15

The Kentucky Friar: More regular expressions

I grew up in the American Deep South where we tend to drop the final "g" of words ending in "ing," like "cookin'" instead of "cooking." We also tend to say "y'all" for the second-person plural pronoun, which makes sense because Standard English is missing a distinctive word for this. In this exercise, we'll write a program called friar.py that will accept some input as a single positional argument and transform the text by replacing the final "g" with an apostrophe (') for two-syllable words ending in "ing" and changing "you" to "y'all." Granted, we have no way to know if we're changing the first- or second-person "you," but it makes for a fun challenge nonetheless.

Figure 15.1 is a string diagram that will help you see the inputs and outputs. When run with no arguments or with the `-h` or `--help` flags, your program should present the following usage statement:

```
$ ./friar.py -h
usage: friar.py [-h] text

Southern fry text

positional arguments:
  text        Input text or file

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

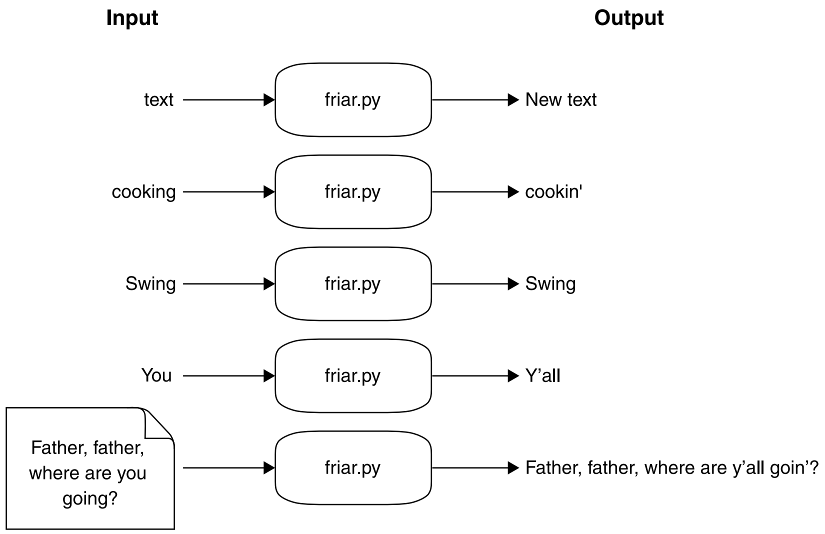


Figure 15.1 Our program will modify the input text to give it a Southern lilt.

We will only change "-ing" words with *two syllables*, so "cooking" becomes "cookin'" but "swing" will stay the same. Our heuristic for identifying two-syllable "-ing" words is to inspect the part of the word before the "-ing" ending to see if it contains a vowel, which in this case will include "y." We can split "cooking" into "cook" and "ing," and because there is an "o" in "cook," we should drop the final "g":

```
$ ./friar.py Cooking
Cookin'
```

When we remove "ing" from "swing," though, we're left with "sw," which contains no vowel, so it will remain the same:

```
$ ./friar.py swing
swing
```

When changing "you" to "y'all," be mindful to keep the case the same on the first letter. For example, "You" should become "Y'all":

```
$ ./friar.py you
y'all
$ ./friar.py You
Y'all
```

As in several previous exercises, the input may name a file, in which case you should read the file for the input text. To pass the tests, you will need to preserve the line structure of the input, so I recommend you read the file line by line. Given this input,

```
$ head -2 inputs/banner.txt
O! Say, can you see, by the dawn's early light,
What so proudly we hailed at the twilight's last gleaming -
```

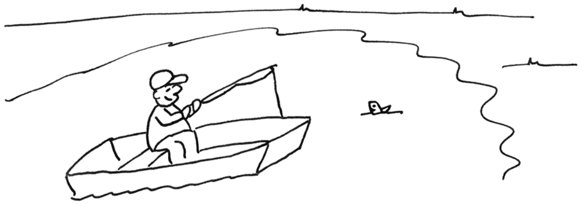
the output should have the same line breaks:

```
$ ./friar.py inputs/banner.txt | head -2
O! Say, can y'all see, by the dawn's early light,
What so proudly we hailed at the twilight's last gleamin' -
```

To me, it's quite amusing to transform texts this way, but maybe I'm just weird:

```
$ ./friar.py inputs/raven.txt
Presently my soul grew stronger; hesitatin' then no longer,
"Sir," said I, "or Madam, truly your forgiveness I implore;
But the fact is I was nappin', and so gently y'all came rappin',
And so faintly y'all came tappin', tappin' at my chamber door,
That I scarce was sure I heard y'all" - here I opened wide the door: -
Darkness there and nothin' more.
```

In this exercise you will



- Learn more about using regular expressions
- Use both `re.match()` and `re.search()` to find patterns anchored to the beginning of a string or anywhere in the string, respectively
- Learn how the `$` symbol in a regex anchors a pattern to the *end* of a string
- Learn how to use `re.split()` to split a string
- Explore how to write a manual solution for finding two-syllable "-ing" words or the word "you"

15.1 Writing friar.py

As usual, I recommend you start with `new.py friar.py` or copy the template/template.py file to 15_friar/friar.py. I suggest you start with a simple version of the program that echoes back the input from the command line:

```
$ ./friar.py cooking
cooking
```

Or from a file:

```
$ ./friar.py inputs/blake.txt
Father, father, where are you going?
 Oh do not walk so fast!
Speak, father, speak to your little boy,
 Or else I shall be lost.
```

We need to process the input line by line, and then word by word. You can use the `str.splitlines()` method to get each line of the input, and then use the `str.split()` method to break the line on spaces into word-like units. This code,

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

should create this output:

```
$ ./friar.py tests/blake.txt
['Father,', 'father,', 'where', 'are', 'you', 'going?']
['Oh', 'do', 'not', 'walk', 'so', 'fast!']
['Speak,', 'father,', 'speak', 'to', 'your', 'little', 'boy,']
['Or', 'else', 'I', 'shall', 'be', 'lost.']
```

If you look closely, it's going to be difficult to handle some of these word-like units because the adjacent punctuation is still attached to the words, as in `'Father,'` and `'going?'` Splitting the text on spaces is not sufficient, so I'll show you how to split the text *using a regular expression.*

15.1.1 Splitting text using regular expressions

As in chapter 14, we need to `import re` to use regexes:

```
>>> import re
```

For demonstration purposes, I'm going to set `text` to the first line:

```
>>> text = 'Father, father, where are you going?'
```

By default, `str.split()` breaks text on spaces. Note that whatever text is used for splitting will be missing from the result, so here there are no spaces:

```
>>> text.split()
['Father,', 'father,', 'where', 'are', 'you', 'going?']
```

You can pass an optional value to `str.split()` to indicate the string you want to use for splitting. If we choose the comma, we'll end up with three strings instead of six. Note that there are no commas in the resulting list, as that is the argument to `str.split()`:

```
>>> text.split(',')
['Father', ' father', ' where are you going?']
```

The `re` module has a function called `re.split()` that works similarly. I recommend you read `help(re.split)`, as this is a very powerful and flexible function. Like `re.match()`, which we used in chapter 14, this function wants at least a `pattern` and a `string`. We can use `re.split()` with a comma to get the same output as `str.split()`, and, as before, the commas are missing from the result:

```
>>> re.split(',', text)
['Father', ' father', ' where are you going?']
```

15.1.2 Shorthand classes

We are after the things that look like "words," in that they are composed of the characters that normally occur in words. The characters that *don't* normally occur in words (things like punctuation) are what we want to use for splitting. You've seen before that we can create a *character class* by putting literal values inside square brackets, like `'[aeiou]'` for the vowels. What if we create a character class where we enumerate all the non-letter characters? We could do something like this:

```
>>> import string
>>> ''.join([c for c in string.printable if c not in string.ascii_letters])
'0123456789!"#$%&\'()*+,-./:;<=>?@[\\]^_`{|}~ \t\n\r\x0b\x0c'
```

That won't be necessary, because almost every implementation of regular expression engines define shorthand character classes. Table 15.1 lists some of the most common shorthand classes and how they can be written longhand.

Table 15.1 Regex shorthand classes

Character class	Shorthand	Other ways to write the class
Digits	`\d`	`[0123456789]`, `[0-9]`
Whitespace	`\s`	`[ \t\n\r\x0b\x0c]`, same as `string.whitespace`
Word characters	`\w`	`[a-zA-Z0-9_-]`

NOTE There is a basic flavor of regular expression syntax that is recognized by everything from Unix command-line tools like `awk` to regex support inside of languages like Perl, Python, and Java. Some tools add extensions to their regexes that may not be understood by other tools. For example, there was a time when Perl's regex engine added many new ideas that eventually became a dialect known as "PCRE" (Perl-Compatible Regular Expressions). Not every tool that understands regexes will understand every flavor of regex, but in all my years of writing and using regexes, I've rarely found this to be a problem.

The shorthand `\d` means any *digit* and is equivalent to `'[0123456789]'`. I can use the `re.search()` method to look anywhere in a string for any digit. In the following example, it will find the character `'1'` in the string `'abc123!'` because this is the first digit in the string (see figure 15.2):

```
>>> re.search('\d', 'abc123!')
<re.Match object; span=(3, 4), match='1'>
```

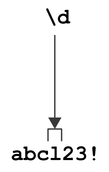


Figure 15.2 The digit shorthand will match any single digit.

That is the same as using the longhand version (see figure 15.3):

```
>>> re.search('[0123456789]', 'abc123!')
<re.Match object; span=(3, 4), match='1'>
```

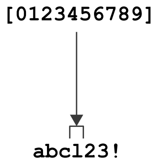


Figure 15.3 We can also create a character class enumerating all the digits.

It's also the same as the version that uses the range of characters `'[0-9]'` (see figure 15.4):

```
>>> re.search('[0-9]', 'abc123!')
<re.Match object; span=(3, 4), match='1'>
```

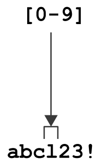


Figure 15.4 Character classes can use a range of contiguous values, like 0–9.

To have it find *one or more digits in a row*, add the `+` (see figure 15.5):

```
>>> re.search('\d+', 'abc123!')
<re.Match object; span=(3, 6), match='123'>
```

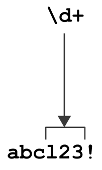


Figure 15.5 The plus signs means to match one or more of the preceding expression.

The `\w` shorthand means "any word-like character." It includes all the Arabic numbers, the letters of the English alphabet, the dash (`'-'`), and the underscore (`'_'`). The first match in the string is `'a'` (see figure 15.6):

```
>>> re.search('\w', 'abc123!')
<re.Match object; span=(0, 1), match='a'>
```

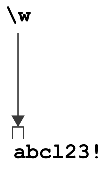


Figure 15.6 The shorthand for word characters is `\w`.

If you add the + as in figure 15.7, it matches one or more word characters in a row, which includes abc123 but not the exclamation mark (!):

```
>>> re.search('\w+', 'abc123!')
<re.Match object; span=(0, 6), match='abc123'>
```

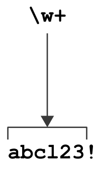


Figure 15.7 Add the plus sign to match one or more word characters.

15.1.3 Negated shorthand classes

You can complement or "negate" a character class by putting the caret (^) *immediately inside* the character class as in figure 15.8. One or more of any character *not* a digit is '[^0-9]+'. With it, 'abc' is found:

```
>>> re.search('[^0-9]+', 'abc123!')
<re.Match object; span=(0, 3), match='abc'>
```

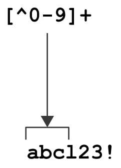


Figure 15.8 A caret just inside a character class will negate or complement the characters. This regex matches non-digits.

The shorthand class of non-digits [^0-9]+ can also be written as \D+ as in figure 15.9:

```
>>> re.search('\D+', 'abc123!')
<re.Match object; span=(0, 3), match='abc'>
```

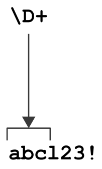


Figure 15.9 The shorthand \D+ matches one or more non-digits.

The shorthand for non-word characters is \W, which will match the exclamation point (see figure 15.10):

```
>>> re.search('\W', 'abc123!')
<re.Match object; span=(6, 7), match='!'>
```

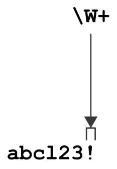


Figure 15.10 The \W will match anything that is *not* a letter, digit, underscore, or dash.

Table 15.2 summarizes these shorthand classes and how they can be expanded.

Table 15.2 Negated regex shorthand classes

Character class	Shorthand	Other ways to write the class
Not a digit	`\D`	`[^0123456789]`, `[^0-9]`
Not whitespace	`\S`	`[^ \t\n\r\x0b\x0c]`
Not word characters	`\W`	`[^a-zA-Z0-9_-]`

15.1.4 Using re.split() with a captured regex

We can use `\W` as the argument to `re.split()`:

```
>>> re.split('\W', 'abc123!')
['abc123', '']
```

NOTE Pylint will complain if we use `'\W'` in a regular expression in our program, returning the message "Anomalous backslash in string: `'\W'`. String constant might be missing an `r` prefix." We can use the `r` prefix to create a "raw" string, one where Python does not try to interpret the `\W` as it will, for instance, interpret `\n` to mean a newline or `\r` to mean a carriage return. From this point on, I will use the r-string syntax to create a raw string.

There is a problem, though, because the result of `re.split()` *omits those strings matching the pattern*. Here we've lost the exclamation point! If we read `help(re.split)` closely, we can find the solution:

> *If* ***capturing parentheses are used in [the] pattern****, then the text of all groups in the pattern are also returned as part of the resulting list.*

We used capturing parentheses in chapter 14 to tell the regex engine to "remember" certain patterns, like the consonant(s), vowel, and the rest of a word. When the regex matched, we were able to use `match.groups()` to retrieve strings that were found by the patterns. Here we will use the parentheses around the pattern to `re.split()` so that the strings matching the pattern will also be returned:

```
>>> re.split(r'(\W)', 'abc123!')
['abc123', '!', '']
```

If we try that on our `text`, the result is a `list` of strings that match and do not match the regular expression:

```
>>> re.split(r'(\W)', text)
['Father', ',', '', ' ', 'father', ',', '', ' ', 'where', ' ', 'are', ' ',
     'you', ' ', 'going', '?', '']
```

I'd like to group all the non-word characters together by adding + to the regex (see figure 15.11):

```
>>> re.split(r'(\W+)', text)
['Father', ', ', 'father', ', ', 'where', ' ', 'are', ' ', 'you', ' ', 'going
     ', '?', '']
```

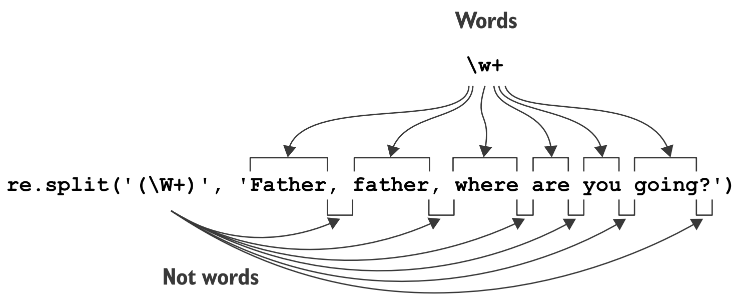


Figure 15.11 The `re.split()` function can use a captured regex to return both the parts that match the regex and those that do not.

That is so cool! Now we have a way to process each *actual* word and the bits in between them.

15.1.5 Writing the fry() function

Our next step is to write a function that will decide whether and how to modify *just one word*. That is, rather than thinking about how to handle all the text at once, we'll think about how to handle one word at a time. We can call this function `fry()`.

To help us think about how this function should work, let's start off by writing the `test_fry()` function and a stub for the actual `fry()` function that contains just the single command `pass`, which tells Python to do nothing. To get started on this, you can paste this into your program:

```
def fry(word):
    pass

def test_fry():
    assert fry('you') == "y'all"
    assert fry('You') == "Y'all"
    assert fry('fishing') == "fishin'"
    assert fry('Aching') == "Achin'"
    assert fry('swing') == "swing"
```

pass is a way to do nothing. You might call it a "no-operation" or "NO-OP," which kind of looks like "NOPE," which is another way to remember that it does nothing. We're just defining this fry() function as a placeholder so we can write the test.

The word "you" should become "y'all."

Ensure the word's capitalization is preserved.

This is a two-syllable "-ing" word that should be changed by dropping the final "g" for an apostrophe.

This is a two-syllable "-ing" word that starts with a vowel. It should likewise be changed.

This is a one-syllable "-ing" word that should not be changed.

The test_fry() function will pass in words we expect to be changed or not. We can't check every word, so we'll rely on spot-checking the major cases.

Now run `pytest friar.py` to see that, as expected, the test will fail:

```
=================================== FAILURES ===================================
___________________________________ test_fry ___________________________________

    def test_fry():
>       assert fry('you') == "y'all"
E       assert None == "y'all"
E        +  where None = fry('you')

friar.py:47: AssertionError
========================== 1 failed in 0.08 seconds ============================
```

The first test is failing.

The result of fry('you') was None, which does not equal "y'all."

Let's change our `fry()` function to handle that string:

```
def fry(word):
    if word == 'you':
        return "y'all"
```

Now let's run our tests again:

```
=================================== FAILURES ===================================
___________________________________ test_fry ___________________________________

    def test_fry():
        assert fry('you') == "y'all"
>       assert fry('You') == "Y'all"
E       assert None == "Y'all"
E        +  where None = fry('You')

friar.py:49: AssertionError
========================== 1 failed in 0.16 seconds ============================
```

Now the first test passes.

The second test fails because the "You" is capitalized.

The function returned None but should have returned "Y'all."

Let's handle those:

```
def fry(word):
    if word == 'you':
        return "y'all"
    elif word == 'You':
        return "Y'all"
```

If you run the tests now, you'll see that the first two tests pass; however, I'm definitely not happy with that solution. There is already a good bit of duplicated code. Can we find a more elegant way to match both "you" and "You" and still return the correctly capitalized answer? Yes, we can!

```
def fry(word):
    if word.lower() == 'you':
        return word[0] + "'all"
```

Better still, we can write a regular expression! There is one difference between “you” and “You”—the “y” or “Y”—that we can represent using the character class `'[yY]'` (see figure 15.12). This will match the lowercase version:

```
>>> re.match('[yY]ou', 'you')
<re.Match object; span=(0, 3), match='you'>
```

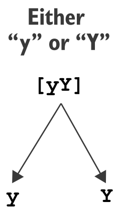


Figure 15.12 We can use a character class to match lower- and uppercase Y.

It will also match the capitalized version (see figure 15.13):

```
>>> re.match('[yY]ou', 'You')
<re.Match object; span=(0, 3), match='You'>
```

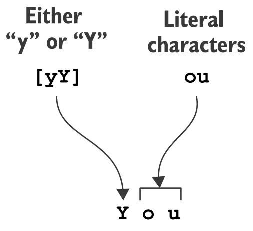


Figure 15.13 This regex will match “you” and “You.”

Now we want to reuse the initial character (either “y” or “Y”) in the return value. We could *capture* it by placing it into parentheses. Try to rewrite your `fry()` function using this idea, and getting it to pass the first two tests again, before moving on:

```
>>> match = re.match('([yY])ou', 'You')
>>> match.group(1) + "'all"
"Y'all"
```

The next step is to handle a word like “fishing”:

```
================================== FAILURES ===================================
__________________________________ test_fry ___________________________________

    def test_fry():
        assert fry('you') == "y'all"
        assert fry('You') == "Y'all"
>       assert fry('fishing') == "fishin'"
E       assert None == "fishin'"
E        +  where None = fry('fishing')
```

The third test fails.

The return from fry('fishing') was None, but the value “fishin'” was expected.

```
friar.py:52: AssertionError
========================== 1 failed in 0.10 seconds ===========================
```

How can we identify a word that ends with “ing”? With the `str.endswith()` function:

```
>>> 'fishing'.endswith('ing')
True
```

A regular expression to find “ing” at the end of a string would use `$` (pronounced “dollar”) at the end of the expression to *anchor* the expression to the end of the string (see figure 15.14):

```
>>> re.search('ing$', 'fishing')
<re.Match object; span=(4, 7), match='ing'>
```

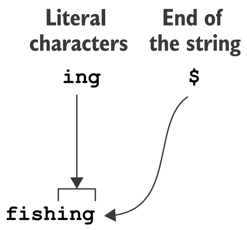


Figure 15.14 The dollar sign indicates the end of the word.

As shown in figure 15.15, we can use a string slice to get all the characters up to the last at index `-1` and then append an apostrophe.

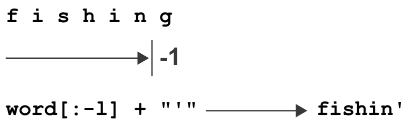


Figure 15.15 Use a string slice to get all the letters up to the last one and add an apostrophe.

Add this to your `fry()` function and see how many tests you pass:

```
if word.endswith('ing'):
    return word[:-1] + "'"
```

Or you could use a group within the regex to capture the first part of the word (see figure 15.16):

```
>>> match = re.search('(.+)ing$', 'fishing')
>>> match.group(1) + "in'"
"fishin'"
```

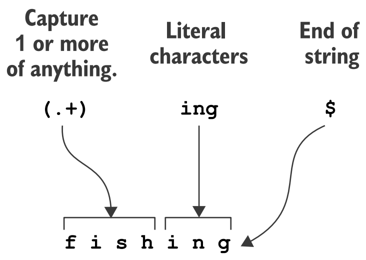


Figure 15.16 Using a capture group so we can access the matching string

You should be able to get results like this:

```
=================================== FAILURES ===================================
___________________________________ test_fry ___________________________________

    def test_fry():
        assert fry('you') == "y'all"
        assert fry('You') == "Y'all"
        assert fry('fishing') == "fishin'"
        assert fry('Aching') == "Achin'"
>       assert fry('swing') == "swing"
E       assert "swin'" == 'swing'
E         - swin'
E         ?     ^
E         + swing
E         ?     ^

friar.py:59: AssertionError
=========================== 1 failed in 0.10 seconds ===========================
```

This test failed.

The result of fry('swing') was "swin'," but it should have been "swing."

Sometimes the test results will be able to highlight the exact point of failure. Here you are being shown that there is an apostrophe (') where there should be a "g."

We need a way to identify words that have two syllables. I mentioned before that we'll use a heuristic that looks for a vowel, `'[aeiouy]'`, in the part of the word *before* the "ing" ending, as shown in figure 15.17. Another regex could do the trick:

Here we know there will be a match value, so we can use match.group(1) to get the first capture group, which will be anything immediately before "ing." In actual code, we should check that match is not None or we'd trigger an exception by trying to execute the group method on a None.

The (.+) will match and capture one or more of anything followed by the characters "ing." The return from re.search() will either be a re.Match object if the pattern was found or None to indicate it was not.

```
>>> match = re.search('(.+)ing$', 'fishing')
>>> first = match.group(1)
>>> re.search('[aeiouy]', first)
<re.Match object; span=(1, 2), match='i'>
```

We can use re.search() on the first part of the string to look for a vowel.

As the return from re.search() is a re.Match object, we know there is a vowel in the first part, so the word looks to have two syllables.

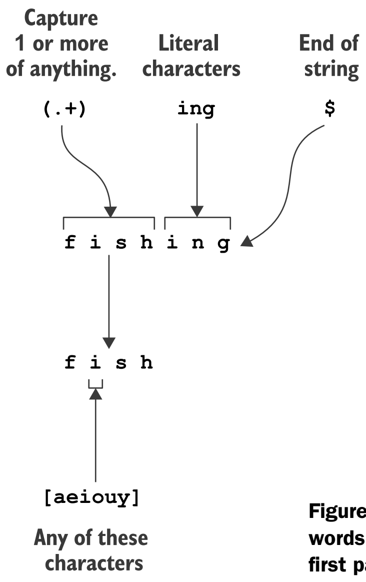


Figure 15.17 A possible way to find two-syllable words ending in "ing" is to look for a vowel in the first part of the word.

If the word matches this test, return the word with the final "g" replaced with an apostrophe; otherwise, return the word unchanged. I suggest you not proceed until you are passing all of `test_fry()`.

15.1.6 Using the fry() function

Now your program should be able to

1. Read input from the command line or a file
2. Read the input line by line
3. Split each line into words and non-words
4. `fry()` any individual word

The next step is to apply the `fry()` function to all the word-like units. I hope you can see a familiar pattern emerging—applying a function to all elements of a list! You can use a `for` loop:

```
for line in args.text.splitlines():
    words = []
    for word in re.split(r'(\W+)', line.rstrip()):
        words.append(fry(word))
    print(''.join(words))
```

Preserve the structure of the newlines in args.text by using str.splitlines().

Create a words variable to hold the transformed words.

Split each line into words and non-words.

Add the fried word to the words list.

Print a new string of the joined words.

That (or something like it) should work well enough to pass the tests. Once you have a version that works, see if you can rewrite the `for` loop as a list comprehension and a `map()`.

Alrighty! Time to bear down and write this.

15.2 Solution

This reminds me of when Robin Hood's mate Friar Tuck was captured by the Sheriff of Nottingham. The Friar was sentenced to be boiled in oil, to which he replied "You can't boil me, I'm a friar!"

```
#!/usr/bin/env python3
"""Kentucky Friar"""

import argparse
import os
import re


# --------------------------------------------------
def get_args():
    """get command-line arguments"""
    parser = argparse.ArgumentParser(
        description='Southern fry text',
        formatter_class=argparse.ArgumentDefaultsHelpFormatter)

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

    args = parser.parse_args()

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

    return args


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

    args = get_args()

    for line in args.text.splitlines():
        print(''.join(map(fry, re.split(r'(\W+)', line.rstrip()))))


# --------------------------------------------------
def fry(word):
    """Drop the `g` from `-ing` words, change `you` to `y'all`"""
```

If the argument is a file, replace the text value with the contents from the file.

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

Get the command-line arguments. The text value will either be the command-line text or the contents of a file by this point.

Define a fry() function that will handle one word.

Map the pieces of text split by the regular expression through the fry() function, which will return the words modified as needed. Use str.join() to turn that resulting list back into a string to print.

```
    ing_word = re.search('(.+)ing$', word)
    you = re.match('([Yy])ou$', word)

    if ing_word:
        prefix = ing_word.group(1)
        if re.search('[aeiouy]', prefix, re.IGNORECASE):
            return prefix + "in'"
    elif you:
        return you.group(1) + "'all"

    return word


# --------------------------------------------------
def test_fry():
    """Test fry"""

    assert fry('you') == "y'all"
    assert fry('You') == "Y'all"
    assert fry('fishing') == "fishin'"
    assert fry('Aching') == "Achin'"
    assert fry('swing') == "swing"


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

Check if the search for "ing" returned a match.

Get the prefix (the bit before the "ing"), which is in group number 1.

Return the captured first character plus "'all."

Otherwise, return the word unaltered.

The tests for the fry() function

Check if the match for "you" succeeded.

Append "in'" to the prefix and return it to the caller.

Perform a case-insensitive search for a vowel (plus "y") in the prefix. If nothing is found, None will be returned, which evaluates to False in this Boolean context. If a match is returned, the not-None value will evaluate to True.

Search for "you" or "You" starting from the beginning of word. Capture the [yY] alternation in a group.

Search for "ing" anchored to the end of word. Use a capture group to remember the part of the string before the "ing."

15.3 Discussion

Again, there is nothing new in `get_args()`, so let's just move to breaking the text into lines. In several previous exercises, I used a technique of reading an input file into the `args.text` value. If the input is coming from a file, there will be newlines separating each line of text. I suggested using a `for` loop to handle each line of input text returned by `str.splitlines()` to preserve the newlines in the output. I also suggested you start with a second `for` loop to handle each word-like unit returned by the `re.split()`:

```
for line in args.text.splitlines():
    words = []
    for word in re.split(r'(\W+)', line.rstrip()):
        words.append(fry(word))
    print(''.join(words))
```

That's five lines of code that could be written in two if we replace the second `for` with a list comprehension:

```
for line in args.text.splitlines():
    print(''.join([fry(w) for w in re.split(r'(\W+)', line.rstrip())]))
```

Or it could be slightly shorter using a `map()`:

```
for line in args.text.splitlines():
    print(''.join(map(fry, re.split(r'(\W+)', line.rstrip()))))
```

One other way to slightly improve readability is to use the `re.compile()` function to compile the regular expression. When you use the `re.split()` function inside the `for` loop, the regex must be compiled anew each iteration. By compiling the regex first, the compilation happens just once, so your code is (maybe just slightly) faster. More importantly, though, I think this is slightly easier to read, and the benefits are greater when the regex is more complicated:

```
splitter = re.compile(r'(\W+)')
for line in args.text.splitlines():
    print(''.join(map(fry, splitter.split(line.rstrip()))))
```

15.3.1 Writing the fry() function manually

You were not required, of course, to write a `fry()` function. However you wrote your solution, I hope you wrote tests for it!

The following version is fairly close to some of the suggestions I made earlier in the chapter. This version uses no regular expressions:

```
def fry(word):
    """Drop the `g` from `-ing` words, change `you` to `y'all`"""

    if word.lower() == 'you':
        return word[0] + "'all"

    if word.endswith('ing'):
        if any(map(lambda c: c.lower() in 'aeiouy', word[:-3])):
            return word[:-1] + "'"
        else:
            return word

    return word
```

Force the word to lowercase and see if it matches "you."

If so, return the first character (to preserve the case) plus "'all."

Check if the word ends with "ing."

Check if it's True that any of the vowels are in the word up to the "ing" suffix.

If so, return the word up to the last index plus the apostrophe.

Otherwise, return the word unchanged.

If the word is neither an "ing" or "you" word, return it unchanged.

Let’s take a moment to appreciate the `any()` function as it’s one of my favorites. The preceding code uses a `map()` to check if each of the vowels exists in the portion of the `word` before the “ing” ending:

```
>>> word = "cooking"
>>> list(map(lambda c: (c, c.lower() in 'aeiouy'), word[:-3]))
[('c', False), ('o', True), ('o', True), ('k', False)]
```

The first character of “cooking” is “c,” and it does not appear in the string of vowels. The next two characters (“o”) do appear in the vowels, but “k” does not.

Let’s reduce this to just the `True`/`False` values:

```
>>> list(map(lambda c: c.lower() in 'aeiouy', word[:-3]))
[False, True, True, False]
```

Now we can use `any` to tell us if *any* of the values are `True`:

```
>>> any([False, True, True, False])
True
```

It’s the same as joining the values with `or`:

```
>>> False or True or True or False
True
```

The `all()` function returns `True` only if *all* the values are true:

```
>>> all([False, True, True, False])
False
```

That’s the same as joining those values on `and`:

```
>>> False and True and True and False
False
```

If it’s `True` that one of the vowels appears in the first part of the `word`, we have determined that this is (probably) a two-syllable word, and we can return the `word` with the final “g” replaced with an apostrophe. Otherwise, we return the unaltered `word`:

```
if any(map(lambda c: c.lower() in 'aeiouy', word[:-3])):
    return word[:-1] + "'"
else:
    return word
```

This approach works fine, but it’s quite manual as we have to write quite a bit of code to find our patterns.

15.3.2 Writing the fry() function with regular expressions

Let's revisit the version of the `fry()` function that uses regular expressions:

The re.match() starts matching at the beginning of the given word, and it is looking for either an upper- or lowercase "y" followed by "ou" and then the end of the string ($).

We use re.search() to look anywhere in the prefix for any of the vowels (plus "y") in a case-insensitive fashion. Remember that re.match() would start at the beginning of word, which is not what we want.

The pattern '(.+)ing$' matches one or more of anything followed by "ing." The dollar sign anchors the pattern to the end of the string, so this is looking for a string that ends in "ing," but the string cannot just be "ing" as it has to have at least one of something before it. The parentheses capture the part before the "ing."

```
def fry(word):
    """Drop the `g` from `-ing` words, change `you` to `y'all`"""

    ing_word = re.search('(.+)ing$', word)
    you = re.match('([Yy])ou$', word)

    if ing_word:
        prefix = ing_word.group(1)
        if re.search('[aeiouy]', prefix, re.IGNORECASE):
            return prefix + "in'"
    elif you:
        return you.group(1) + "'all"

    return word
```

Return the prefix plus the string "in'" so as to drop the final "g."

If re.match() for the "you" pattern fails, then "you" will be None. If it is not None, then it matched, and "you" is a re.Match object.

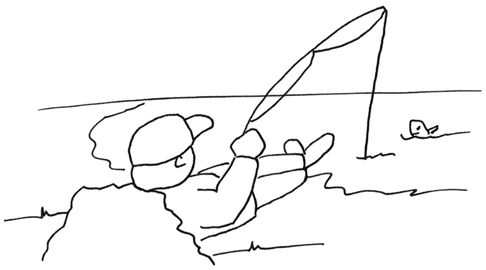
The prefix is the bit before the "ing" that we wrapped in parentheses. Because it is the first set of parentheses, we can fetch it with ing_word.group(1).

We used parentheses to capture the first character so as to maintain the case. That is, if the word was "You," we want to return "Y'all." Here we return that first group plus the string "'all."

If ing_word is None, that means it failed to match. If it is not None (so it is "truthy"), that means it is a re.Match object we can use.

If the word matched neither a two-syllable "ing" pattern or the word "you," we return the word unchanged.

I've been using regexes for maybe 20 years, so this version seems much simpler to me than the manual version. You may feel differently. If you are completely new to regexes, trust me that they are so very worth the effort to learn. I absolutely would not be able to do much of my work without them.



15.4 Going further

- You could also replace "your" with "y'all's." For instance, "Where are your britches?" could become "Where are y'all's britches?"
- Change "getting ready" or "preparing" to "fixin'," as in "I'm getting ready to eat" to "I'm fixin' to eat." Also change the string "think" to "reckon," as in "I

think this is funny" to "I reckon this is funny." You should also change "thinking" to "reckoning," which then should become "reckonin'." That means you either need to make two passes for the changes or find both "think" and "thinking" in the one pass.

- Make a version of the program for another regional dialect. I lived in Boston for a while and really enjoyed saying "wicked" all the time instead of "very," as in "IT'S WICKED COLD OUT!"

Summary

- Regular expressions can be used to find patterns in text. The patterns can be quite complicated, like a grouping of non-word characters in between groupings of word characters.
- The `re` module has seriously handy functions like `re.match()` to find a pattern at the beginning of some text, `re.search()` to find a pattern anywhere inside some text, `re.split()` to break text on a pattern, and `re.compile()` to compile a regex so you can use it repeatedly.
- If you use capturing parentheses on the pattern for `re.split()`, the captured split pattern will be included in the returned values. This allows you to reconstruct the original string with the strings that are described by the pattern.