Chapter 17: 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 exercise, we will write a program called `scrambler.py` that will take a single position positional argument that is text or a file and then scramble each "word" of the text. The scrambling should only work on words with 4 characters or more and should scramble the letters in the middle of the word, leaving the first and last characters unchanged. The program should take a `-s` or `--seed` option (an `int` with default `None`) to pass to `random.seed`.

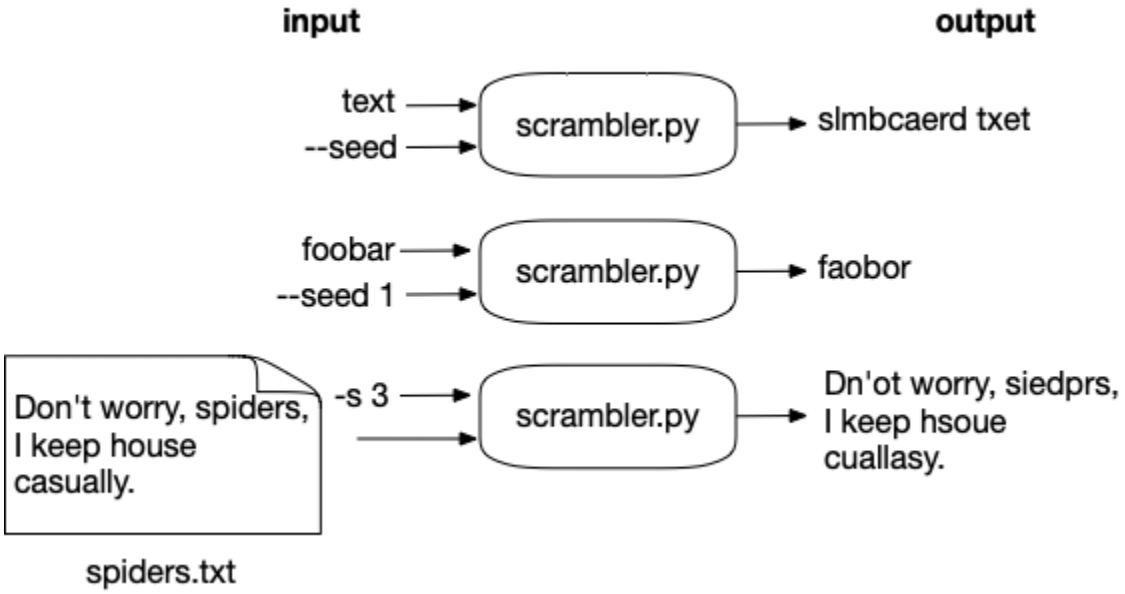
It should handle text on the command line:

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

Or 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.
```

Here's a string diagram to help you think about it:



In this exercise, 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 unchanged.

Writing `scrambler.py`

I recommend you start by using `new.py scrambler.py` to create the program in the `scrambler` directory. You can also copy `template/template.py` to `scrambler/scramber.py`. You can refer to previous exercises like "Howler" to remember how to handle our positional argument that might be text or might a file of text to read. When run with no arguments or the flags `-h` or `--help`, it should present usage:

```
$ ./scrambler.py -h
usage: scrambler.py [-h] [-s int] str

Scramble the letters of words

positional arguments:
  str                 Input text or file

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

Once your program's usage statement matches this, then change your `main` definition to this:

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

And 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.
```

Breaking the text into lines and words

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

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

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

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

We need to break the `line` into "words." In the "Words Count" exercise, it was enough to use `str.split`. This leaves punctation stuck to our words — both `worry,` and `spiders,` have commas:

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

In the "Friar," 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`. When I was working through this, my next idea was to use `\b` to break on *word boundaries*. Note I have to put an `r` in front of the first quote `r'\b'` to denote that it is a "raw" string.

Raw strings

The `r` before the opening quote in the expression `r'\b'` creates a "raw" string which prevents Python from trying to interpret escaped sequences in the string. For instance, we usually type `\n` to represent a newline and `\t` for a tab character, but `\b` isn't supposed to be turned in any other character. We want Python to pass the literal string `\b` (that is, a backslash and then the character `b`) to the regex engine where it can be interpreted as "word boundary." It's generally safer to create regexes as raw strings because of the number of backslash-escaped sequences that are used like `\w` for "word character" or `\s` for "whitespace."

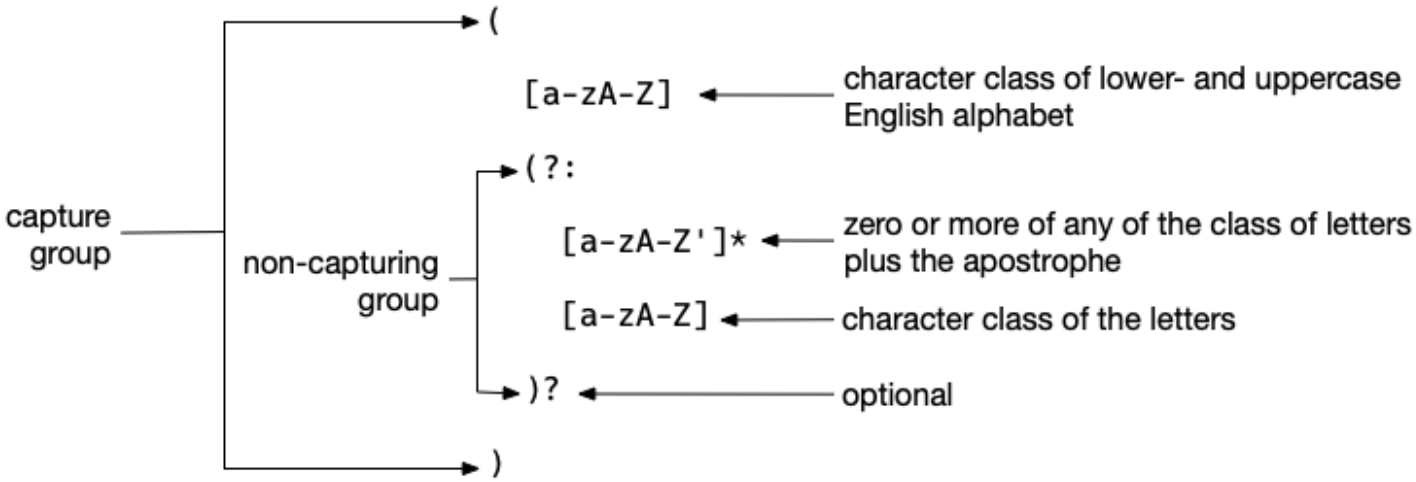
This still won't work because `\b` thinks the apostrophe is a word boundary and so splits the contracted word:

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

While searching the Internet for a regex to split the text properly, I found this pattern on a Java discussion board which works just perfectly to separate *words* from *non-words*:

```
>>> 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:



Capturing, non-capturing, and optional groups

Above you see that groups can contain other groups. For instance, I can capture the entire string "foobarbaz" as well as the contained string "bar" like so:

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

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

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

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

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

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

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

This is often used when you are using a grouping primarily for the purpose of making it optional by placing a `?` after the closing paren. For instance, I 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'>
```

Compiling a regex

You will often see the `re.compile` is used to store a regular expression into a variable that has some meaningful name. I often do this, especially if I use the same regex at different points in my code:

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

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, so we can *compile* the regex just once and reuse it over and over. When you compile the regex and assign it to a variable, the parsing step is already done when you call a method like `search` or `split` which improves performance.

Scrambling a word

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

- If the word is 3 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 a test for it. Feel free to add this to your program:

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

    pass


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)
```

Inside your `scramble` function, you 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'
```

If we want 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 `shuffle` method. Like the `list.sort` and `list.reverse` functions, the argument will be shuffled **in-place**, and the function will return `None`. That is, you might be tempted to write code like this:

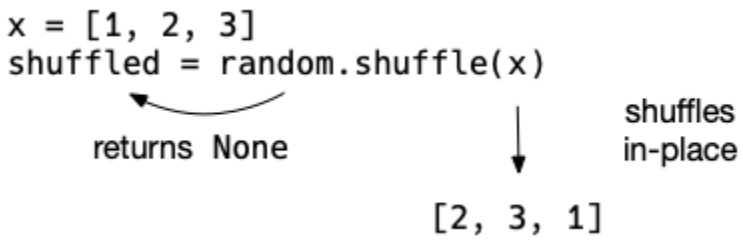
```
>>> import random
>>> x = [1, 2, 3]
>>> shuffled = random.shuffle(x)
```

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

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

The `x` list has been shuffled:

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



And 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']
```

And 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*, then save your program and run the tests!

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?

```
for line in args.text.splitlines():
    for word in re.split("([a-zA-Z](?:[a-zA-Z']*[a-zA-Z])?)", 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 and feed them to the `scramble` function, then join them back together to reconstruct the structure of the original text.

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

Solution

```
#!/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='str', help='Input text or file') 1

    parser.add_argument('-s',           2
                        '--seed',
                        help='Random seed',
                        metavar='int',
                        type=int,
                        default=None)

    args = parser.parse_args()          3

    if os.path.isfile(args.text):       4
        args.text = open(args.text).read()

    return args                         5


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

    args = get_args()                   6
    random.seed(args.seed)              7
    splitter = re.compile("([a-zA-Z](?:[a-zA-Z']*[a-zA-Z])?)") 8

    for line in args.text.splitlines(): 9
        print(''.join(map(scramble, splitter.split(line)))) 10


# --------------------------------------------------
def scramble(word):                                 11
    """For words over 3 characters, shuffle the letters in the middle"""

    if len(word) > 3 and re.match(r'\w+', word):    12
        middle = list(word[1:-1])                   13
        random.shuffle(middle)                      14
        word = word[0] + ''.join(middle) + word[-1] 15

    return word      16


# --------------------------------------------------
def test_scramble(): 17
    """Test scramble"""

    random.seed(1)
```

```
64      assert scramble("a") == "a"
65      assert scramble("ab") == "ab"
66      assert scramble("abc") == "abc"
67      assert scramble("abcd") == "acbd"
68      assert scramble("abcde") == "acbde"
69      assert scramble("abcdef") == "aecbdf"
70      assert scramble("abcde'f") == "abcd'ef"
71      random.seed(None)
72
73
74  # --------------------------------------------------
75  if __name__ == '__main__':
76      main()
```

1. The `text` argument may be plain text on the command line or the name of a file to read.
2. The `seed` option is an `int` that defaults to `None`.
3. Get the `args` so we can check the `text` value.
4. If `args.text` names an existing file, replace the value of `args.text` with the result of opening and reading the file's contents.
5. Return the arguments to the caller.
6. Get the command-line arguments.
7. Use the `args.seed` to set the `random.seed` value. If `args.seed` is the default `None`, then this is the same as not setting the seed.
8. Save the compiled regex into a variable.
9. Use `str.splitlines` to preserve the line breaks in `args.text`.
10. Use the `splitter` to break the `line` into a new a list that `map` will feed into the `scramble` function. Join the resulting `list` on the empty string to create a new `str` to print.
11. Define a function to `scramble` a single `word`.
12. Only scramble words with 4 or more characters if they contain word characters.
13. Copy the second to second-to-last characters of the `word` into a new `list` called `middle`.
14. Shuffle the `middle`.
15. Set the `word` equal to the first character plus the middle plus the last character.
16. Return the `word` which may have been altered if it met the criteria.
17. The test for the `scramble` function.

Discussion

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

Processing the text

As mentioned in the introduction, I often will 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])?)")
```

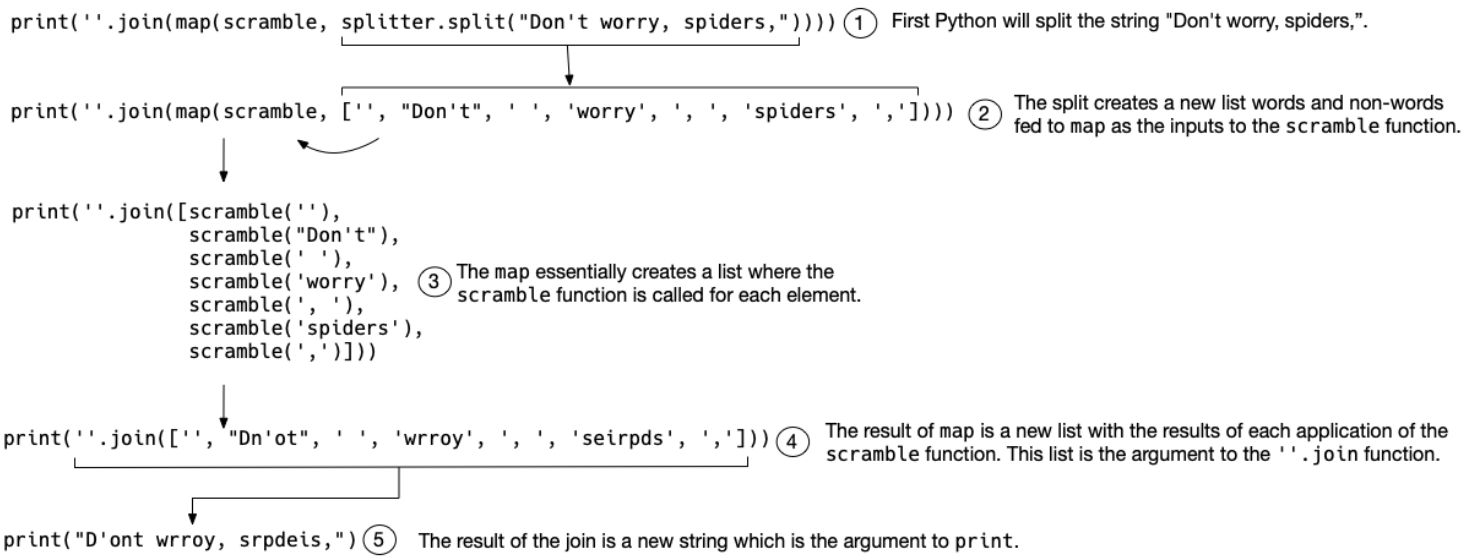
The other reason I like to use `re.compile` is because I feel it can make my code more readable. Without this, I would have to write:

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

I find this 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))))
```

Maybe you still find that code somewhat confusing. Let's diagram the flow of the data:



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

```
for line in args.text.splitlines():    1
    words = []                         2
    for word in splitter.split(line):  3
        words.append(scramble(word))   4
    print(''.join(words))              5
```

1. Use `str.splitlines()` to preserve the original line breaks.
2. For each `line` of input, create a `words` variable to hold the scrambled words.
3. Use the `splitter` to split the `line`.
4. Add the result of `scramble(word)` to the `words` list.
5. Join the `words` on the empty string and pass to `print`.

You could write this with a list comprehension, too:

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

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

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

The last one reminds me of a programmer I used to work 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:

```
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.

Scrambling a word

Let's take a closer look at my `scramble` function because I wrote it in a way that would make it easy to incorporate into a `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):     1
        middle = list(word[1:-1])                    2
        random.shuffle(middle)                       3
        word = word[0] + ''.join(middle) + word[-1]  4

    return word                                      5
```

1. Check if the given `word` is one we ought to scramble. First, it must be longer than 3 characters. Second, it must contain one or more word characters because the function will be passed both "word" and "non-word" strings. If either check returns `False`, we will return the `word` unchanged. The `r'\w+'` is used to create a "raw" string. Note that regex works fine with or without being a raw string, but `pylint` complains about an "invalid escape character" unless it is a raw string.
2. Copy the middle of the word to a new `list` called `middle`.
3. Shuffle the `middle` *in-place*. Remember that this function returns `None`.
4. Reconstruct the `word` by joining together the first character, the shuffled `middle`, and the last character.
5. Return the `word` which may or may not have been shuffled.

One key mistake you could make would be writing this code:

```
middle = random.shuffle(middle)
```

Honestly, I would prefer that `random.shuffle` accept a `list` (or `str`) and return a new, shuffled `list` (or `str`) rather than mutating the given argument, but c'est la vie. The function was written to mutate the data, and that's that.

Review

- The regex we used to split the text into words was quite complex but also gave us exactly what we needed. Writing the program without this piece would have made it 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.

Going Further

- Write a version 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.
- Write a program to *unscramble* the text. For this, you'd probably need to have a dictionary of English words — most Unix systems have a file at `/usr/share/dict/words` for this. You would have to split the scrambled text into words/non-words, then compare each "word" to the words in your dictionary. I would recommend you start by comparing the words as anagrams and then using the first and last letters to positively identify the unscrambled word.

Last updated 2020-01-14 20:38:12 -0700