16 The scrambler: Randomly reordering the middles of words

Yuor brian is an azinamg cmiobiaontn of hdarware and sftraowe. Yoru'e rdineag tihs rhgit now eevn thgouh the wrdos are a mses, but yuor biran can mkae snese of it bceause the frsit and lsat ltrtees of ecah wrod hvae saeytd the smae. Yuor biran de'onst atlaulcy raed ecah lteetr of ecah wrod but rades wlohe wdors. The scamrbeld wrdos difteienly solw you dwon, but y'roue not rlleay eevn tyinrg to ulsrmbance the lrttees, are you? It jsut hnaepps!

In this chapter, you will write a program called scrambler.py that will scramble each word of the text given as an argument. The scrambling should only work on words with four characters or more, and it should only scramble the letters in the middle of the word, leaving the first and last characters unchanged. The program should take an `-s` or `--seed` option (an `int` with default `None`) to pass to `random.seed()`.

It should handle text on the command line:

```
$ ./scrambler.py --seed 1 "foobar bazquux"
faobor buuzaqx
```

Or text from a file:

```
$ cat ../inputs/spiders.txt
Don't worry, spiders,
I keep house
casually.
$ ./scrambler.py ../inputs/spiders.txt
```

```
D'not wrory, sdireps,
I keep hsuoe
csalluay.
```

Figure 16.1 shows a string diagram to help you think about it.

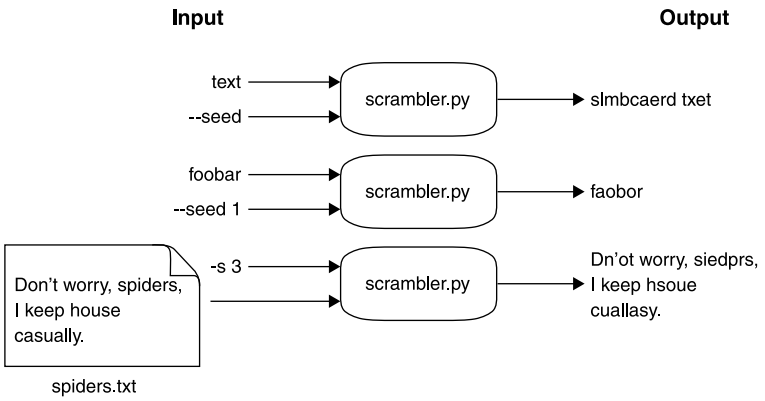


Figure 16.1 Our program will take input text from the command line or a file and will scramble the letters in words with four or more characters.

In this chapter you will

- Use a regular expression to split text into words
- Use the `random.shuffle()` function to shuffle a `list`
- Create scrambled versions of words by shuffling the middle letters while leaving the first and last letters unchanged

16.1 Writing scrambler.py

I recommend you start by using `new.py scrambler.py` to create the program in the 16_scrambler directory. Alternatively, you can copy template/template.py to 16_scrambler/scrambler.py. You can refer to previous exercises, like the one in chapter 5, to remember how to handle a positional argument that might be text or might be a text file to read.

When run with no arguments or the flags `-h` or `--help`, scrambler.py should present a usage statement:

```
$ ./scrambler.py -h
usage: scrambler.py [-h] [-s seed] text

Scramble the letters of words

positional arguments:
  text                  Input text or file
```

```
optional arguments:
  -h, --help            show this help message and exit
  -s seed, --seed seed  Random seed (default: None)
```

Once your program's usage statement matches this, change your `main()` definition as follows:

```
def main():
    args = get_args()
    print(args.text)
```

Then verify that your program can echo text from the command line:

```
$ ./scrambler.py hello
hello
```

Or from an input file:

```
$ ./scrambler.py ../inputs/spiders.txt
Don't worry, spiders,
I keep house
casually.
```

16.1.1 Breaking the text into lines and words

As in chapter 15, we want to preserve the line breaks of the input text by using `str.splitlines()`:

```
for line in args.text.splitlines():
    print(line)
```

If we are reading the spiders.txt haiku, this is the first line:

```
>>> line = "Don't worry, spiders,"
```

We need to break the `line` into words. In chapter 6 we used `str.split()`, but that approach leaves punctuation stuck to our words—both `worry` and `spiders` have commas:

```
>>> line.split()
["Don't", 'worry,', 'spiders,']
```

In chapter 15 we used the `re.split()` function with the regular expression `(\W+)` to split text on one or more non-word characters. Let's try that:

```
>>> re.split('(\W+)', line)
['Don', "'", 't', ' ', 'worry', ', ', 'spiders', ',', '']
```

That won't work because it splits `Don't` into three parts: `Don`, `'`, and `t`.

Perhaps we could use `\b` to break on *word boundaries*. Note that we'd have to put an `r''` in front of the first quote, `r'\b'`, to denote that it is a "raw" string.

This still won't work because \b thinks the apostrophe is a word boundary and so splits the contracted word:

```
>>> re.split(r'\b', "Don't worry, spiders,")
['', 'Don', "'", 't', ' ', 'worry', ', ', 'spiders', ',']
```

While searching the internet for a regex to split this text properly, I found the following pattern on a Java discussion board. It perfectly separates *words* from *non-words*:[1]

```
>>> re.split("([a-zA-Z](?:[a-zA-Z']*[a-zA-Z])?)", "Don't worry, spiders,")
['', "Don't", ' ', 'worry', ', ', 'spiders', ',']
```

The beautiful thing about regular expressions is that they are their own language—one that is used inside many other languages from Perl to Haskell. Let's dig into this pattern, shown in figure 16.2.

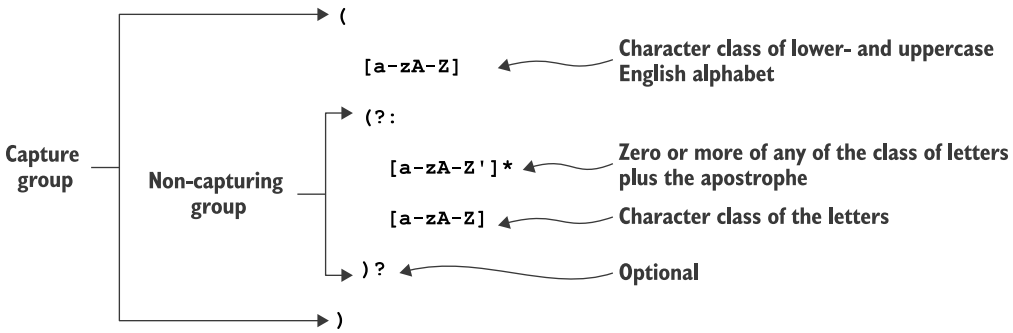


Figure 16.2 A regular expression that will find words that include an apostrophe

16.1.2 Capturing, non-capturing, and optional groups

In figure 16.2 you can see that groups can contain other groups. For instance, here is a regex that can capture the entire string "foobarbaz" as well as the substring "bar":

```
>>> match = re.match('(foo(bar)baz)', 'foobarbaz')
```

Capture groups are numbered by the position of their left parenthesis. Since the first left parenthesis starts the capture starting at "f" and going to "z," that is group 1:

```
>>> match.group(1)
'foobarbaz'
```

[1] I would like to stress that a significant part of my job is spent looking for answers both in the books I own but also on the internet!

The second left parenthesis starts just before the "b" and goes to the "r":

```
>>> match.group(2)
'bar'
```

We can also make a group *non-capturing* by using the starting sequence `(?:`. If we use this sequence on the second group, we no longer capture the substring "bar":

```
>>> match = re.match('(foo(?:bar)baz)', 'foobarbaz')
>>> match.groups()
('foobarbaz',)
```

Non-capturing groups are commonly used when you are grouping primarily for the purpose of making it optional by placing a `?` after the closing parenthesis. For instance, we can make the "bar" optional and then match both "foobarbaz,"

```
>>> re.match('(foo(?:bar)?baz)', 'foobarbaz')
<re.Match object; span=(0, 9), match='foobarbaz'>
```

as well as "foobaz":

```
>>> re.match('(foo(?:bar)?baz)', 'foobaz')
<re.Match object; span=(0, 6), match='foobaz'>
```

16.1.3 Compiling a regex

I mentioned the `re.compile()` function in chapter 15 as a way to incur the cost of compiling a regular expression just once. Whenever you use something like `re.search()` or `re.split()`, the regex engine must parse the `str` value you provide for the regex into something it understands and can use. This parsing step must happen *each time* you call the function. When you compile the regex and assign it to a variable, the parsing step is done before you call the function, which improves performance.

I especially like to use `re.compile()` to assign a regex to a meaningful variable name and/or reuse the regex in multiple places in my code. Because this regex is quite long and complicated, I think it makes the code more readable to assign it to a variable called `splitter`, which will help me to remember how it will be used:

```
>>> splitter = re.compile("([a-zA-Z](?:[a-zA-Z']*[a-zA-Z])?)")
>>> splitter.split("Don't worry, spiders,")
['', "Don't", ' ', 'worry', ', ', 'spiders', ',']
```

16.1.4 Scrambling a word

Now that we have a way to process the *lines* and then *words* of the text, let's think about how we'll scramble the words by starting with just *one word*. You and I will need to use the same algorithm for scrambling the words in order to pass the tests, so here are the rules:

- If the word is three characters or shorter, return the word unchanged.
- Use a string slice to copy the characters, not including the first and last.
- Use the `random.shuffle()` method to mix up the letters in the middle.
- Return the new "word" by combining the first, middle, and last parts.

I recommend you create a function called `scramble()` that will do all this, and also create a test for it. Feel free to add this to your program:

```
def scramble(word):
    """Scramble a word"""
    pass


def test_scramble():
    """Test scramble"""
    state = random.getstate()
    random.seed(1)
    assert scramble("a") == "a"
    assert scramble("ab") == "ab"
    assert scramble("abc") == "abc"
    assert scramble("abcd") == "acbd"
    assert scramble("abcde") == "acbde"
    assert scramble("abcdef") == "aecbdf"
    assert scramble("abcde'f") == "abcd'ef"
    random.setstate(state)
```

The pass is a no-op (no operation), so this function literally does nothing. This is just a placeholder so that we can write a test and verify that the function fails.

The change we'll make by setting the random.seed() in the next line will be a global change. We'll want to restore the state after testing, so here we use random.getstate() to get the current state of the random module.

Set random.seed() to a known value for testing.

Words with three characters or fewer should be returned unchanged.

This word looks unchanged, but that's just because with the seed of 1 the shuffling didn't end up changing the middle characters.

Now it's more evident that the word is being scrambled.

Restore the state to the previous value.

Inside the `scramble()` function, we will have a word like "worry." We can use list slices to extract part of a string. Since Python starts numbering at `0`, we use `1` to indicate the *second* character:

```
>>> word = 'worry'
>>> word[1]
'o'
```

The last index of any string is `-1`:

```
>>> word[-1]
'y'
```

To get a slice, we use the `list[start:stop]` syntax. Since the `stop` position is not included, we can get the `middle` like so:

```
>>> middle = word[1:-1]
>>> middle
'orr'
```

We can `import random` to get access to the `random.shuffle()` function. As with the `list.sort()` and `list.reverse()` methods, the argument will be shuffled *in place*, and the function will return `None`. That is, you might be tempted to write code like this:

```
>>> import random
>>> x = [1, 2, 3]
>>> shuffled = random.shuffle(x)
```

What is the value of `shuffled`? Is it something like `[3, 1, 2]`, or is it `None`?

```
>>> type(shuffled)
<class 'NoneType'>
```

The `shuffled` value now holds `None`, while the `x` list has been shuffled *in place* (see figure 16.3):

```
>>> x
[2, 3, 1]
```

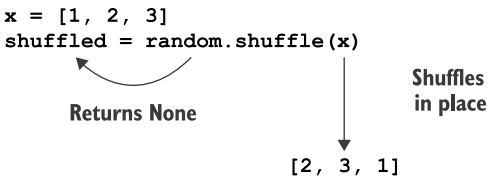


Figure 16.3 The return from `random.shuffle()` was `None`, so `shuffled` was assigned `None`.

If you've been following along, it turns out that we cannot shuffle the `middle` like this:

```
>>> random.shuffle(middle)
Traceback (most recent call last):
  File "<stdin>", line 1, in <module>
  File "/Users/kyclark/anaconda3/lib/python3.7/random.py", line 278, in shuffle
    x[i], x[j] = x[j], x[i]
TypeError: 'str' object does not support item assignment
```

The `middle` variable is a `str`:

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

The `random.shuffle()` function is trying to directly modify a `str` value in place, but `str` values in Python are *immutable*. One workaround is to make `middle` into a new `list` of the characters from `word`:

```
>>> middle = list(word[1:-1])
>>> middle
['o', 'r', 'r']
```

That is something we can shuffle:

```
>>> random.shuffle(middle)
>>> middle
['r', 'o', 'r']
```

Then it's a matter of creating a new string with the original first letter, the shuffled middle, and the last letter. I'll leave that for you to work out.

Use `pytest scrambler.py` to have Pytest execute the `test_scramble()` function to see if it works correctly. Run this command *after every change to your program*. Ensure that your program always compiles and runs properly. Only make one change at a time, and then save your program and run the tests.

16.1.5 Scrambling all the words

As in several previous exercises, we're now down to applying the `scramble()` function to all the words. Can you see a familiar pattern?

```
splitter = re.compile("([a-zA-Z](?:[a-zA-Z']*[a-zA-Z])?)")
for line in args.text.splitlines():
    for word in splitter.split(line):
        # what goes here?
```

We've talked about how to apply a function to each element in a sequence. You might try a `for` loop, a list comprehension, or maybe a `map()`. Think about how you can split the text into words, feed them to the `scramble()` function, and then join them back together to reconstruct the text.

Note that this approach will pass both the words and the non-words (the bits in between each word) to the `scramble()` function. You don't want to modify the non-words, so you'll need a way to check that the argument looks like a word. Maybe a regular expression?

That should be enough to go on. Write your solution and use the included tests to check your program.

16.2 Solution

To me, the program comes down to properly splitting the words and then figuring out the `scramble()` function. Then it's a matter of applying the function and reconstructing the text.

```
#!/usr/bin/env python3
"""Scramble the letters of words"""

import argparse
import os
import re
import random


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

    parser = argparse.ArgumentParser(
        description='Scramble the letters of words',
        formatter_class=argparse.ArgumentDefaultsHelpFormatter)

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

    parser.add_argument('-s',
                        '--seed',
                        help='Random seed',
                        metavar='seed',
                        type=int,
                        default=None)

    args = parser.parse_args()

    if os.path.isfile(args.text):
        args.text = open(args.text).read().rstrip()

    return args


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

    args = get_args()
    random.seed(args.seed)
    splitter = re.compile("([a-zA-Z](?:[a-zA-Z']*[a-zA-Z])?)")

    for line in args.text.splitlines():
        print(''.join(map(scramble, splitter.split(line))))
```

The text argument may be plain text on the command line or the name of a file to read.

The seed option is an int that defaults to None.

Get the arguments so we can check the text value.

If args.text names an existing file, replace the value of args.text with the result of opening and reading the file's contents.

Return the arguments to the caller.

Get the command-line arguments.

Use args.seed to set the random.seed() value. If args.seed is the default None, this is the same as not setting the seed.

Save the compiled regex into a variable.

Use str.splitlines() to preserve the line breaks in the input text.

Use the splitter to break the line into a new list that map() will feed into the scramble() function. Join the resulting list on the empty string to create a new str to print.

```
# --------------------------------------------------
def scramble(word):
    """For words over 3 characters, shuffle the letters in the middle"""

    if len(word) > 3 and re.match(r'\w+', word):
        middle = list(word[1:-1])
        random.shuffle(middle)
        word = word[0] + ''.join(middle) + word[-1]

    return word

# --------------------------------------------------
def test_scramble():
    """Test scramble"""

    random.seed(1)
    assert scramble("a") == "a"
    assert scramble("ab") == "ab"
    assert scramble("abc") == "abc"
    assert scramble("abcd") == "acbd"
    assert scramble("abcde") == "acbde"
    assert scramble("abcdef") == "aecbdf"
    assert scramble("abcde'f") == "abcd'ef"
    random.seed(None)


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

Define a function to scramble() a single word.

Shuffle the middle letters.

Only scramble words with four or more characters if they contain word characters.

Copy the second through the second-to-last characters of the word into a new list called middle.

Set the word equal to the first character, plus the middle, plus the last character.

Return the word, which may have been altered if it met the criteria.

The test for the scramble() function

16.3 Discussion

There is nothing new in `get_args()`, so I trust you'll understand that code. Refer to chapter 5 if you want to revisit how to handle the `args.text` coming from the command line or from a file.

16.3.1 Processing the text

As mentioned earlier in the chapter, I often assign a *compiled* regex to a variable. Here I did it with the `splitter`:

```
splitter = re.compile("([a-zA-Z](?:[a-zA-Z']*[a-zA-Z])?)")
```

The other reason I like to use `re.compile()` is because I feel it can make my code more readable. Without it, I would have to write this:

```
for line in args.text.splitlines():
    print(''.join(map(scramble, re.split("([a-zA-Z](?:[a-zA-Z']*[a-zA-
     Z])?)", line))))
```

That ends up creating a line of code that is 86 characters wide, and the PEP 8 style guide (www.python.org/dev/peps/pep-0008/) recommends we "limit all lines to a maximum of 79 characters." I find the following version much easier to read:

```
splitter = re.compile("([a-zA-Z](?:[a-zA-Z']*[a-zA-Z])?)")
for line in args.text.splitlines():
    print(''.join(map(scramble, splitter.split(line))))
```

You may still find that code somewhat confusing. Figure 16.4 shows the flow of the data:

1. First Python will split the string `"Don't worry, spiders,"`.
2. The splitter creates a new list composed of words (that matched our regex) and non-words (the bits in between).
3. The `map()` function will apply the `scramble()` function to each element of the list.
4. The result of `map()` is a new list with the results of each application of the `scramble()` function.
5. The result of `str.join()` is a new string, which is the argument to `print()`.

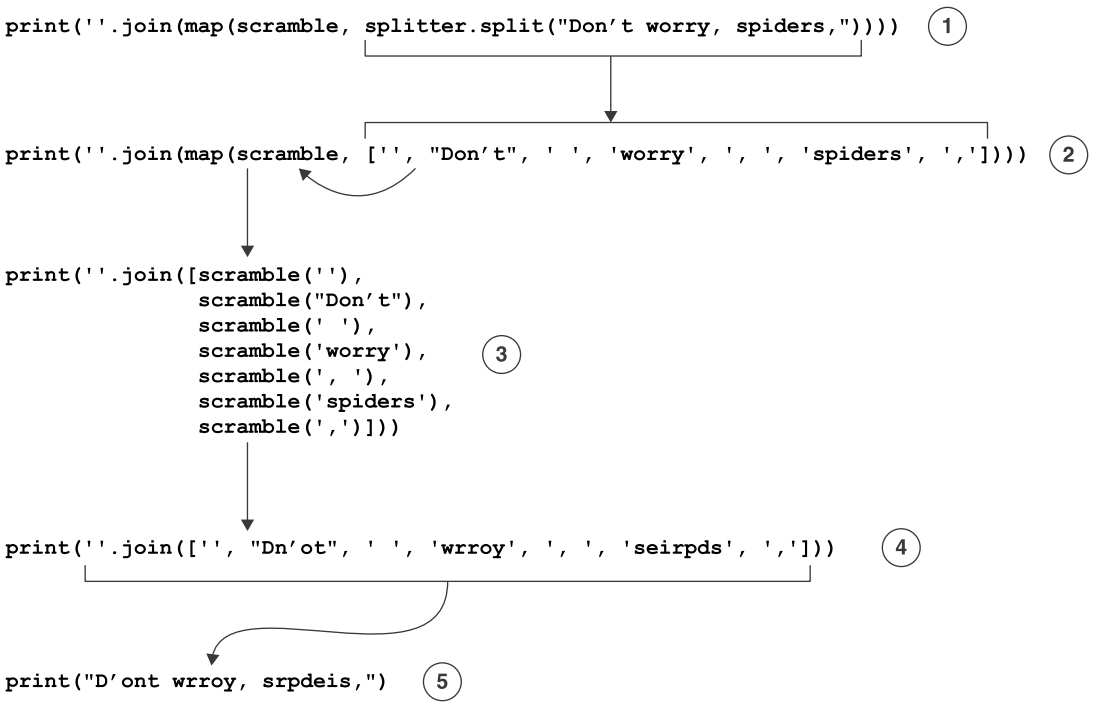
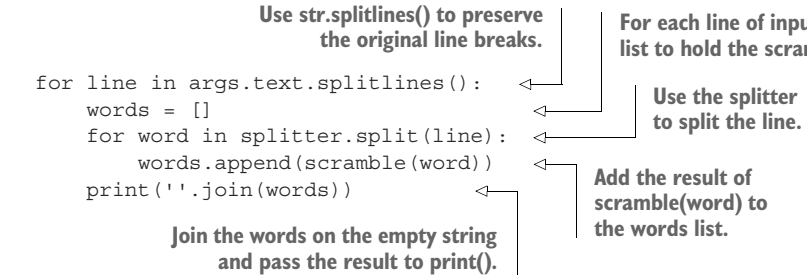


Figure 16.4 A visualization of how data moves through the `map()` function

A longer way to write this with a `for` loop might look like this:




```
for line in args.text.splitlines():
    words = []
    for word in splitter.split(line):
        words.append(scramble(word))
    print(''.join(words))
```

Because the goal is to create a new `list`, this is better written as a list comprehension:

```
for line in args.text.splitlines():
    words = [scramble(word) for word in splitter.split(line)]
    print(''.join(words))
```

Or you could go in quite the opposite direction and replace all the `for` loops with `map()`:

```
print('\n'.join(
    map(lambda line: ''.join(map(scramble, splitter.split(line))),
        args.text.splitlines())))
```

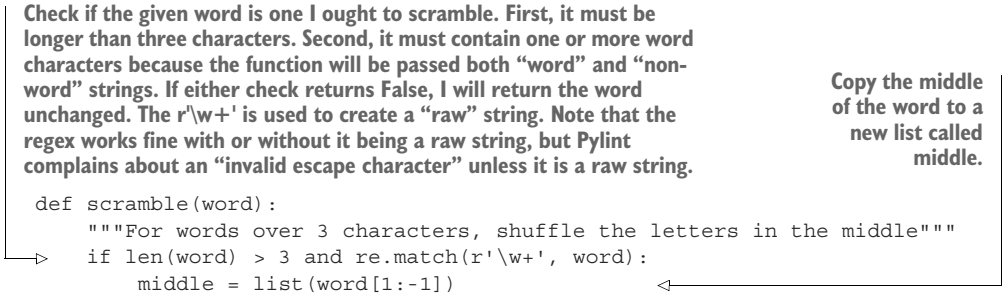
This last solution reminds me of a programmer I used to work with who would jokingly say, "If it was hard to write, it should be hard to read!" It becomes somewhat clearer if you rearrange the code. Note that Pylint will complain about assigning a `lambda`, but I really don't agree with that criticism:

```
scrambler = lambda line: ''.join(map(scramble, splitter.split(line)))
print('\n'.join(map(scrambler, args.text.splitlines())))
```

Writing code that is correct, tested, and understandable is as much an art as it is a craft. Choose the version that you (and your teammates!) believe is the most readable.

16.3.2 Scrambling a word

Let's take a closer look at my `scramble()` function. I wrote it in a way that would make it easy to incorporate into `map()`:




```
def scramble(word):
    """For words over 3 characters, shuffle the letters in the middle"""
    if len(word) > 3 and re.match(r'\w+', word):
        middle = list(word[1:-1])
```

```
    random.shuffle(middle)
    word = word[0] + ''.join(middle) + word[-1]

    return word
```

Shuffle the middle in place. Remember that this function returns None.

Reconstruct the word by joining together the first character, the shuffled middle, and the last character.

Return the word, which may or may not have been shuffled.

16.4 Going further

- Write a version of the program where the `scramble()` function sorts the middle letters into alphabetical order rather than shuffling them.
- Write a version that reverses each word rather than scrambles them.
- Write a program to *unscramble* the text. For this, you need to have a dictionary of English words, which I have provided as inputs/words.txt.zip. You will need to split the scrambled text into words and non-words, and then compare each "word" to the words in your dictionary. I recommend you start by comparing the words as anagrams (that is, they have the same composition/frequency of letters) and then using the first and last letters to positively identify the unscrambled word.

Summary

- The regex we used to split the text into words was quite complex, but it also gave us exactly what we needed. Writing the program without this piece would have been significantly more difficult. Regexes, while complex and deep, are wildly powerful black magic that can make your programs incredibly flexible and useful.
- The `random.shuffle()` function accepts a `list`, which is mutated in place.
- List comprehensions and `map()` can often lead to more compact code, but going too far can reduce readability. Choose wisely.