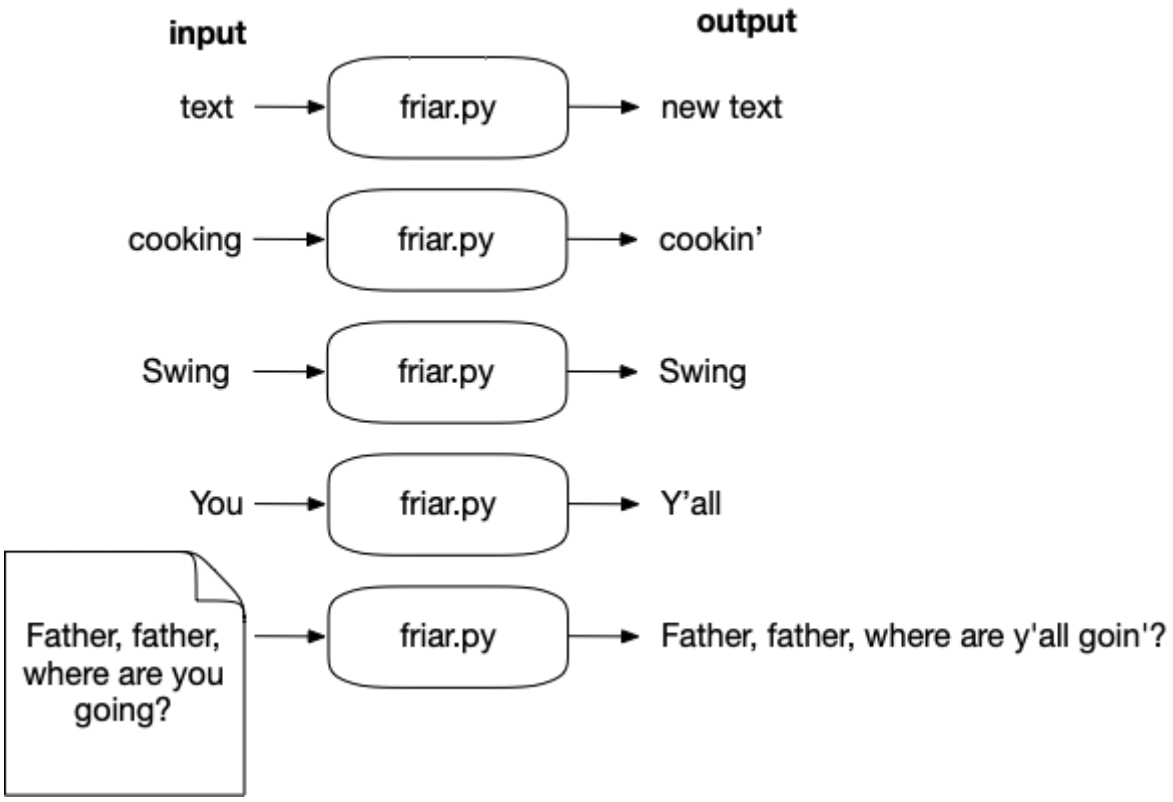
1. The Kentucky Friar: More regular expressions

Your author 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 transforms the text by replacing the "g" with an apostrophe (') words ending in "-ing" and also changes "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.

Here is a string diagram to help you see the inputs and outputs:



When run with no arguments or the `-h` or `--help` flags, your program should present usage:

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

Southern fry text

positional arguments:
  str         Input text or file

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

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"; since 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
```

Be mindful to keep the case the same on the first letter, e.g, "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. So given this input:

```
$ head -2 tests/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 tests/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 tests/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."

1.1. Writing `friar.py`

As usual, I would recommend you start with `new.py friar.py` or copy the `template/template.py` to `friar/friar.py`. I recommend you start with a simple version that echos back the input from the command line:

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

Or a file:

```
$ ./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.
```

We need to process the input line-by-line, then word-by-word. You can use the `str.splitlines` method to get each line of the input, and then 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())
```

Will 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 as the adjacent punctuation is still attached to the words as in `'Father,'` and `'going?'`. We could write a regular expression to find just the parts of each "word" composed of ASCII letters, but I'd like to show you another way to split the text *using a regular expression*!

1.1.1. Splitting text using regular expressions

As in "Rhymer," 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 is 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 splittin. If we choose the comma, then we end up with 3 strings instead of 6. Note that the commas which are used to split are missing from the result:

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

The `re` module has a function called `split` that works similarly. I recommend you read `help(re.split)` as this is a very powerful and flexible function. Like `re.match` that we used in "Rhymer," 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?']
```

1.1.2. 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. We'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'
```

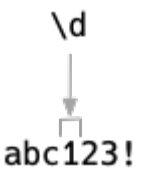
That won't be necessary because almost every implementation of regular expression engines define shorthand character classes.[1] Here is a table of some of the most common shorthand classes and how they can be written longhand:

Table 15. 1. Regex shorthand classes

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

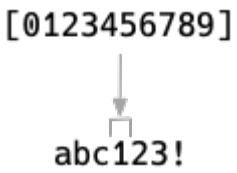
The shorthand `\d` means any *digit* and is equivalent to `'[0123456789]'`. I can use the `re.search` method to look anywhere in a string for any digit. Here it will find the character `'1'` in the string `'abc123!'` because this is the *first* digit it finds:

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



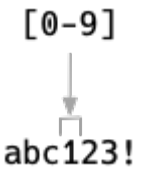
Which is the same as using the longhand version:

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



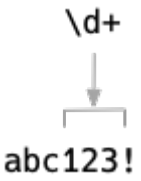
Or the version that uses the range of characters `'[0-9]'`:

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



To have it find *one or more digits in a row*, add the `+`:

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

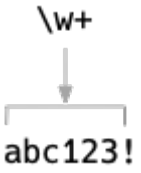


Similarly, `\w` means "any word-like character." It includes all the Arabic numbers, the letters of the English alphabet, the dash (`'-'`), and underscore (`'_'`). The first match in the string is `'a'`:

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

If you add the `+`, then it matches one or more word characters in a row which includes `abc123` but not the `!`:

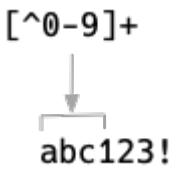
```
>>> re.search('\w+', 'abc123!')
<re.Match object; span=(0, 6), match='abc123'>
```



1.1.3. Negated shorthand classes

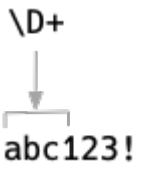
You can complement or "negate" a character class by putting the caret `^` *immediately inside* the character class. One or more of any character *not* a digit is `'[^0-9]+'`, and so `'abc'` is found:

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



The class `[^0-9]+` can also be written as `\D+`:

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



Therefore, anything *not* a word character is `\W`:

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

Table 15. 2. Negated regex shorthand classes

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

1.1.4. Using `re.split` with a captured regex

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

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

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

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

We used capturing parentheses in the "Rhymer" to tell the regex engine to "remember" certain patterns that we then use `re.Match.groups` to retreive after a match. Here it means that we need to

put the split pattern in parens:

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

Let's try that on our `text`:

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

I'd like to group all the non-word characters together by adding `+` to the regex:

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

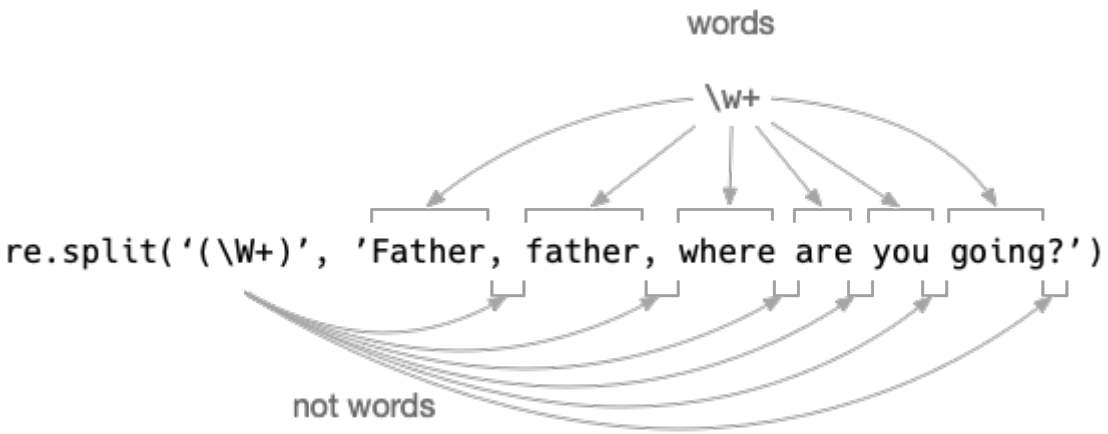


Figure 15. 1. The `re.split` function can use a captured regex to return both the parts that match the regex and the things that did not.

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

1.1.5. Writing the `fry` function

For me, the next step is to write function called `fry` to decide whether to modify *just one word*. That is, rather than thinking about how to handle all the text at once, I just want to think about how I will handle one word at a time. To help me think about how this function should work, I'll start off by writing the `test_fry` function and a stub for the actual `fry` function that has just the single command `pass` which tells Python to do nothing. If you like this idea, feel free to 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"
```

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

```
================================== FAILURES ==================================
_____________________________________ test_fry _____________________________________

    def test_fry():
>       assert fry('you') == "y'all" ①
E       assert None == "y'all"       ②
E        +  where None = fry('you')

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

① The first test is failing.

② The result of `fry('you')` was `None` which does not equal "y'all."

Let's change our `fry` program to handle that string:

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

And run our tests:

```
================================== FAILURES ==================================
_____________________________________ test_fry _____________________________________

    def test_fry():
        assert fry('you') == "y'all" ①
>       assert fry('You') == "Y'all" ②
E       assert None == "Y'all"       ③
E        +  where None = fry('You')

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

① Now the first test passes.

② The second test fails because the "You" is capitalized.

③ The function returned `None` but should have returned "Y'all."

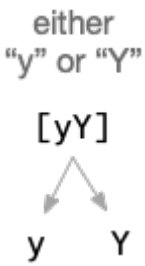
Let's handle that:

```
1 def fry(word):
2     if word == 'you':
3         return "y'all"
4     elif word == 'You':
5         return "Y'all"
```

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

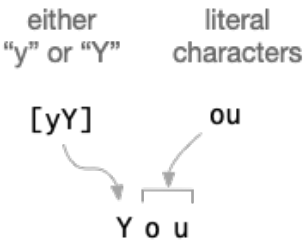
```
1 def fry(word):
2     if word.lower() == 'you':
3         return word[0] + "'all"
```

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]'`. This will match the lowercase version:




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

And the capitalized:




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

Now I'd like to reuse the initial character (either "y" or "Y") in the return value. I 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'" ①
E       assert None == "fishin'"           ②
E        +  where None = fry('fishing')

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

① The third test fails.

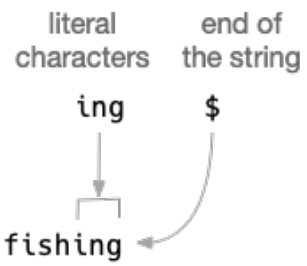
② The return from `fry('fishing')` was `None` but the value "fishin'" was expected.

How can we identify a word that ends with "ing." There is actually a `str.endswith` function:

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

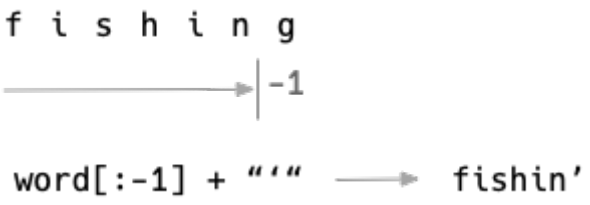
A regular expression to find "ing" at the end of a string would use `$` (pronounced "dollar") to *anchor* an expression:

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



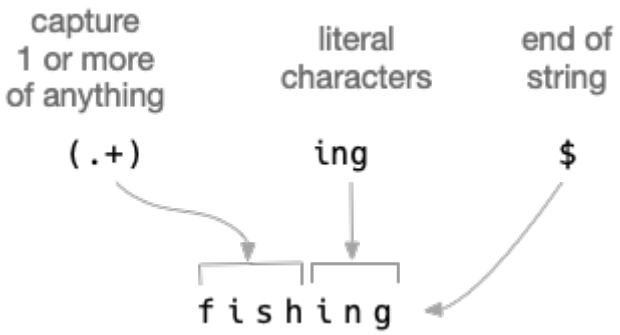
We can use a string slice to get all the characters up to the last at index `-1` and then append an apostrophe. Add this to your `fry` function and see how many tests you pass:

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



Or you could use a groups with a regex to capture the first part of the word:

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



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" ①
E       assert "swin'" == 'swing'      ②
E         - swin'  ③
E         ?     ^
E         + swing
E         ?     ^

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

① This test failed.

② The result of `fry('swing')` was "swin'" but should have been "swing."

③ Sometimes the test results will be able to highlight the exact point of failure. Here you are being shown that there is a `'` 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. Another regex could do the trick:

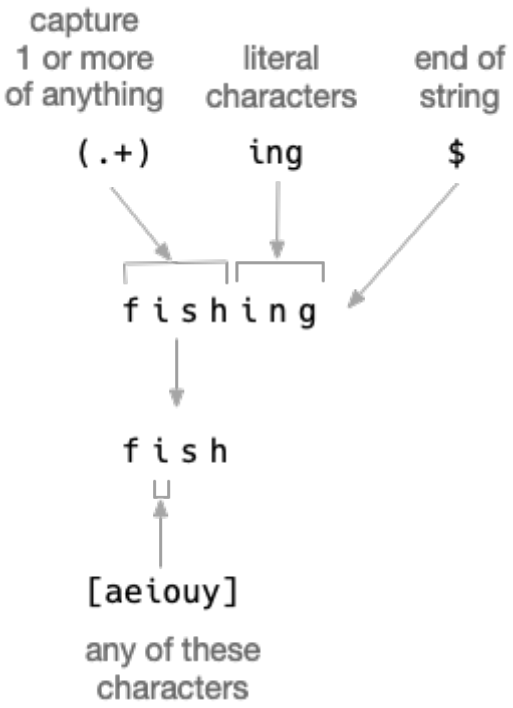
```
>>> match = re.search('(.+)ing$', 'fishing') ①
>>> first = match.group(1)                   ②
>>> re.search('[aeiouy]', first)             ③
<re.Match object; span=(1, 2), match='i'>    ④
```

① The `(.+)` will match and capture one or more of anything followed by the characters `ing`. The return from `re.search` will either be an `re.Match` object if the pattern was found or `None` to indicate it was not.

② 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`.

③ We can use `re.search` on the `first` part of the string to look for a vowel.

④ As the return from `re.search` is an `re.Match` object, we know there is a vowel in the first part, so the word looks to have two syllables.



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

1.1.6. 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():                ①
    words = []                                     ②
    for word in re.split(r'(\W+)', line.rstrip()): ③
        words.append(fry(word))                    ④
    print(''.join(words))                          ⑤
```

① Preserve the structure of the newlines in `args.text` by using `str.splitlines()`.

② Create a `words` variable to hold the transformed words.

③ Split each `line` into words and non-words.

④ Add the "fried" word to the `words` list.

⑤ 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.

1.2. Solution

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

    args = parser.parse_args()

    if os.path.isfile(args.text): ①
        args.text = open(args.text).read()

    return args


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

    args = get_args() ②

    for line in args.text.splitlines(): ③
        print(''.join(map(fry, re.split(r'(\W+)', line.rstrip())))) ④


# --------------------------------------------------
def fry(word): ⑤
    """Drop the `g` from `-ing` words, change `you` to `y'all`"""

    ing_word = re.search('(.+)ing$', word) ⑥
    you = re.match('([Yy])ou$', word)      ⑦

    if ing_word:                           ⑧
        prefix = ing_word.group(1)         ⑨
        if re.search('[aeiouy]', prefix, re.IGNORECASE): ⑩
            return prefix + "in'"          ⑪
    elif you:                              ⑫
        return you.group(1) + "'all"       ⑬
```

```
49
50      return word                                 ⑭
51
52
53  # --------------------------------------------------
54  def test_fry():                                 ⑮
55      """Test fry"""
56
57      assert fry('you') == "y'all"
58      assert fry('You') == "Y'all"
59      assert fry('fishing') == "fishin'"
60      assert fry('Aching') == "Achin'"
61      assert fry('swing') == "swing"
62
63
64  # --------------------------------------------------
65  if __name__ == '__main__':
66      main()
```

① If the argument is a file, replace the `text` value with the contents from the file.

② Get the command-line arguments. The `text` value will either be the command-line text or the contents of a file by this point.

③ Use the `str.splitlines()` method to preserve the line breaks in the input text.

④ `map` the pieces of text split by the regular expression through the `fry` function which will return the words modified as needed. Use the `str.join` method to turn that resulting `list` back into a `str` to print.

⑤ Define a `fry` function that will handle one `word`.

⑥ Search for "ing" anchored to the end of `word`. Use a capture to remember the part of the string *before* the "ing."

⑦ Search for "you" or "You" starting from the beginning of `word`. Capture the `[yY]` alternation in a group.

⑧ If the search for "ing" returned a match…

⑨ Get the `prefix` (the bit before the "ing") which is `group` number 1.

⑩ Perform a case-insensitive search for a vowel (plus "y") in the `prefix`. If nothing is found, then `None` will be returned which evaluates to `False` in this boolean context. If a match is returned, then the not-`None` value will evaluate to `True`.

⑪ Append "in'" to the `prefix` and return it to the caller.

⑫ If the match for "you" succeeded…

⑬ Then return the captured first character plus "'all."

⑭ Otherwise, return the `word` unaltered.

⑮ The tests for `fry`.

1.3. Discussion

1.3.1. Breaking 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, then there will be newlines separating each line of text. I had suggested a `for` loop to handle each line of input text returned by `str.splitlines()` to preserve the newlines in the output. I suggested you also 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('(\W+)', line.rstrip()):
        words.append(fry(word))
    print(''.join(words))
```

That's a total of 5 LOC that could be written in 2 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())]))
```

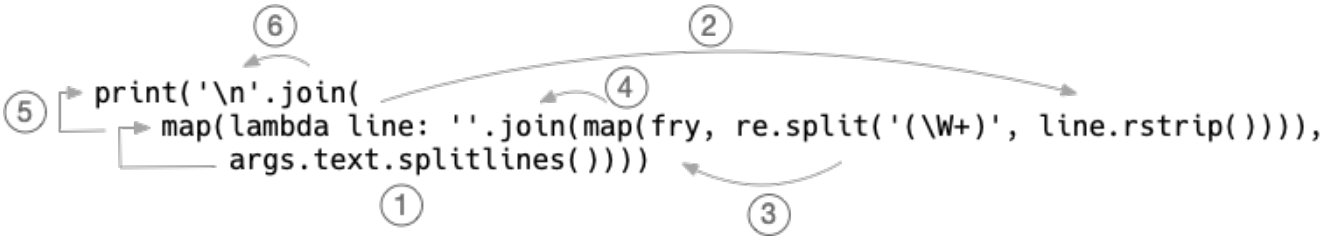
Or slightly shorter using a `map`:

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

Or, if you're just a little bit weird, you can replace both `for` loops with `map`:

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

Here's annotation to help you follow that:



1. `args.text` is split on newlines, each of which is passed to as `line` to the `lambda`.
2. The `line` is stripped of whitespace on the right and fed as the argument to `re.split`.
3. Each word-like unit from `re.split` is passed by `map` through the `fry` function.

4. The new `list` returned by `map` is joined on the empty string to create a new `str`.
5. The `map` returns a `list` of new `str` values created from the processed lines of text.
6. These new strings are joined back together on newlines to reconstitute the original line endings.

I find that last one a little hard to write and even harder to read, so I'd tend to use the version before it. Remember that code is *read* far more often than it is *written*. The last version is probably too clever by half.

1.3.2. 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. Here is a version that is fairly close to some of the suggestions I made in the introduction. This version uses no regular expressions:

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

    if word.lower() == 'you':      ①
        return word[0] + "'all"    ②

    if word.endswith('ing'):       ③
        if any(map(lambda c: c.lower() in 'aeiouy', word[:-3])): ④
            return word[:-1] + "'" ⑤
        else:
            return word            ⑥

    return word                    ⑦
```

① Force the `word` to lowercase and see if it matches "you."

② If so, return the first character (to preserve the case) plus "'all".

③ If the `word` ends with "ing"...

④ Check if it's `True` that any of the characters in our vowel set are in the `word` up to the "ing" suffix.

⑤ If so, return the `word` up to the last index plus the apostrophe.

⑥ Else return the `word` unchanged.

⑦ If the `word` is neither an "ing" or "you" word, then return it unchanged.

Let's take a moment to appreciate the `any` function as it's one of my favorites. Here I'm using 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]
```

And now we can use `any` to tells us if *any* of the values are `True`:

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

It's exactly 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
```

Which is 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 this is (probably) a two-syllable word and 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 all quite manual as we have to write quite a bit of code to find our patterns.

1.3.3. Writing the `fry` function with regular expressions

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

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

    ing_word = re.search('(.+)ing$', word) ①
    you = re.match('([Yy])ou$', word)      ②

    if ing_word:                           ③
        prefix = ing_word.group(1)         ④
        if re.search('[aeiouy]', prefix, re.IGNORECASE): ⑤
            return prefix + "in'"          ⑥
    elif you:                              ⑦
        return you.group(1) + "'all"       ⑧

    return word                            ⑨
```

① The pattern `'(.+)ing$'` matches one or more of anything followed by "ing." The `$` 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."

② The `re.match` starts matching at the beginning of the given `word` and is looking for either an upper- or lowercase "y" followed by "ou" and then the end of the string (`$`).

③ If `ing_word` is `None`, that means it failed to match. If it is not `None` (so it is "truthy"), that means it is an `re.Match` object we can use.

④ The `prefix` is the bit before the "ing" that we wrapped in parentheses. Because it is the first set of parens, we can fetch it with `ing_word.group(1)`.

⑤ 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 look at the beginning of `word`, which is not what we want.

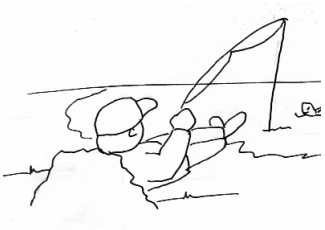
⑥ Return the prefix plus the string "in'" so as to drop the final "g."

⑦ If the `re.match` for the "you" pattern fails, then `you` will be `None`. If it is not `None`, then it matched and `you` is an `re.Match` object.

⑧ We used parens to capture the first character so as to maintain the case. That is, if the word was "You," then we want to return "Y'all." Here we return that first group plus the string "'all."

⑨ If the `word` matched neither a two-syllable "ing" patter or the word "you," then 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!



1.4. Review

- 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 the functions `re.match` to find pattern at the beginning of some text, `re.search` to find a pattern anywhere inside some text, and `re.split` to break text on a pattern.
- If you use capturing parens on the pattern for `re.split`, then the captured split pattern will be included in the returned values. This allows you to reconstruct the original string with the separators.

1.5. Going Further

- You could also replace "your" with "y'all's." For instance, "Where are your britches?" could become "Whare 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."
- 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!"

[1] 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.