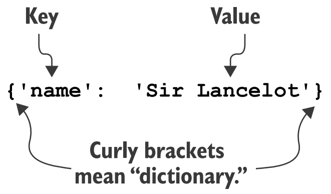


Figure 4.3 A dictionary is printed inside curly braces. The keys are separated from the values by a colon.

You can verify this with the `type()` function:

```
>>> type(answers)
<class 'dict'>
```

Next the Keeper asks, "What is your quest?" to which Lancelot answers "To seek the Holy Grail." Let's add "quest" to `answers`:

```
>>> answers['quest'] = 'To seek the Holy Grail'
```

There's no return value to let us know something happened, so type `answers` to inspect the variable again to ensure the new key/value was added:

```
>>> answers
{'name': 'Sir Lancelot', 'quest': 'To seek the
     Holy Grail'}
```

Finally the Keeper asks, "What is your favorite color?" and Lancelot answers, "Blue."

```
>>> answers['favorite_color'] = 'blue'
>>> answers
{'name': 'Sir Lancelot', 'quest': 'To seek the Holy Grail', 'favorite_color':
     'blue'}
```

NOTE I'm using "favorite_color" (with an underscore) as the key, but I could use "favorite color" (with a space) or "FavoriteColor" or "Favorite color," but each one of those would be a separate and distinct string, or key. I prefer to use the PEP 8 naming conventions for dictionary keys and variable and functions names. PEP 8, the "Style Guide for Python Code" (www.python.org/dev/peps/pep-0008/), suggests using lowercase names with words separated by underscores.

If you knew all the answers beforehand, you could create `answers` using the `dict()` function with the following syntax, where you do *not* have to quote the keys, and the keys are separated from the values with equal signs:

```
>>> answers = dict(name='Sir Lancelot', quest='To seek the Holy Grail',
     favorite_color='blue')
```

Or you could use the following syntax using curlies `{}`, where the keys must be quoted and they are followed by a colon (`:`):

```
>>> answers = {'name': 'Sir Lancelot', 'quest': 'To seek the Holy Grail',
     'favorite_color': 'blue'}
```

It might be helpful to think of the `answers` dictionary as a box holding key/value pairs that describe Lancelot's answers (see figure 4.4), just the way the "quirky" dictionary holds all the information about that word.

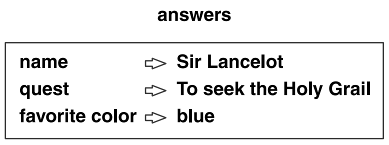


Figure 4.4 Just like the "quirky" dictionary entry, a Python dictionary can contain many key/value pairs.

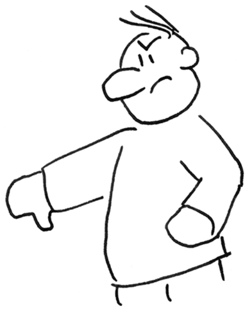
4.1.2 Accessing dictionary values

To retrieve the values, you use the key name inside square brackets (`[]`). For instance, you can get the `name` like so:

```
>>> answers['name']
'Sir Lancelot'
```

Let's request his "age":

```
>>> answers['age']
Traceback (most recent call last):
  File "<stdin>", line 1, in <module>
KeyError: 'age'
```



As you can see, you will cause an exception if you ask for a dictionary key that doesn't exist!

Just as with strings and lists, you can use `x in y` to first see if a key exists in the `dict`:

```
>>> 'quest' in answers
True
>>> 'age' in answers
False
```

The `dict.get()` method is a *safe* way to ask for a value:

```
>>> answers.get('quest')
'To seek the Holy Grail'
```

When the requested key does not exist in the `dict`, it will return the special value `None`:

```
>>> answers.get('age')
```

That doesn't print anything because the REPL won't print a `None`, but we can check the `type()`. Note that the type of `None` is the `NoneType`:

```
>>> type(answers.get('age'))
<class 'NoneType'>
```

There is an optional second argument you can pass to `dict.get()`, which is the value to return *if the key does not exist*:

```
>>> answers.get('age', 'NA')
'NA'
```

That's going to be important for the solution because we will only need to represent the characters 0–9.

4.1.3 Other dictionary methods

If you want to know how "big" a dictionary is, the `len()` (length) function on a `dict` will tell you how many key/value pairs are present:

```
>>> len(answers)
3
```

The `dict.keys()` method will give you just the keys:

```
>>> answers.keys()
dict_keys(['name', 'quest', 'favorite_color'])
```

And `dict.values()` will give you just the values:

```
>>> answers.values()
dict_values(['Sir Lancelot', 'To seek the Holy Grail', 'blue'])
```

Often we want both together, so you might see code like this:

```
>>> for key in answers.keys():
...     print(key, answers[key])
...
name Sir Lancelot
quest To seek the Holy Grail
favorite_color blue
```

An easier way to write this would be to use the `dict.items()` method, which will return the contents of the dictionary as a new `list` containing each key/value pair:

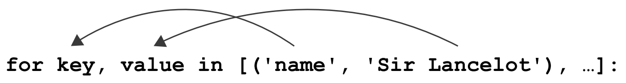
```
>>> answers.items()
dict_items([('name', 'Sir Lancelot'), ('quest', 'To seek the Holy Grail'),
('favorite_color', 'blue')])
```

The preceding `for` loop could also be written using the `dict.items()` method:

```
>>> for key, value in answers.items():
 ...     print(f'{key:15} {value}')
...
name            Sir Lancelot
quest           To seek the Holy Grail
favorite_color  blue
```

Unpack each key/value pair into the variables key and value (see figure 4.5). Note that you don't have to call them key and value. You could use k and v or question and answer.

Print the key in a left-justified field 15 characters wide. The value is printed normally.



```
for key, value in [('name', 'Sir Lancelot'), …]:
```

Figure 4.5 We can unpack the key/value pairs returned by `dict.items()` into variables.

In the REPL you can execute `help(dict)` to see all the methods available to you, like `dict.pop()`, which removes a key/value, or `dict.update()`, which merges one dictionary with another.

TIP Each key in the `dict` is unique.

That means if you set a value for a given key twice,

```
>>> answers = {}
>>> answers['favorite_color'] = 'blue'
>>> answers
{'favorite_color': 'blue'}
```

you will not have two entries but one entry with the *second* value:

```
>>> answers['favorite_color'] = 'red'
>>> answers
{'favorite_color': 'red'}
```

Keys don't have to be strings—you can also use numbers like the `int` and `float` types. Whatever value you use must be immutable. For instance, lists could not be used because they are mutable, as you saw in the previous chapter. You'll learn which types are immutable as we go further.

4.2 Writing jump.py

Now let's get started with writing our program. You'll need to create a program called jump.py in the 04_jump_the_five directory so you can use the test.py that is there. Figure 4.6 shows a diagram of the inputs and outputs. Note that your program will only affect the numbers in the text. Anything that is *not* a number will remain unchanged.

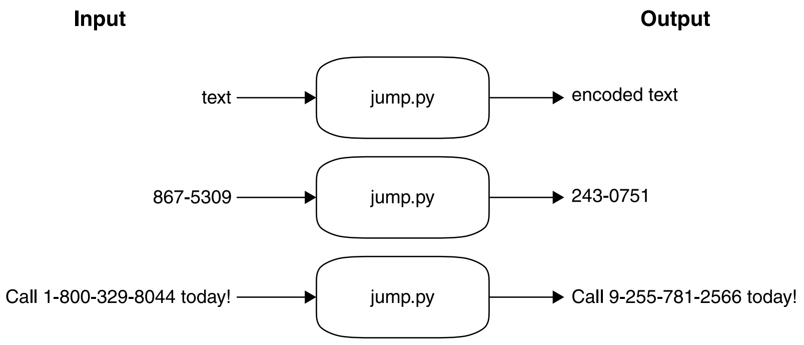


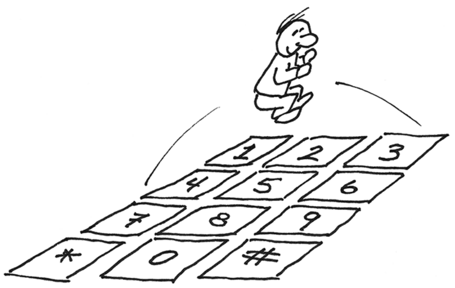
Figure 4.6 A string diagram for the jump.py program. Any number in the input text will be changed to a corresponding number in the output text.

When your program is run with no arguments, `-h`, or `--help`, it should print a usage message:

```
$ ./jump.py -h
usage: jump.py [-h] str

Jump the Five

positional arguments:
  str         Input text

optional arguments:
  -h, --help  show this help message and exit
```

Note that we will be processing *text* representations of the "numbers," so the string `'1'` will be converted to the string `'9'`. We won't be changing the actual integer value `1` to the integer value `9`. Keep that in mind as you figure out a way to represent the substitutions in table 4.1.

Table 4.1 The encoding table for the numeric characters in the text

`1 => 9`	
`2 => 8`	
`3 => 7`	
`4 => 6`	
`5 => 0`	
`6 => 4`	
`7 => 3`	
`8 => 2`	
`9 => 1`	
`0 => 5`	

How would you represent this using a `dict`? Try creating a `dict` called `jumper` in the REPL with the preceding key/value pairs, and then see if the following `assert` statements will execute with exceptions. Remember that `assert` will return nothing if the statement is `True`.

```
>>> assert jumper['1'] == '9'
>>> assert jumper['5'] == '0'
```

Next, you will need a way to visit each character. I suggest you use a `for` loop, like so:

```
>>> for char in 'ABC123':
...     print(char)
...
A
B
C
1
2
3
```

Rather than printing the `char`, print the value of `char` in the `jumper` table, or print the `char` itself. Look at the `dict.get()` method! Also, if you read `help(print)`, you'll see there is an `end` option to replace the newline that gets stuck onto the end with something else.

Here are some other hints:

- The numbers can occur anywhere in the text, so I recommend you process the input character by character with a `for` loop.
- Given any one character, how can you look it up in your table?
- If the character is in your table, how can you get the value (the translation)?
- How can you `print()` the translation or the value without printing a newline? Look at `help(print)` in the REPL to read about the options for `print()`.
- If you read `help(str)` on Python's `str` class, you'll see that there is a `str.replace()` method. Could you use that?

Now spend the time to write the program on your own before you look at the solutions. Use the tests to guide you.

4.3 Solution

Here is one solution that satisfies the tests. I will show some variations after we discuss this first version.

```
#!/usr/bin/env python3
"""Jump the Five"""

import argparse


# --------------------------------------------------
def get_args():
    """Get command-line arguments"""

    parser = argparse.ArgumentParser(
        description='Jump the Five',
        formatter_class=argparse.ArgumentDefaultsHelpFormatter)

    parser.add_argument('text', metavar='str', help='Input text')

    return parser.parse_args()


# --------------------------------------------------
def main():
    """Make a jazz noise here"""

    args = get_args()
    jumper = {'1': '9', '2': '8', '3': '7', '4': '6', '5': '0',
              '6': '4', '7': '3', '8': '2', '9': '1', '0': '5'}
```

Define the get_args() function first, so it's easy to find when I read the program.

Define one positional argument called "text."

Define a main() function where the program starts.

Get the command-line arguments from get_args ().

Create a dictionary for the lookup table.

```
    for char in args.text:
        print(jumper.get(char, char), end='')
    print()


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

Process each character in the input text.

Print either the value of the character from the "jumper" table or the character itself. Change the "end" value to print() so as to avoid adding a newline.

Print a newline after I am done processing the characters.

Call the main() function if the program is in the "main" namespace.

4.4 Discussion

Let's break this program down into the big ideas, like how we define the parameters, define and use a dictionary, process the input text, and print the output.

4.4.1 Defining the parameters

As usual, the `get_args()` function is defined first. The program needs to define one positional argument. Since I'm expecting some "text," I call the argument `'text'` and then assign that to a variable called `text`:

```
parser.add_argument('text', metavar='str', help='Input text')
```

While that seems rather obvious, I think it's very important to name things for *what they are*. That is, please don't leave the name of the argument as `'positional'`—that does not describe what it *is*.

It may seem like overkill to use `argparse` for such a simple program, but it handles the validation of the correct *number* and *type* of the arguments as well as generating help documentation, so it's well worth the effort.

4.4.2 Using a dict for encoding

I suggested you could represent the substitution table as a `dict`, where each number `key` has its substitute as the `value` in the `dict`. For instance, I know that if I jump from 1 over the 5, I should land on 9:

```
>>> jumper = {'1': '9', '2': '8', '3': '7', '4': '6', '5': '0',
...           '6': '4', '7': '3', '8': '2', '9': '1', '0': '5'}
>>> jumper['1']
'9'
```

Since there are only 10 numbers to encode, this is probably the easiest way to write this. Note that the numbers are written with quotes around them, so they are actually of the type `str` and not `int` (integers). I do this because I will be reading characters from a `str`. If I stored them as actual numbers, I would have to coerce the `str` types using the `int()` function:

```
>>> type('4')
<class 'str'>
```

```
>>> type(4)
<class 'int'>
>>> type(int('4'))
<class 'int'>
```

4.4.3 Various ways to process items in a series

As you've seen before, strings and lists in Python are similar in how you can index them. Both strings and lists are essentially sequences of elements—strings are sequences of characters, and lists can be sequences of anything at all.

There are several different ways to process any sequence of items, which here will be characters in a string.

METHOD 1: USING A FOR LOOP TO PRINT() EACH CHARACTER

As I suggested in the introduction, we can process each character of the `text` using a `for` loop. To start, I might first see if each character of the text is in the `jumper` table using the `x in y` construct:

```
>>> text = 'ABC123'
>>> for char in text:
...     print(char, char in jumper)
...
A False
B False
C False
1 True
2 True
3 True
```

NOTE When `print()` is given more than one argument, it will put a space between each bit of text. You can change that with the `sep` argument. Read `help(print)` to learn more.

Now let's try to translate the numbers. I could use an `if` expression, where I print the value from the `jumper` table if `char` is present, and, otherwise, print the `char`:

```
>>> for char in text:
...     print(char, jumper[char] if char in jumper else char)
...
A A
B B
C C
1 9
2 8
3 7
```

It's a bit laborious to check for every character, but it's necessary because, for instance, the letter "A" is not in `jumper`. If I try to retrieve that value, I'll get an exception:

```
>>> jumper['A']
Traceback (most recent call last):
  File "<stdin>", line 1, in <module>
KeyError: 'A'
```

The `dict.get()` method allows me to safely ask for a value if it is present. Asking for "A" will not produce an exception, but it will also not show anything in the REPL because it returns the `None` value:

```
>>> jumper.get('A')
```

It's a bit easier to see if we try to `print()` the values:

```
>>> for char in text:
...     print(char, jumper.get(char))
...
A None
B None
C None
1 9
2 8
3 7
```

I can provide a second, optional argument to `dict.get()`, which is the default value to return when the key does not exist. In this program, I want to print the character itself when it does not exist in `jumper`. For instance, if I had "A," I'd want to print "A":

```
>>> jumper.get('A', 'A')
'A'
```

But if I have "5," I want to print "0":

```
>>> jumper.get('5', '5')
'0'
```

I can use that to process all the characters:

```
>>> for char in text:
...     print(jumper.get(char, char))
...
A
B
C
9
8
7
```

I don't want that newline printing after every character, so I can use `end=''` to tell Python to put the empty string at the `end` instead of a newline.

When I run this in the REPL, the output is going to look funny because I have to press Enter after the `for` loop to run it. Then I'll be left with `ABC987` with no newline, and then the `>>>` prompt:

```
>>> for char in text:
...     print(jumper.get(char, char), end='')
...
ABC987>>>
```

In your code, you'll have to add another `print()`.

It's useful that you can change what is added at the `end`, and that you can `print()` with no arguments to print a newline. There are several other really cool things `print()` can do, so I encourage you to read `help(print)` and try them out.

METHOD 2: USING A FOR LOOP TO BUILD A NEW STRING

There are several other ways you could solve this. While it was fun to explore all the things we can do with `print()`, that code is a bit ugly. I think it's cleaner to create a `new_text` variable and call `print()` once with that:

```
def main():
    args = get_args()
    jumper = {'1': '9', '2': '8', '3': '7', '4': '6', '5': '0',
              '6': '4', '7': '3', '8': '2', '9': '1', '0': '5'}
    new_text = ''
    for char in args.text:
        new_text += jumper.get(char, char)
    print(new_text)
```

Create an empty new_text variable.

Use the same for loop.

Append either the encoded number or the original char to the new_text.

Print the new_text.

In this version, I start by setting `new_text` equal to the empty string:

```
>>> new_text = ''
```

I use the same `for` loop to process each character in the `text`. Each time through the loop, I use `+=` to append the right side of the equation to the left side. The `+=` adds the value on the right to the variable on the left:

```
>>> new_text += 'a'
>>> assert new_text == 'a'
>>> new_text += 'b'
>>> assert new_text == 'ab'
```

On the right, I'm using the `jumper.get()` method. Each character will be appended to the `new_text`, as shown in figure 4.7.

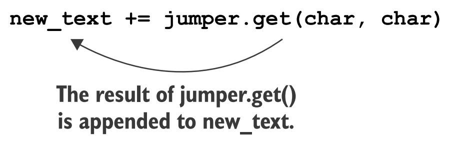

new_text += jumper.get(char, char)
The result of jumper.get() is appended to new_text.

Figure 4.7 The += operator will append the string on the right to the variable on the left.

```
>>> new_text = ''
>>> for char in text:
...     new_text += jumper.get(char, char)
...
```

Now I can call `print()` once with the new value:

```
>>> print(new_text)
ABC987
```

Method 3: Using a for loop to build a new list

This method is the same as the preceding one, but rather than `new_text` being a `str`, it's a `list`:

```
def main():
    args = get_args()
    jumper = {'1': '9', '2': '8', '3': '7', '4': '6', '5': '0',
              '6': '4', '7': '3', '8': '2', '9': '1', '0': '5'}
    new_text = []
    for char in args.text:
        new_text.append(jumper.get(char, char))
    print(''.join(new_text))
```

Initialize new_text as an empty list.

Iterate through each character of the text.

Append the results of the jumper.get () call to the new_text variable.

Join the new_text on the empty string to create a new string to print.

As we go through the book, I'll keep reminding you how Python treats strings and lists similarly. Here I'm using `new_text` exactly the same as I did before, starting with an empty structure and then making it longer for each character. I could actually use the exact same += syntax instead of the `list.append()` method:

```
for char in args.text:
    new_text += jumper.get(char, char)
```

After the `for` loop is done, I have all the new characters that need to be put back together using `str.join()` into a new string that I can `print()`.

Method 4: Turning a for loop into a list comprehension

A shorter solution uses a *list comprehension*, which is basically a one-line `for` loop inside square brackets (`[]`) that results in a new `list` (see figure 4.8).

```
def main():
    args = get_args()
    jumper = {'1': '9', '2': '8', '3': '7', '4': '6', '5': '0',
              '6': '4', '7': '3', '8': '2', '9': '1', '0': '5'}
    print(''.join([jumper.get(char, char) for char in args.text]))
```

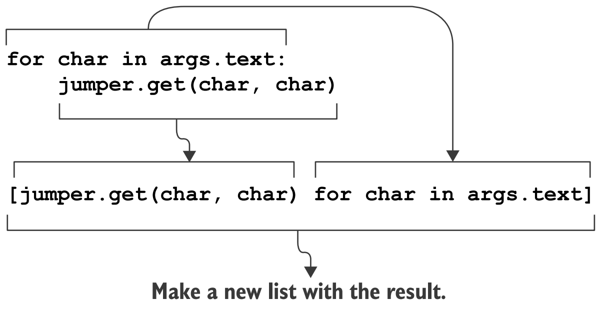

for char in args.text:
    jumper.get(char, char)
[jumper.get(char, char) for char in args.text]
Make a new list with the result.

Figure 4.8 A list comprehension will generate a new `list` with the results of iterating with a `for` statement.

A list comprehension is read backwards from a for loop, but it's all there. It's one line of code instead four!

```
>>> text = '867-5309'
>>> [jumper.get(char, char) for char in text]
['2', '4', '3', '-', '0', '7', '5', '1']
```

You can use str.join() on the empty string to turn that list into a new string you can print():

```
>>> print(''.join([jumper.get(char, char) for char in text]))
243-0751
```

The purpose of a list comprehension is to create a new list, which is what we were trying to do with the for loop code before. A list comprehension makes much more sense and uses far fewer lines of code.

METHOD 5: USING THE STR.TRANSLATE() FUNCTION

This last approach uses a really powerful method from the str class to change all the characters in one step:

```
def main():
    args = get_args()
    jumper = {'1': '9', '2': '8', '3': '7', '4': '6', '5': '0',
              '6': '4', '7': '3', '8': '2', '9': '1', '0': '5'}
    print(args.text.translate(str.maketrans(jumper)))
```

The argument to str.translate() is a translation table that describes how each character should be translated. That's exactly what jumper does.

```
>>> text = 'Jenny = 867-5309'
>>> text.translate(str.maketrans(jumper))
'Jenny = 243-0751'
```

I'll explain this in much greater detail in chapter 8.

4.4.4 (Not) using str.replace()

I asked earlier whether you could use str.replace() to change all the numbers. It turns out you cannot, because you'll end up changing some of the values twice so that they end up at their original values.

Watch how we start off with this string:

```
>>> text = '1234567890'
```

When you change "1" to "9," now you have two 9's:

```
>>> text = text.replace('1', '9')
>>> text
'9234567890'
```

This means that when you try to change all the 9's to 1's, you end up with two 1's. The `1` in the first position is changed to `9` and then back to `1` again:

```
>>> text = text.replace('9', '1')
>>> text
'1234567810'
```

So if you go through each number in "1234567890" and try to change them using `str.replace()`, you'll end up with the value "1234543215":

```
>>> text = '1234567890'
>>> for n in jumper.keys():
...     text = text.replace(n, jumper[n])
...
>>> text
'1234543215'
```

But the correctly encoded string is "9876043215." The `str.translate()` function exists to change all the values in one move, all while leaving the unchanging characters alone.

4.5 Going further

- Try creating a similar program that encodes the numbers with strings (for example, "5" becomes "five," "7" becomes "seven"). Be sure to write the necessary tests in `test.py` to check your work!
- What happens if you feed the output of the program back into itself? For example, if you run `./jump.py 12345`, you should get `98760`. If you run `./jump.py 98760`, do you recover the original numbers? This is called *round-tripping*, and it's a common operation with algorithms that encode and decode text.

Summary

- You can create a new dictionary using the `dict()` function or with empty curly brackets (`{}`).
- Dictionary values are retrieved using their keys inside square brackets or by using the `dict.get()` method.
- For a `dict` called `x`, you can use `'key' in x` to determine if a key exists.
- You can use a `for` loop to iterate through the characters of a `str` just like you can iterate through the elements of a `list`. You can think of strings as lists of characters.
- The `print()` function takes optional keyword arguments like `end=''`, which you can use to print a value to the screen without a newline.