Chapter 15: Rhymer: Using regular expressions to create rhyming words

Did you write your own input file for the "Gashlycrumb" exercise? When I was writing mine, I would come up with a word like maybe "cyanide" and wonder what I could rhyme with that. Mentally I start with the first consonant sound of the alphabet and start substituting "b" for "byanide," skip "c" because that's already the first character, then "d" for "dynanide," and so forth. This is effective but tedious, so I decided to write a program to do this for me, as one does.

This is basically another find-and-replace type of program that we've seen before like swapping all the numbers in a string in "Jump The Five" or all the vowels in a string in "Apples and Bananas." We wrote those programs using very manual, *imperative* methods like iterating through all the characters of a string, comparing them to some wanted value, and possibly returning a new value. In the solution for "Apples and Bananas," we briefly touched on "regular expressions" (AKA "regexes" [1]) which provide us a *declarative* way to describe a pattern of text. Now we're going to really dig into regexes to see what we can do!

In this exercise, we're going to take a given word and create "words" that rhyme. For instance, the word "bake" rhymes with words like "cake," "make," and "thrake," the last of which isn't actually a dictionary word but just a new string we create by replacing the "b" in "bake" with "thr." The algorithm we'll use is to split a word into any initial consonants and the rest of the word, so "bake" is split into "b" and "ake." We replace the "b" with all the other consonants from the alphabet plus these consonant clusters:

```
bl br ch cl cr dr fl fr gl gr pl pr sc sh sk sl sm sn sp st
sw th tr tw thw wh wr sch scr shr sph spl spr squ str thr
```

Be sure the output is sorted. For instance, these are the first three words our program will produce for "cake":

```
$ ./rhymer.py cake | head -3
bake
blake
brake
```

And the last three:

```
$ ./rhymer.py cake | tail -3
xake
yake
zake
```

We’ll replace any leading consonants with a list of other consonant sounds to create a total of 56 words:

```
$ ./rhymer.py cake | wc -l
      56
```

Note that we’ll replace *all* the leading consonants, not just the first one. For instance, with the word "chair" we need to replace "ch":

```
$ ./rhymer.py chair | tail -3
xair
yair
zair
```

If a word like "apple" does not start with a consonant, then we’ll append all the consonant sounds to the beginning to create words like "bapple" and "shrapple."

```
$ ./rhymer.py apple | head -3
bapple
blapple
brapple
```

Because there is no consonant to *replace*, words that start with a vowel will produce 57 rhyming words:

```
$ ./rhymer.py apple | wc -l
      57
```

To make this a bit easier, the output should always be all lowercase even if the input has uppercase letters:

```
$ ./rhymer.py GUITAR | tail -3
xuitar
yuitar
zuitar
```

If a word contains *nothing but consonants*, then print a message that the word cannot be rhymed:

```
$ ./rhymer.py RDNZL
Cannot rhyme "RDNZL"
```

The task of finding these initial consonants is made significantly easier with regexes.

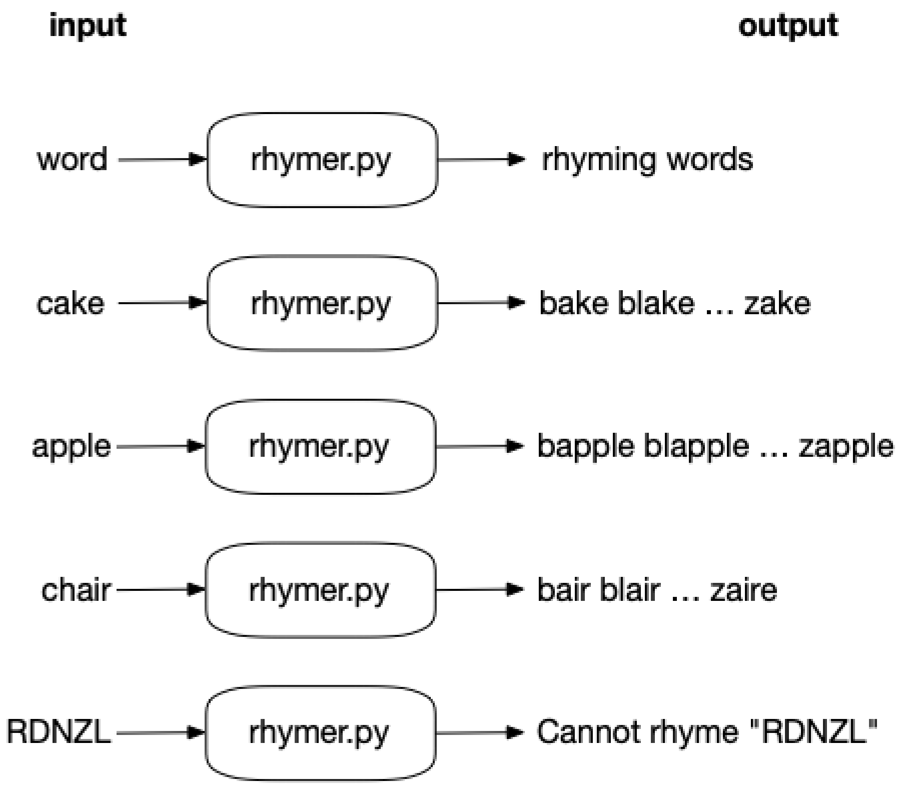
In this program, you will:

- Learn to write and use regular expressions
- Use a guard with a list comprehension
- Explore the similarities of list comprehension and guard to the `filter` function

- Entertain ideas of "truthiness" when evaluating Python types in a boolean context

Writing `rhymer.py`

The program takes a single, positional argument which is the string to rhyme. Here is a snazzy, jazzy, frazzy, thwazzy string diagram:



If given no arguments or the `-h` or `--help` flags, it should print a usage statement:

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

Make rhyming "words"

positional arguments:
  str         A word to rhyme

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

Breaking a word

To me, the main problem of the program is breaking the given word into the leading consonant sounds and the rest (something like the "stem") of the word. I wrote a function called called `stemmer`, and this is my test for it:

```
def test_stemmer():
    """ Test stemmer """
    assert stemmer('') == ('', '')            1
    assert stemmer('cake') == ('c', 'ake')    2
    assert stemmer('chair') == ('ch', 'air')  3
    assert stemmer('APPLE') == ('', 'apple')  4
    assert stemmer('RDNZL') == ('rdnzl', '')  5
```

The tests cover the following good and bad inputs:

1 The empty string.

2 A word with a single leading consonant.

3 A word with a leading consonant cluster.

4 A word with no initial consonants. Also an uppercase word, so checking that lowercase is returned.

5 A word with no vowels.

My `stemmer` function always returns a 2-tuple of the `(start, rest)` of the word, so the argument "cake" produces a tuple with two values of `('c', 'ake')`. The argument "chair" is split into the leading "ch" and the "air" strings. The argument "APPLE" has no `start` and only the `rest` of the word, which is lowercase. A string with no vowels like "RDNZL" returns the (lowercased) string as the `start` and nothing for the `rest`.

The first and last tests use "bad" values that can’t be rhymed. It’s up to me as the programmer to decide what to do with those. Should my `stemmer` function throw an exception or return `None` to indicate a failure? In this case, I decided to always return the same tuple and let the calling code deal with the problem. If there is nothing in the second position of the tuple, then there is nothing to rhyme. You don’t have to make the same decisions I made. If you prefer to throw an exception for unrhymable words, then change your test to handle an exception.

Using regular expressions

It’s certainly *possible* to write this program without regular expressions, but I hope you’ll see how radically different using regexes can be from manually writing your own search-and-replace code. To start off, we need to bring in the `re` module:

```
>>> import re
```

I would encourage you to then read `help(re)` to get a feel for all that you can do with regexes. They are a deep subject with books [2] and whole branches of academia devoted to them. We will only scratch the surface of what they can do.

We need to write a regex that will find consonants at the beginning of a string. We can define consonants as the characters of the English which are not the vowels, "a," "e," "i," "o," and "u." Our `stemmer` will only return lowercase letters, and so there are only 21 consonants we need to define. You could write them out, but I’d rather write a bit of code!

I can start with `string.ascii_lowercase`:

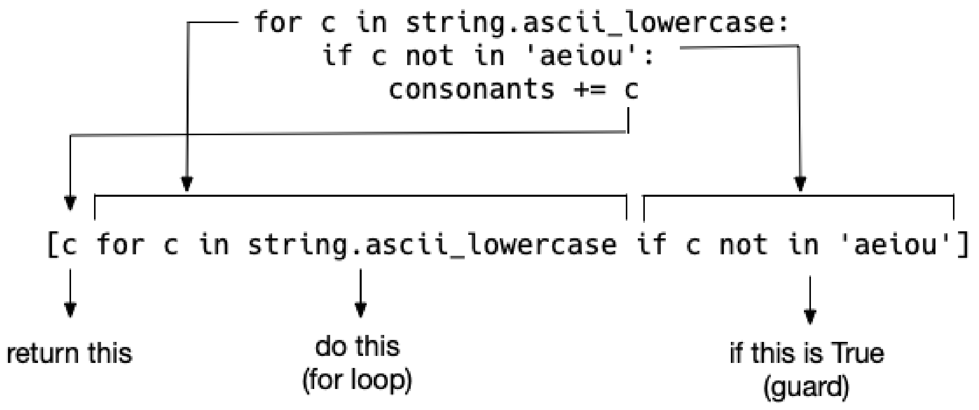
```
>>> import string
>>> string.ascii_lowercase
'abcdefghijklmnopqrstuvwxyz'
```

And use a list comprehension with a "guard" clause to filter out the vowels. As I want a `str` of consonants and not a `list`, I use `str.join` to make a new `str` value:

```
>>> consonants = ''.join([c for c in string.ascii_lowercase if c not in 'aeiou'])
>>> consonants
'bcdfghjklmnpqrstvwxyz'
```

The longer way to write this with a `for` loop and an `if` statement is this:

```
consonants = ''
for c in string.ascii_lowercase:
    if c not in 'aeiou':
        consonants += c
```


for c in string.ascii_lowercase:
if c not in 'aeiou':
consonants += c
[c for c in string.ascii_lowercase if c not in 'aeiou']
return this
do this
(for loop)
if this is True
(guard)

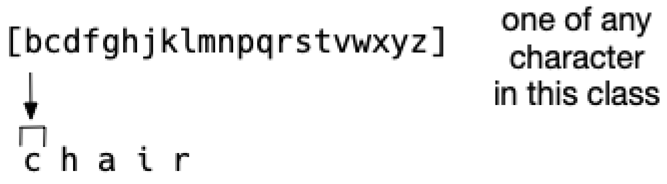
To use the `consonants` in a regex, I need to create a "character class" that includes all these values, which I can do by putting the characters inside square brackets:

```
>>> pattern = '[' + consonants + ']'
>>> pattern
'[bcdfghjklmnpqrstvwxyz]'
```

The `re` module has two search-like functions called `match` and `search`, and I always get them confused. They both look for a `pattern` (the first argument) in some `text`, but the `re.match` functions starts *from the beginning* of the `text` while the `re.search` function will match starting *anywhere* in the `text`.

As it happens, `re.match` is just fine because we are looking for consonants at the beginning of a string.

```
>>> text = 'chair'
>>> re.match(pattern, text)                          1
<re.Match object; span=(0, 1), match='c'>  2
```



1 Try to match the given `pattern` in the given `text`. If this succeeds, we get an `re.Match` object; otherwise, the value `None` is returned.

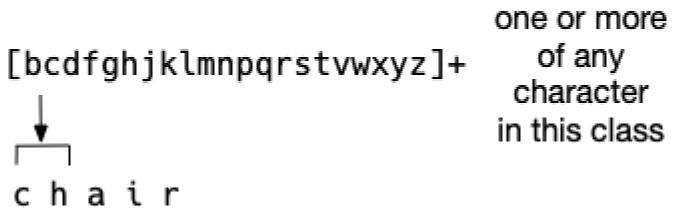
2 The match was successful, so we see a "stringified" version of the `re.Match` object.

The `match='c'` shows us that the regular expression found the string `'c'` at the beginning. Both the `re.match` and `re.search` functions will return an `re.Match` object on success. You can read `help(re.Match)` to learn more about all the cool things you can do with them:

```
>>> match = re.match(pattern, text)
>>> type(match)
<class 're.Match'>
```

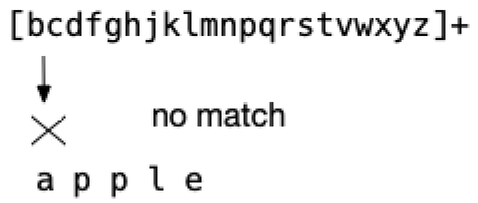
How do we get our regex to match the letters `'ch'` ? We can put a `'+'` sign after the character class to say we want *one or more*. (Does this sound a bit like `nargs='+'` to say one or more arguments?) I will use an f-string here to create the pattern:

```
>>> re.match(f'[{consonants}]+', 'chair')
<re.Match object; span=(0, 2), match='ch'>
```



What does it give us for a string with no leading consonants like "apple"?

```
>>>
re.match(f'[{conso
nants}]+',
'apple')
```



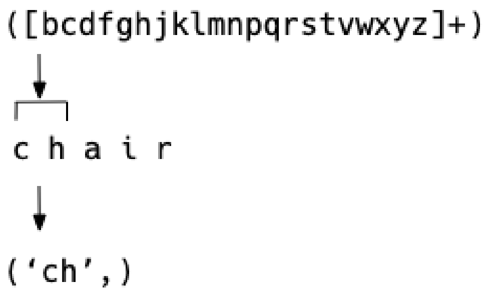
It appears like we got nothing back from that. What is the `type` of that return value?

```
>>> type(re.match(f'[{consonants}]+', 'apple'))
<class 'NoneType'>
```

Both the `re.match` and `re.search` functions return `None` to indicate a failure to match any text. We know that only some words will have a leading consonant sound, so this is not surprising. We'll see in a moment how to make this an optional match.

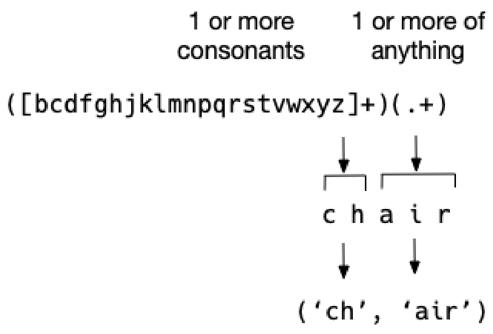
Using capture groups

It's all well and good to have found (or not) the leading consonants, but the goal here is to split the `text` into two parts: the consonants (if any) and the rest of the word. We can wrap parts of the regex in parentheses to create "capture groups." If the regex matches successfully, we can recover the parts using the `re.Match` object's `groups` method:



```
>>> match = re.match(f'([{consonants}]+)', 'chair')
>>> match.groups()
('ch',)
```

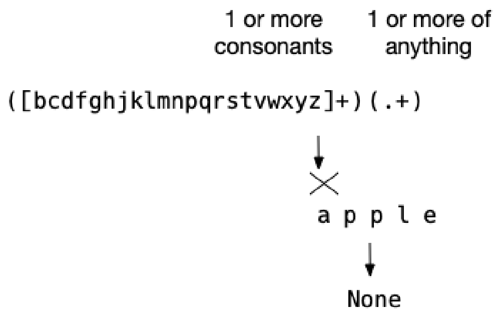
To capture everything that comes after the `consonants`, we can use the `.` to match anything and add `+` to mean one or more. We put that into parens to capture it:



```
>>> match = re.match(f'([{consonants}]+)(.+)', 'chair')
>>> match.groups()
('ch', 'air')
```

What happens when we try to use this on "apple"? It fails to make the first match on the consonants, and so *the whole match fails* and returns `None` :

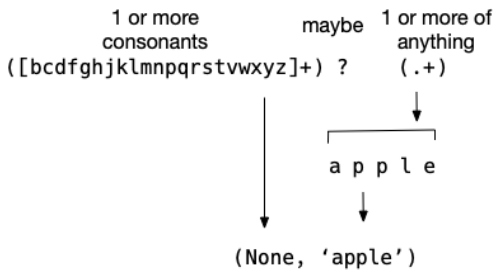
```
>>> match = re.match(f'([{consonants}]+)(.+)', 'apple')
>>> match.groups()
Traceback (most recent call last):
  File "<stdin>", line 1, in <module>
AttributeError: 'NoneType' object has no attribute 'groups'
```



Remember that `re.match` returns `None` when it fails to find the pattern. We can add `?` at the end of the `consonants` pattern to make it optional:

```
>>> match = re.match(f'([{consonants}]+)?(.+)', 'apple')
>>> match.groups()
(None, 'apple')
```

The `match.groups` function returns a `tuple` containing the matches for each grouping created by the parentheses. You can also use the `match.group` (singular) with a group number to get a specific group. Note that these start numbering from 1:



```
>>> match.group(1)  1
>>> match.group(2)  2
'apple'
```

1 There was no match for the first group on "apple," so this is a `None`.

2 The second group captured the entire word.

If you match on the "chair," there are values for both groups:

```
>>> match = re.match(f'([{consonants}]+)?(.+)', 'chair')
>>> match.group(1)
'ch'
>>> match.group(2)
'air'
```

So far we’ve only dealt with lowercase text because our program will always emit lowercase values. Still, let’s explore what happens when we try to match uppercase text:

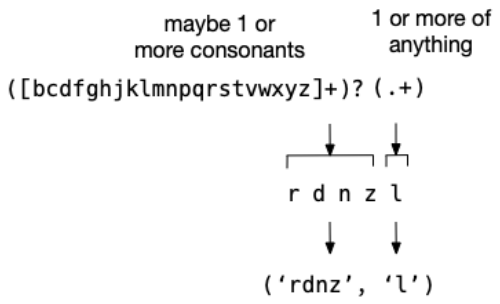
```
>>> match = re.match(f'([{consonants}]+)?(.+)', 'CHAIR')
>>> match.groups()
(None, 'CHAIR')
```

Not surprisingly, that fails. Our pattern only defines lowercase characters. We could add all the uppercase consonants, but it’s a bit easier to use a third optional argument to `re.match` to tell indicate case-insensitive searching:

```
>>> match = re.match(f'([{consonants}]+)?(.+)', 'CHAIR', re.IGNORECASE)
>>> match.groups()
('CH', 'AIR')
```

What do you get when you search on text that has nothing but consonants?

```
>>> match = re.match(f'([{consonants}]+)?(.+)', 'rdnzl')
>>> match.groups()
('rdnz', 'l')
```

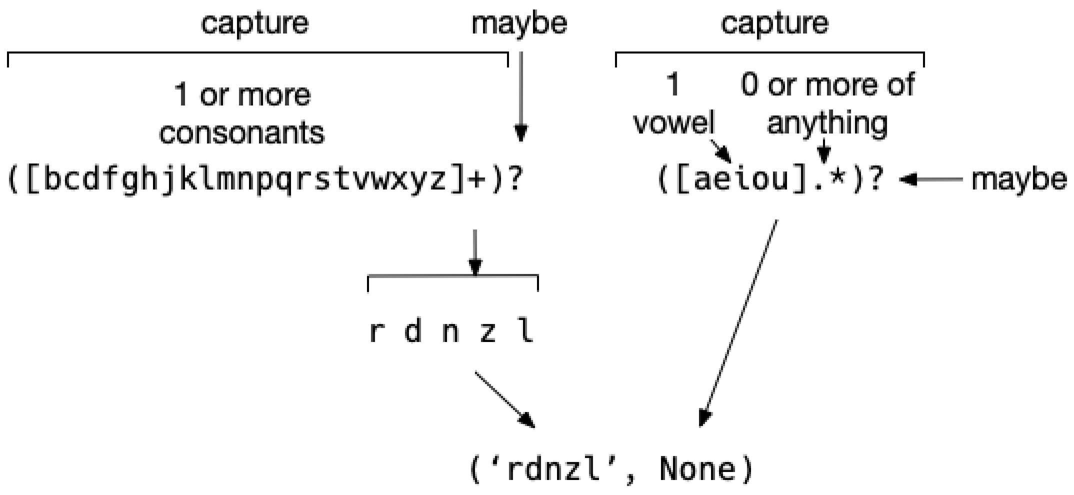


Were you expecting the first group to include *all* the consonants and the second group to have nothing? It might seem a bit odd that it decided to split off the "l" into the last group, but we have to think *extremely literally* about how the regex engine is working. We described an optional group of one or more consonants that *must be followed* by one or more of anything else. The "l" counts as one or more of anything else, so the regex matched exactly what we requested. If we change the `(.+)` to `(.*)` to make it *zero or more*, then it works as expected:

```
>>> match = re.match(f'([{consonants}]+)?(.*)', 'rdnzl')
>>> match.groups()
('rdnzl', '')
```

Our regex is not quite complete. We want to indicate that there should be a vowel after the consonants, and then anything else. We can use another character class to describe the vowels, followed by *zero or more* (`*`) of anything (`.`). The whole second capture group is optional, so we add `?` :

```
>>> match = re.match(f'([{consonants}]+)?([aeiou].*)?', 'rdnzl')
>>> match.groups()
('rdnzl', None)
```



Truthiness

The last thing is to return the correct values from the `stemmer` function. I decided that I would always return a 2-tuple of `(start, rest)`, and that I would always use the empty string to denote a missing value rather than a `None` :

```
return (match.group(1) or '', match.group(2) or '') if match else ('', '')
```

I'm using the `or` operator to decide between something on the left *or* something on the right. The `or` will return the first "truthy" value, the one that sort of-kind of evaluates to `True` in a boolean context:

```
>>> True or False      1
True
>>> False or True      2
True
>>> 1 or 0             3
1
>>> 0 or 1             4
1
>>> 0.0 or 1.0         5
1.0
>>> '0' or ''          6
'0'
>>> 0 or False         7
False
>>> [] or ['foo']      8
['foo']
>>> {} or dict(foo=1)  9
{'foo': 1}
```

1 It’s easiest to see with literal `True` and `False` values.

2 No matter the order, the `True` value will be taken.

3 In a boolean context, the `int` value `0` is "falsey" and any other value is "truthy."

4 The number values behave exactly like actual boolean values.

5 `float` values also behave like `int` values.

6 With `str` values, the empty string is the "falsey" and so anything else is "truthy." It may look odd because it returns `'0'` but that’s not the *numeric* value zero but the *string* we use to represent the value of zero. Wow, such philosophical.

7 If no value is "truthy," then the last value is returned.

8 The empty `list` is "falsey," and so any non-empty `list` is "truthy."

9 The same is true of dictionaries. The empty `dict` is "falsey," and any non-empty `dict` is "truthy."

If you use the above regex and `return` in your `stemmer`, it should pass the `test_stemmer` test.

Creating the output

Let’s review what the program should do:

- Take a positional string argument.
- Try to split it into two parts: any leading consonants and the rest of the word.
- If the split is successful, combine the "rest" of the word (which might actually be the entire word if there are no leading consonants) with all the other consonant sounds listed above. Be sure to *not* include the original consonant sound and to sort the rhyming strings.
- If you are unable to split the word, then print the message `Cannot rhyme "<word>"`.

OK, time to step off onto the dance floor. Do your best to write this yourself before looking at the solution.

Solution

```
#!/usr/bin/env python3
"""Make rhyming words"""

import argparse
import re       1
import string


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

    parser = argparse.ArgumentParser(
        description='Make rhyming "words"',
        formatter_class=argparse.ArgumentDefaultsHelpFormatter)

    parser.add_argument('word', metavar='str', help='A word to rhyme')

    return parser.parse_args()


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

    args = get_args()                                    2
    prefixes = list('bcdfghjklmnpqrstvwxyz') + (         3
        'bl br ch cl cr dr fl fr gl gr pl pr sc '
        'sh sk sl sm sn sp st sw th tr tw thw wh wr '
        'sch scr shr sph spl spr squ str thr').split()

    start, rest = stemmer(args.word)                     4
    if rest:                                             5
        print('\n'.join(sorted([p + rest for p in prefixes if p != start])))  6
    else:
        print(f'Cannot rhyme "{args.word}"')             7


# --------------------------------------------------
def stemmer(word):
    """Return leading consonants (if any), and 'stem' of word"""

    vowels = 'aeiou'                                     8
    consonants = ''.join(                                9
        [c for c in string.ascii_lowercase if c not in vowels])
    pattern = (                                          10
        '([' + consonants + ']+)?'  # capture one or more consonants, optional
        '('                         # start capture
        '[' + vowels + ']'          # at least one vowel
        '.*'                        # zero or more of anything else
        ')?')                       # end capture, optional group
    match = re.match(pattern, word.lower())              11
    return (match.group(1) or '', match.group(2) or '') if match else ('', '')  12


# --------------------------------------------------
def test_stemmer():  13
    """test the stemmer"""

    assert stemmer('') == ('', '')
    assert stemmer('cake') == ('c', 'ake')
    assert stemmer('chair') == ('ch', 'air')
    assert stemmer('APPLE') == ('', 'apple')
    assert stemmer('RDNZL') == ('rdnzl', '')


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

1 The `re` module is for regular expressions.

2 Get the command-line arguments.

3 Define all the prefixes that will be added to create rhyming words.

4 Split the `word` argument into two possible parts. Because the `stemmer` function always returns a 2-tuple, we can unpack the values into two separate values.

5 Check if there is a part of the word that we can use to create rhyming strings.

6 If there is, use a list comprehension to iterate through all the prefixes and add them to the stem of the word. Use a guard to ensure that any given prefix is not the same as the beginning of the word. Sort all the values and print them, joined on newlines.

7 If there is nothing we can use to create rhymes, let the user know.

8 Since I will use the `vowels` more than once, I assign them to a variable.

9 The `consonants` are the letters that are not `vowels`. I will only match to lowercase letters.

10 The `pattern` is defined using consecutive literal strings that Python will join together into one string. By breaking up the pieces onto separate lines, I can comment on each part of the regular expression.

11 Use the `re.match` function to start matching *at the beginning* of the the given `word` which I convert to lowercase.

12 Even though I would expect this pattern to never fail, I conservatively use an `if` expression to ensure that I have a value for the `match`. If I do, I return a 2-tuple consisting of the two capture groups, but I ensure that neither group is a `None` but rather an empty string to denote a missing value. If there is no `match`, I return a tuple of two empty strings.

13 The afore-described test function for the `stemmer` function.

Discussion

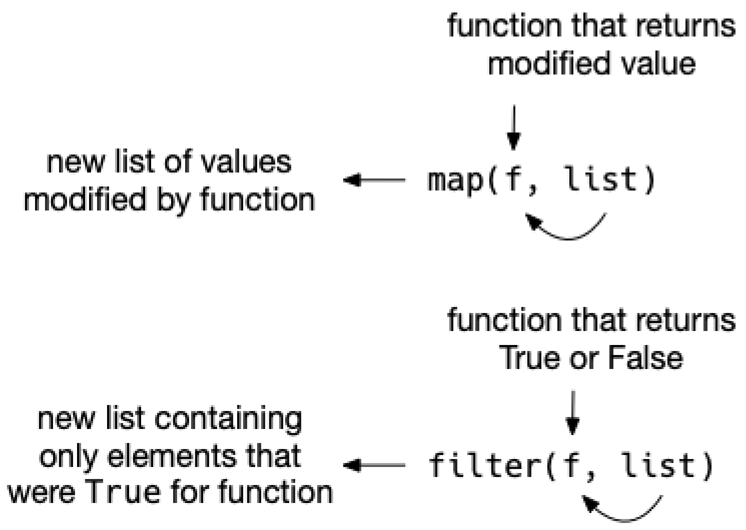
There are many ways you could have written this, but, as always, I wanted to break the problem down into some units I could write and test. For me, this came down to splitting the word into a possible leading consonant sound and the rest of the word. If I can manage that, I can create rhyming strings; if I cannot, then I need to alert the user.

Stemming a word

For the purposes of this program, the "stem" of a word is the part after any initial consonants which I define using a list comprehension with a guard to take only the letters that are not vowels:

```
>>> vowels = 'aeiou'
>>> consonants = ''.join([c for c in string.ascii_lowercase if c not in vowels])
```

I showed how this is a more concise way to write a `for` loop with an `if` statement. We've looked at `map` several times now and talked about how it is a *higher-order function* (HOF) because it takes *another function* as the first argument, applying it to all the members of a sequence to produce a new, transformed sequence (like painting cars blue). Here I'd like to introduce another HOF called `filter` which takes a function and some *iterable* (something that can be *iterated* like a `list`). As with `map`, the function is applied to all the elements of the sequence. The return value of function will be evaluated by its "truthiness." Only those elements that evaluate as "truthy" will be returned by `filter`.


function that returns modified value
new list of values modified by function
map(f, list)
function that returns True or False
new list containing only elements that were True for function
filter(f, list)

Here is another way to write the idea of the list comprehension using a `filter`:

```
>>> consonants = ''.join(filter(lambda c: c not in
vowels, string.ascii_lowercase))
```

Just as with `map`, I use the `lambda` keyword to create an *anonymous function*. The `c` is the name of the variable that will hold the argument which, in this case, will be each character from `string.ascii_lowercase`. The entire body of the function is the evaluation `c not in vowels`. Each of the vowels will return `False` for this:

```
>>> 'a' not in vowels
False
```

And each of the consonants will return `True`:

```
>>> 'b' not in vowels
True
```

Therefore only the consonants will be allowed to pass through the `filter`. To think back to our "blue" cars, let's write a `filter` that only accepts cars that start with the string "blue ":

```
>>> list(filter(lambda car: car.startswith('blue '), ['blue Honda', 'red Chevy', 'blue Ford']))
['blue Honda', 'blue Ford']
```

When the `car` variable has the value "red Chevy," the `lambda` returns `False`, and so that value is rejected:

```
>>> car = 'red Chevy'
>>> car.startswith('blue ')
False
```

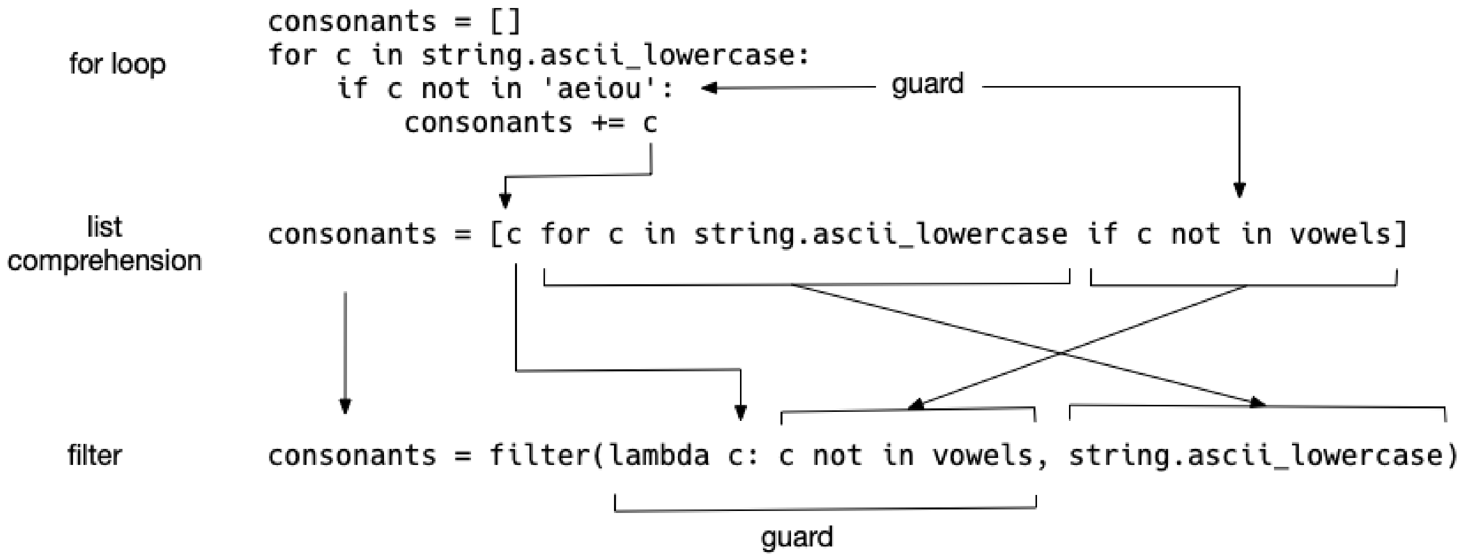
Note that if none of the elements from the original iterable are accepted, then `filter` will produce an empty `list` (`[]`). For example, I could `filter` for numbers greater than 10. Note that `filter` is another *lazy* function that I must coerce using the `list` function in the REPL:

```
>>> list(filter(lambda n: n > 10, range(0, 5)))
[]
```

A list comprehension would also return an empty list:

```
>>> [n for n in range(0, 5) if n > 10]
[]
```

Here is a diagram showing the relationship of creating a new `list` called `consonants` using an imperative, `for`-loop approach, an idiomatic list comprehension with a guard, and a purely functional approach using `filter`. All of these are perfectly acceptable, though the most Pythonic way is probably the list comprehension. The `for` loop would be very familiar to a C or Java programmer, while the `filter` would be immediately recognizable to the Haskeller or even someone from a Lisp-like language. The `filter` might be slower than the list comprehension, especially if the iterable were large. Choose whichever way makes more sense for your style and application.



```
consonants = []
for c in string.ascii_lowercase:
    if c not in 'aeiou':
        consonants += c

consonants = [c for c in string.ascii_lowercase if c not in vowels]

consonants = filter(lambda c: c not in vowels, string.ascii_lowercase)
```

Figure 1. The `filter` *function is similar to a list comprehension with a guard.*

Writing a regular expression

We talked in the introduction about the individual parts of the regular expression I ended up using. I'd like to take a moment to mention the way I formatted the regex in the code using an interesting trick of the Python interpreter that will stitch together string literals into one long string if they are enclosed in a grouping like parentheses:

```
>>> this_is_just_to_say = ('I have eaten '
... 'the plums '
... 'that were in '
... 'the icebox')
>>> this_is_just_to_say
'I have eaten the plums that were in the icebox'
```

Note that there are no commas after each string as that would create a `tuple` with 4 individual strings:

```
>>> this_is_just_to_say = ('I have eaten ',
... 'the plums ',
... 'that were in ',
... 'the icebox')
>>> this_is_just_to_say
('I have eaten ', 'the plums ', 'that were in ', 'the icebox')
```

The advantage of writing out the regular expression like this is to add comments to each important part:

```
pattern = (
    '([' + consonants + ']+)?'  # capture one or more consonants, optional
    '('                         # start capture
    '[' + vowels + ']'          # at least one vowel
    '.*'                        # zero or more of anything else
    ')?')                       # end capture, optional group
```

I could have written the entire regex on one line. Ask yourself which version would you rather read and maintain:[3]

```
pattern = '([' + consonants + ']+)?([' + vowels + '].*)?'
```

I really worked hard on creating the regular expression, and I have a good bit of confidence that this regex will never fail to match something, even the empty string, because both elements (the leading consonant cluster and the rest of the word) are optional. Still, I will very conservatively call the `re.match` so I can check the return value which I called `match`. Note that I lowercase the `word`:

```
match = re.match(pattern, word.lower())
```

It's possible my function might receive some value which might cause the `re.match` to return a `None`. I use the `match` value in an `if` expression to be absolutely sure that I can safely call `match.group`. As stated in the introduction, I would prefer to use the empty string rather than `None` to indicate a missing group, so I use the `or` operator to choose the first "truthy" value for both groups. If there is no `match`, I return a 2-tuple of empty strings:

```
return (match.group(1) or '', match.group(2) or '') if match else ('', '')
```

Testing and using the `stemmer` function

I first introduced the `test_stemmer` so we could think about what values we might pass in, both good and bad, and what we might expect to receive:

```
def test_stemmer():
    """test the stemmer"""
    assert stemmer('') == ('', '')
    assert stemmer('cake') == ('c', 'ake')
    assert stemmer('chair') == ('ch', 'air')
    assert stemmer('APPLE') == ('', 'apple')
    assert stemmer('RDNZL') == ('rdnzl', '')
```

I really recommend that you always write a `test_x` function *before* you write the `x` function as it can help you think of the function as this black box. Something goes in, something comes out, and what happens inside is a separate concern.

One of the very interesting things about Python code is that your `rhymer.py` program is also — kind of, sort of — a *module* of code. That is, you haven't explicitly written it to be this container of reusable functions, but it is. You can even run the functions from inside the REPL. For this to work, be sure you run `python3` inside the same directory as the `rhymer.py` code:

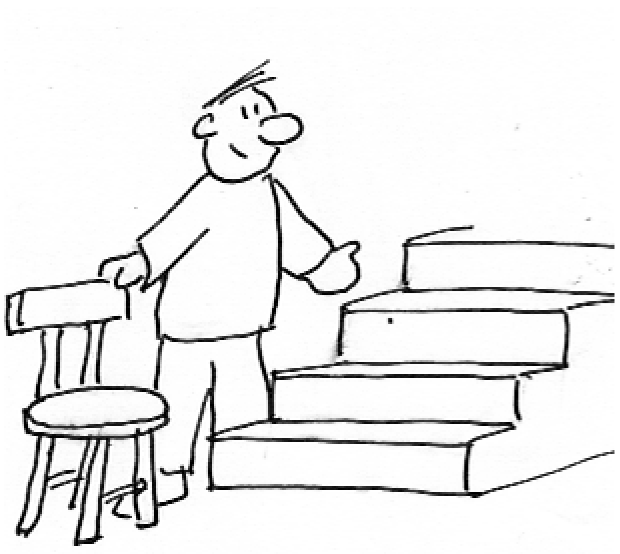
```
>>> from rhymer import stemmer
```

And now you can run and test your `stemmer` function manually:

```
>>> stemmer('apple')
('', 'apple')
>>> stemmer('banana')
('b', 'anana')
```

If you don’t explicitly `import` a function, then you can use the fully qualified function name by adding the module name to the front:

```
>>> import rhymer
>>> rhymer.stemmer('apple')
('', 'apple')
>>> rhymer.stemmer('banana')
('b', 'anana')
```



There are many advantages to writing many small functions rather than long, sprawling programs. One is that small functions are much easier to write, understand, and test. Another is that you can put your tidy, tested functions into modules and share them across different programs you write. As you write more and more programs, you will find yourself solving some of the same problems repeatedly. It’s far better to create modules with reusable code rather than copying pieces from one program to another. If you ever find a bug in a shared function, you can fix it once and all the programs sharing the function get the fix. The other way is to find the duplicated code in every program and change it (and hoping that this doesn’t introduce even more problems because the code is entangled with other code!).

Creating rhyming strings

The `stemmer` function should always return a 2-tuple of the `(start, rest)` of a given word. As such, I can unpack the two values into separate variables:

```
>>> start, rest = stemmer('cat')
>>> start
'c'
>>> rest
'at'
```

If there is a value for `rest`, I can add all my `prefixes` to the beginning:

```
>>> prefixes = list('bcdfghjklmnpqrstvwxyz') + (
...     'bl br ch cl cr dr fl fr gl gr pl pr sc '
...     'sh sk sl sm sn sp st sw th tr tw wh wr'
...     'sch scr shr sph spl spr squ str thr').split()
```

I decided to use another list comprehension with a guard to skip any prefix that is the same as the `start` of the word. The result will be a new `list` which I pass to the `sorted` function to get the correctly ordered strings:

```
>>> sorted([p + rest for p in prefixes if p != start])
['bat', 'blat', 'brat', 'chat', 'clat', 'crat', 'dat', 'drat', 'fat', 'flat', 'frat',
 'gat', 'glat', 'grat', 'hat', 'jat', 'kat', 'lat', 'mat', 'nat', 'pat', 'plat', 'prat',
 'qat', 'rat', 'sat', 'scat', 'schat', 'scrat', 'shat', 'shrat', 'skat', 'slat', 'smat',
 'snat', 'spat', 'sphat', 'splat', 'sprat', 'squat', 'stat', 'strat', 'swat', 'tat', 'that',
 'thrat', 'thwat', 'trat', 'twat', 'vat', 'wat', 'what', 'wrat', 'xat', 'yat', 'zat']
```

I then `print` that `list`, joined on newlines. If there is no `rest` of the given word, I `print` a message that the word cannot be rhymed:

```
if rest:
    print('\n'.join(sorted([p + rest for p in prefixes if p != start])))
else:
    print(f'Cannot rhyme "{args.word}"')
```

Writing `stemmer` without regular expressions

It is certainly possible to write a solution that does not use regular expressions. My idea was to find the first position of a vowel in the given string. If one is present, use a list slice to return the portion of the string up to that position and the portion starting at that position:

```
def stemmer(word):
    """Return leading consonants (if any), and 'stem' of word"""
    word = word.lower()           1
    pos = list(                   2
        filter(lambda v: v >= 0,  3
               map(lambda c: word.index(c) if c in word else -1, 'aeiou')))  4
    if pos:                       5
        first = min(pos)          6
        return (word[:first], word[first:])  7
    else:
        return (word, '')         8
```

1 Lowercase the given word to avoid dealing with uppercase letters.

2 Coerce the lazy `filter` to a `list`.

3 Only allow values greater than 0. The value `-1` indicates "not found" as 0 is a valid index for a string.

4 `map` over the vowels `'aeiou'` to get their index if they are present in the given word. Use `-1` to indicate the vowel is not present.

5 The `pos` variable is a `list`. In a boolean context, the empty list evaluates to "falsey" and anything else is "truthy."

6 The `first` position of a vowel will be the *minimum* from the positions (`min(pos)`).

7 Use the `first` to slice the portion of `word` up to (but not including) `first` and then the portion from `first` to the end.

8 If `pos` is the empty `list` (meaning no vowels were present), then return a 2-tuple of the `word` and the empty string to indicate there is no `rest` of the word to use for rhyming.

This function will also pass the `test_stemmer` function. By writing a test just for the idea of this one function and exercising it with all the different values I would expect, I'm free to *refactor* my code. As stated before, the `stemmer` is a black box. What goes on inside the function is of no concern to the code that calls it. As long as the function passes the tests, then it is "correct" (for certain values of "correct").

Small functions and their *tests* will set you free to improve your programs! First make something work, and make it beautiful. Then try to make it better, using your tests to ensure it keeps working as expected.

Review

- Regular expressions allow us to declare a pattern that we wish to find. The regex *engine* will sort out whether the pattern is found or not. This is a *declarative* approach to program rather than the *imperative* method of manually seeking out patterns by writing code ourselves.
- We can wrap parts of the pattern in parentheses to "capture" them into groups that we can fetch from the result of `re.match` or `re.search`.
- You can add a guard to a list comprehension to avoid taking some elements from an iterable.
- The `filter` function is another way to write a list comprehension with a guard. Like `map`, it is a lazy, higher-order function that takes a function that will be applied to every element of an iterable. Only those elements which are "truthy" are returned.
- Python can evaluate many types including strings, numbers, lists, and dictionaries in a *boolean context* to arrive at a sense of "truthiness." That is, you are not restricted to just `True` and `False` in `if` expressions. The empty string `''`, the `int 0`, the `float 0.0`, the empty `list []`, and the empty `dict {}` are all considered "falsey," and so any value from those types like the non-empty `str`, `list` or `dict`, or any numeric value not zero-ish is considered "truthy."
- You can break long string literals into shorter strings in your code and then group them with parentheses to have Python join them into on long string. It’s advisable to break long regexes into shorter strings and add comments on each line to document the function of each pattern.
- Write small functions *and tests* and share them in modules. Every `.py` file can be a module from which you can `import` functions. Sharing small, tested functions is better than writing long programs and copying/pasting code as needed.

Going Further

- Add an `--output` option to write the words to a given file. The default should be to write to `STDOUT`.
- Read an input file and create rhyming words for all the words in the file. You can borrow from the "Words Count" program to read a file and break it into words, then iterate each word and create an output file for each word with the rhyming words.
- Write a new program that find all unique consonant sounds in a dictionary of English words like the file `/usr/share/dict/words` that usually exists on most Unix systems. Print them out in alphabetical order and use those to expand this program’s consonants.
- Write a program to create Pig Latin where you move the initial consonant sound from the beginning of the word to the end and add "-ay" so that "cat" becomes "at-cay." If a word starts with a vowel, then add "-yay" to the end so that "apple" becomes "apple-yay."

- Write a program to create spoonerisms where the initial consonant sounds of adjacent words are switched so you get "blushing crow" instead of "crushing blow."

---

1. Pronounced with a hard "g" like in "George"
2. *Mastering Regular Expressions* by Jeffrey Friedl is one I would recommend
3. "Looking at code you wrote more than two weeks ago is like looking at code you are seeing for the first time." - Dan Hurvitz

Last updated 2020-01-14 20:38:07 -0700