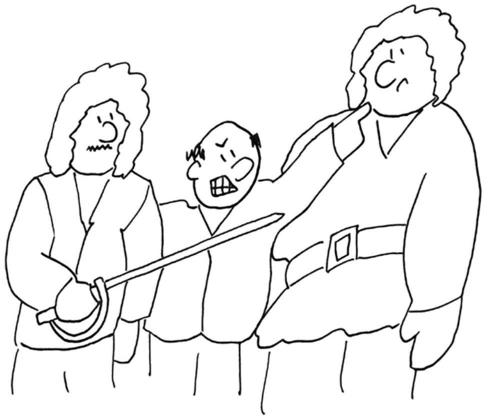
14 Rhymer: Using regular expressions to create rhyming words



In the movie *The Princess Bride*, the characters Inigo and Fezzik have a rhyming game they like to play, especially when their cruel boss, Vizzini, yells at them:

Inigo: That Vizzini, he can fuss.
Fezzik: I think he likes to scream at us.
Inigo: Probably he means no harm.
Fezzik: He's really very short on charm.

When I was writing the alternate.txt for chapter 7, I would come up with a word like "cyanide" and wonder what I could rhyme with that. Mentally I started with the first consonant sound of the alphabet and substituted "b" for "byanide," skipped "c" because that's already the first character, then "d" for "dyanide," 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, like swapping all the numbers in a string in chapter 4 or all the vowels in a string in chapter 8. 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 final solution for chapter 8, we briefly touched on "regular expressions" (also called "regexes"—pronounced with a soft "g" like in "George"), which give us a *declarative* way to describe patterns of text. The material here may seem a bit of a reach, but I really want to help you dig into regexes to see what they can do.

In this chapter, 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 I created by replacing the "b" in "bake" with "thr."

The algorithm we'll use will split a word into any initial consonants and the rest of the word, so "bake" is split into "b" and "ake." We'll 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
```

These are the first three words our program will produce for "cake":

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

And these are the last three:

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

Make sure your output is sorted alphabetically as this is important for the tests.

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, 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:

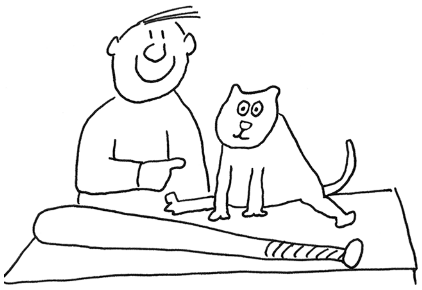
```
$ ./rhymer.py apple | wc -l
      57
```

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, we'll print a message stating that the word cannot be rhymed:

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



The task of finding the 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 with a guard to the `filter()` function
- Entertain ideas of "truthiness" when evaluating Python types in a Boolean context

14.1 Writing rhymer.py

The program takes a single, positional argument, which is the string to rhyme. Figure 14.1 shows 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] word

Make rhyming "words"

positional arguments:
  word        A word to rhyme

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

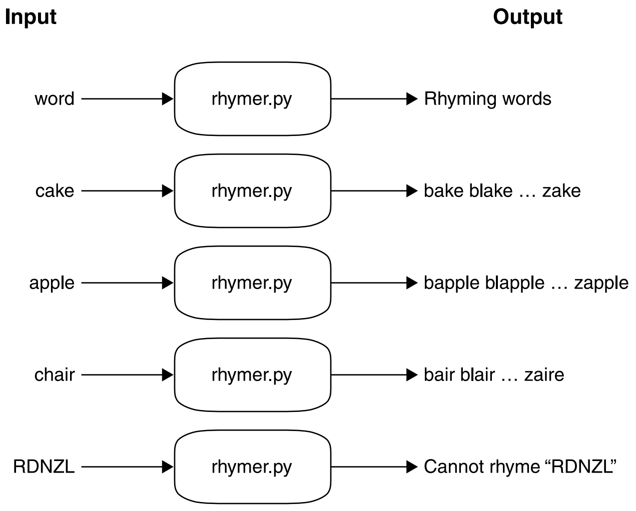


Figure 14.1 The input for our rhymer program should be a word, and the output will be a list of rhyming words or an error.

14.1.1 Breaking a word

To my mind, 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.

To start out, we can define a placeholder for a function I call `stemmer()` that does nothing right now:

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

The pass statement will do nothing at all. Since the function does not return a value, Python will return None by default.

Then we can define a `test_stemmer()` function to help us think about the values we might give the function and what we expect it to return. We want a test with good values like "cake" and "apple" that can be rhymed as well as values like the empty string or a number, which cannot:

```
def test_stemmer():
    """ Test stemmer """
    assert stemmer('') == ('', '')                ❶
❷   assert stemmer('cake') == ('c', 'ake')
    assert stemmer('chair') == ('ch', 'air')      ❸
❹   assert stemmer('APPLE') == ('', 'apple')
    assert stemmer('RDNZL') == ('rdnzl', '')      ❺
❻   assert stemmer('123') == ('123', '')
```

The tests cover the following good and bad inputs:

❶ The empty string
❷ 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 this checks that lowercase is returned
5. A word with no vowels
6. Something that isn't a word at all

I decided that my `stemmer()` function will always returns a 2-tuple of the `(start, rest)` of the word. (You can write a function that does something different, but be sure to change the test to match.) It's the second part of that `tuple`—the `rest`—that we can use to create rhyming words. For instance, the word "cake" produces a `tuple` with `('c', 'ake')`, and "chair" is split into `('ch', 'air')`. The argument "`APPLE`" has no `start` and only the `rest` of the word, which is lowercase.

When I'm writing tests, I usually try to provide both good and bad data to my functions and programs. Three of the test values cannot be rhymed: the empty string (`''`), a string with no vowels (`'RDNZL'`), and a string with no letters (`'123'`). The `stemmer()` function will still return a `tuple` containing the lowercased word in the first position of the tuples and the empty string in the second position for the `rest` of the word. It is up to the calling code to deal with a word that has no part that can be used to rhyme.

14.1.2 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 encourage you to read `help(re)` to get a feel for all that you can do with regexes. They are a deep subject with many books and whole branches of academia devoted to them (*Mastering Regular Expressions* by Jeffrey Friedl (O'Reilly, 2006) is one book I would recommend). There are many helpful websites that can further explain regexes, and some can help you write them (such as https://regexr.com/). We will only scratch the surface of what you can do with regexes.

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

We can start with `string.ascii_lowercase`:

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

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

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

The longer way to write this with a `for` loop and an `if` statement is as follows (see figure 14.2):

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

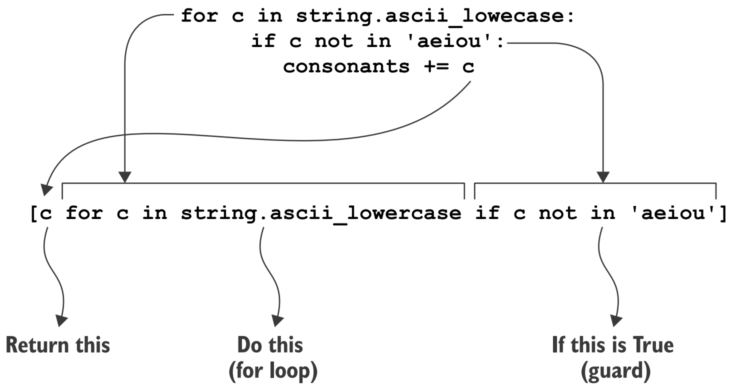


Figure 14.2 The `for` loop (top) can be written as a list comprehension (bottom). This list comprehension includes a guard so that only consonants are selected, which is like the `if` statement at the top.

In chapter 8 we created a "character class" for matching the vowels by listing them in square brackets, like `'[aeiou]'`. We can do the same here with our `consonants`, like so:

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

The `re` module has two search-like functions called `re.match()` and `re.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`, whereas 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 (see figure 14.3).

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

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

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

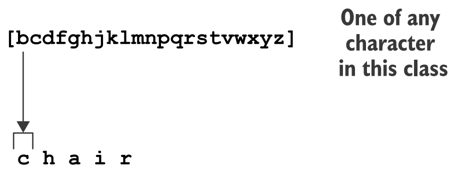


Figure 14.3 The character class of consonants will match the "c" at the beginning of "chair."

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 a `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* (see figure 14.4). (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'>
```

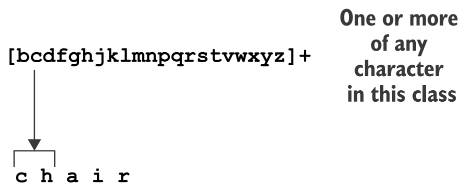


Figure 14.4 Adding a plus sign to the class will match one or more characters.

What does it give us for a string with no leading consonants like "apple," as in figure 14.5?

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

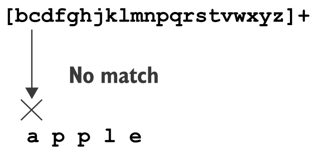


Figure 14.5 This regex fails to match a word that does not start with a consonant.

It seems 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.

14.1.3 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.groups()` method (see figure 14.6):

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

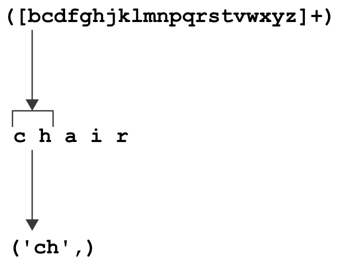


Figure 14.6 Adding parentheses around a pattern causes the matching text to be available as a capture group.

To capture everything that comes after the `consonants`, we can use a period (`.`) to match anything, and add a plus sign (`+`) to mean one or more. We can put that into parentheses to capture it (see figure 14.7):

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

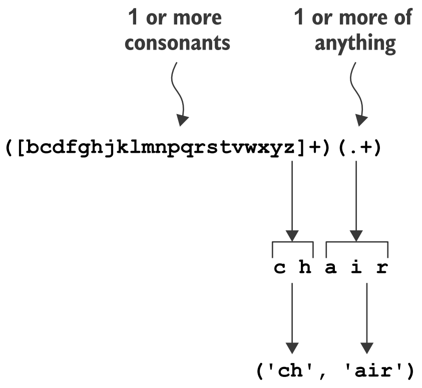


Figure 14.7 We define two capture groups to access the leading consonant sound and whatever follows.

What happens when we try to use this on “apple”? It fails to make the first match on the consonants, so *the whole match fails* and returns `None` (see figure 14.8):

```
>>> 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'
```

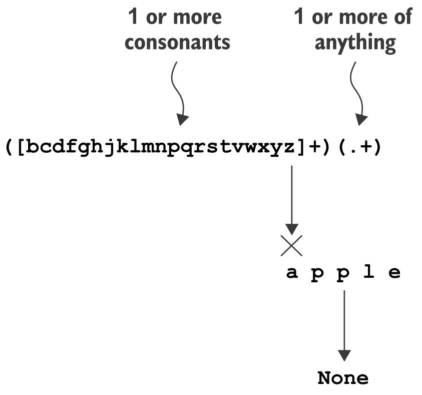


Figure 14.8 The pattern still fails when the text starts with a vowel.

Remember that `re.match()` returns `None` when it fails to find the pattern. We can add a question mark (`?`) at the end of the `consonants` pattern to make it optional (see figure 14.9):

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

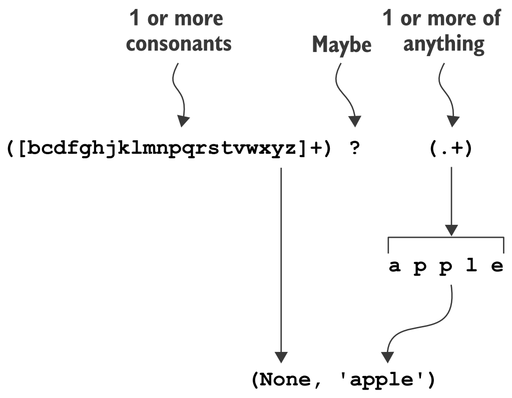


Figure 14.9 A question mark after a pattern makes it optional.

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

```
>>> match.group(1)     <-- There was no match for the first group on "apple," so this is a None.
>>> match.group(2)     <-- The second group captured the entire word.
'apple'
```

If you match on "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 specify case-insensitive searching:

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

Or you can force the text you are searching to lowercase:

```
>>> match = re.match(f'([{consonants}]+)?(.+)', 'CHAIR'.lower())
>>> 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, as shown in figure 14.10, 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.

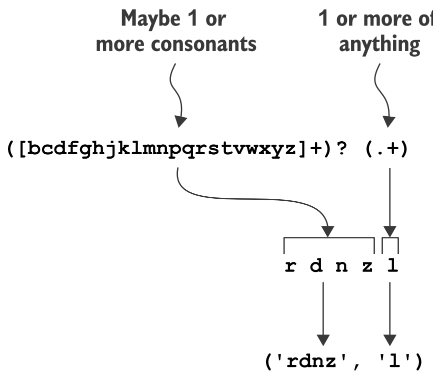


Figure 14.10 The regex does exactly what we ask, but perhaps not what we wanted.

If we change the `(.+)` to `(.*)` to make it *zero or more*, it works as expected:

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

Our regex is not quite complete, as it doesn't handle matching on something like `123`. That is, it matches too well because the period `(.)` will match the digits, which we don't want:

```
>>> re.match(f'([{consonants}]+)?(.*)?', '123')
<re.Match object; span=(0, 3), match='123'>
```

We need to indicate that there should be *at least one vowel* after the consonants, which may be followed by anything else. We can use another character class to describe any vowel. Since we need to capture this, we'll put it in parentheses, so `([aeiou])`. That may be followed by *zero or more* of anything, which also needs to be captured, so `(.*)`, as shown in figure 14.11.

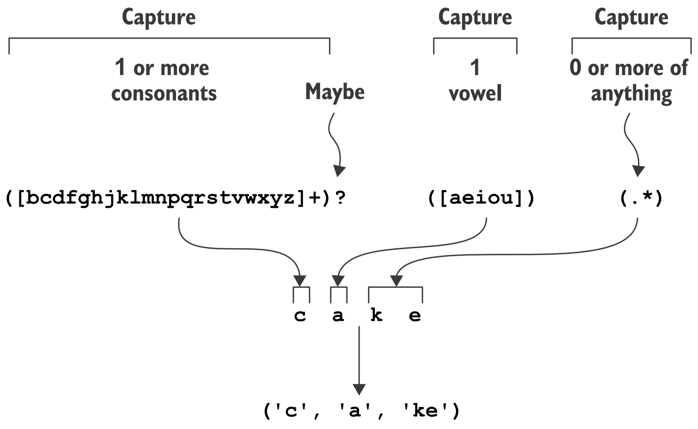


Figure 14.11 The regex now requires the presence of a vowel.

Let's go back and try this on values we expect to work:

```
>>> re.match(f'([{consonants}]+)?([aeiou])(.*)', 'cake').groups()
('c', 'a', 'ke')
>>> re.match(f'([{consonants}]+)?([aeiou])(.*)', 'chair').groups()
('ch', 'a', 'ir')
>>> re.match(f'([{consonants}]+)?([aeiou])(.*)', 'apple').groups()
(None, 'a', 'pple')
```

As you can see, this fails to match when the string contains no vowels or letters:

```
>>> type(re.match(f'([{consonants}]+)?([aeiou])(.*)', 'rdnzl'))
<class 'NoneType'>
>>> type(re.match(f'([{consonants}]+)?([aeiou])(.*)', '123'))
<class 'NoneType'>
```

14.1.4 Truthiness

We know that our program will receive some inputs that cannot be rhymed, so what should the `stemmer()` function do with these? Some people like to use exceptions in cases like this. We've encountered exceptions like asking for a list index or a dictionary key that does not exist. If exceptions are not caught and handled, they cause our programs to crash!

I try to avoid writing code that creates exceptions. I decided that my `stemmer()` function 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`. Here is one way I could write the code for returning those tuples:

```
if match:
    p1 = match.group(1) or ''
    p2 = match.group(2) or ''
    p3 = match.group(3) or ''
    return (p1, p2 + p3)
else:
    return (word, '')
```

The match will be None if the regex failed, which is "falsey." If it succeeds, then it will be "truthy."

There are three capture groups that we can put into three variables. We want to ensure we don't return any None values, so we can use an "or" to evaluate the left side as "truthy" and take the empty string on the right if it's not.

Return a tuple that has the first part of the word (maybe consonants) and the "rest" of the word (the vowel plus anything else).

If the match was None, return a tuple of the word and an empty string to indicate there is no "rest" of the word to rhyme.

Let's take a moment to think about the `or` operator, which we're using 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
True
>>> False or True
True
>>> 1 or 0
1
>>> 0 or 1
1
>>> 0.0 or 1.0
1.0
>>> '0' or ''
'0'
>>> 0 or False
False
>>> [] or ['foo']
['foo']
>>> {} or dict(foo=1)
{'foo': 1}
```

It's easiest to see with literal True and False values.

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

In a Boolean context, the integer value 0 is "falsey," and any other value is "truthy."

The number values behave exactly like actual Boolean values.

Floating-point values also behave like integer values, where 0.0 is "falsey" and anything else is "truthy."

With string values, the empty string is "falsey" and 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, so philosophical.

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

The empty dict is "falsey," and any non-empty dict is "truthy."

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

You should be able to use these ideas to write a `stemmer()` function that will pass the `test_stemmer()` function. Remember, if both of these functions are in your rhymer.py program, you can run the `test_` functions like so:

```
$ pytest -xv rhymer.py
```

14.1.5 Creating the output

Let's review what the program should do:

1. Take a positional string argument.
2. Try to split it into two parts: any leading consonants and the rest of the word.
3. 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. Be sure to *not* include the original consonant sound and to sort the rhyming strings.
4. If you are unable to split the word, print the message `Cannot rhyme "<word>"`.

Now it's time to write the program. Have fun storming the castle!

14.2 Solution

"No more rhymes now, I mean it!"
"Anybody want a peanut?"

Let's take a look at one way to solve this problem. How different was your solution?

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

import argparse
import re
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='word', help='A word to rhyme')

    return parser.parse_args()


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

    args = get_args()
    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 thw wh wr '
        'sch scr shr sph spl spr squ str thr').split()

    start, rest = stemmer(args.word)
    if rest:
```

The re module is for regular expressions.

Get the command-line arguments.

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

Split the word argument into two possible parts. Because the stemmer() function always returns a 2-tuple, we can unpack the values into two variables.

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

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.

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

If there is nothing for the "rest" of the word that can be used to create rhymes, let the user know.

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

    word = word.lower()
    vowels = 'aeiou'
    consonants = ''.join(
        [c for c in string.ascii_lowercase if c not in vowels])
    pattern = (
        '([' + consonants + ']+)?' # capture one or more, optional
        '([' + vowels     + '])'   # capture at least one vowel
        '(.*)'                     # capture zero or more of anything
    )

    match = re.match(pattern, word)
    if match:
        p1 = match.group(1) or ''
        p2 = match.group(2) or ''
        p3 = match.group(3) or ''
        return (p1, p2 + p3)
    else:
        return (word, '')
```

Lowercase the word.

Since we will use the vowels more than once, assign them to a variable.

The consonants are the letters that are not vowels. We will only match to lowercase letters.

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

Use the re.match() function to start matching at the beginning of the word.

The re.match() function will return None if the pattern failed to match, so check if the match is "truthy" (not None).

Put each group into a variable, always ensuring that we use the empty string rather than None.

Return a new tuple that has the "first" part of the word (possible leading consonants) and the "rest" of the word (the vowel plus anything else).

If the match failed, return the word and an empty string for the "rest" of the word to indicate there is nothing to rhyme.

```
# --------------------------------------------------
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', '')
    assert stemmer('123') == ('', '')


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

The tests for the stemmer() function. I usually like to put my unit tests directly after the functions they test.

14.3 Discussion

There are many ways you could have written this, but, as always, I wanted to break the problem down into 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.

14.3.1 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])
```

Throughout the chapters, I have shown how a list comprehension is a concise way to generate a list and is preferable to using a `for` loop to append to an existing list. Here we have added an `if` statement to only include some characters if they are not vowels. This is called a *guard* statement, and only those elements that evaluate as "truthy" will be included in the resulting `list`.

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 and will apply it to all the elements from some *iterable* (something that can be *iterated*, like a `list`). Here I'd like to introduce another HOF called `filter()`, which also takes a function and an iterable (see figure 14.12). As with the list comprehension with the guard, only those elements that return a "truthy" value from the function are allowed in the resulting `list`.

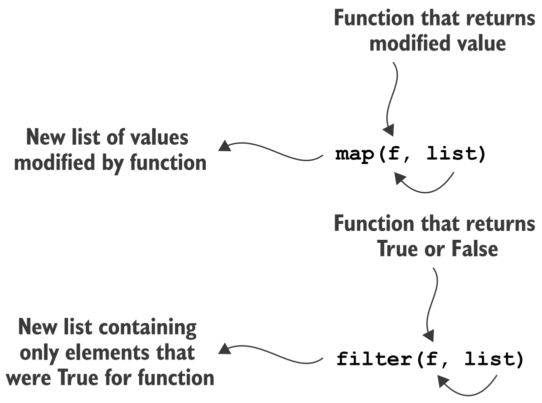

Function that returns modified value
New list of values modified by function
map(f, list)
Function that returns True or False
New list containing only elements that were True for function
filter(f, list)

Figure 14.12 The `map()` and `filter()` functions both take a function and an iterable, and both produce a new list.

Here is another way to write the idea of the list comprehension using `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 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 `filter()`. Think back to our "blue" cars; let's write a `filter()` that only accepts cars that start with the string "blue":

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

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

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

Note that if none of the elements from the original iterable are accepted, `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]
[]
```

Figure 14.13 shows the relationship between 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 technique is probably the list comprehension. The `for` loop would be very familiar to a C or Java programmer, while the `filter()` approach 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.

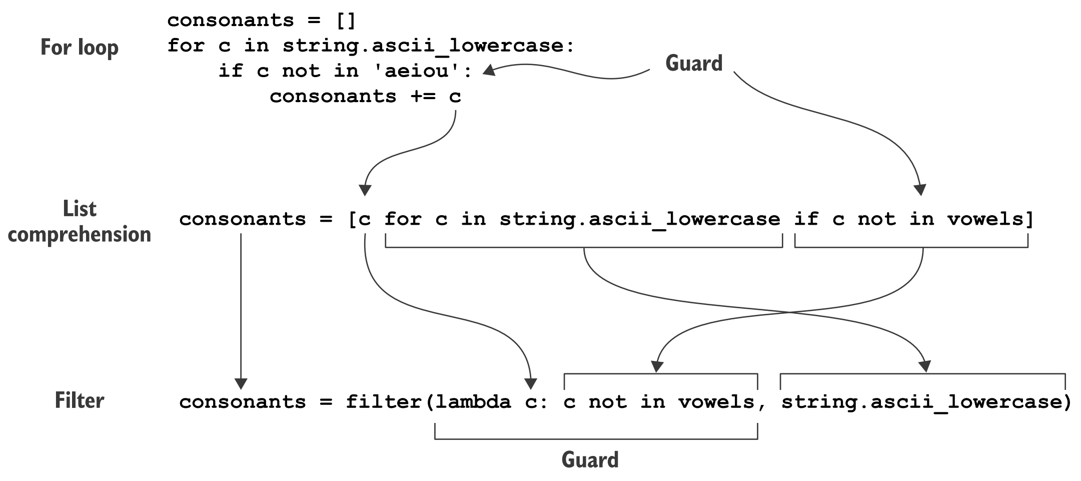

For loop
consonants = []
for c in string.ascii_lowercase:
    if c not in 'aeiou':
        consonants += c
Guard
List comprehension
consonants = [c for c in string.ascii_lowercase if c not in vowels]
Filter
consonants = filter(lambda c: c not in vowels, string.ascii_lowercase)
Guard

Figure 14.13 Three ways to create a list of consonants: using a `for` loop with an `if` statement, a list comprehension with a guard, and a `filter()`

14.3.2 Formatting and commenting the 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. I used an interesting trick of the Python interpreter that will implicitly concatenate adjacent string literals. See how these four strings become one:

```
>>> 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 four 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 on separates lines is that you can add comments to help your reader understand each part:

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

The individual strings will be concatenated by Python into a single string:

```
>>> pattern
'([bcdfghjklmnpqrstvwxyz]+)?([aeiou])(.*)'
```

I could have written the entire regex on one line, but ask yourself which version would you rather read and maintain, the preceding version or the following:[1]

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

14.3.3 Using the stemmer() function outside your program

One of the very interesting things about Python code is that your rhymer.py program is also—kind of, sort of—a sharable *module* of code. That is, you haven't explicitly written it to be a container of reusable (and tested!) 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
```

Now you can run and test your `stemmer()` function manually:

```
>>> stemmer('apple')
('', 'apple')
>>> stemmer('banana')
('b', 'anana')
>>> import string
>>> stemmer(string.punctuation)
('!"#$%&\'()*+,-./:;<=>?@[\\]^_`{|}~', '')
```

The deeper meaning of `if __name__ == '__main__':`

Note that if you were to change the last two lines of rhymer.py from this,

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

to this,

```
main()
```

the `main()` function would be run when you try to import the module:

```
>>> from rhymer import stemmer
usage: [-h] str
: error: the following arguments are required: str
```

[1] "Looking at code you wrote more than two weeks ago is like looking at code you are seeing for the first time."—Dan Hurvitz

(continued)

This is because `import rhymer` causes Python to execute the rhymer.py file to the end. If the last line of the module calls `main()`, then `main()` will run!

The `__name__` variable is set to `'__main__'` when rhymer.py is being *run as a program*. That is the only time `main()` is executed. When the module is being imported by another module, then `__name__` is equal to `rhymer`.

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

```
>>> import rhymer
>>> rhymer.stemmer('cake')
('c', 'ake')
>>> rhymer.stemmer('chair')
('ch', 'air')
```

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 than to copy 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 alternative is to find the duplicated code in every program and change it (hoping that this doesn't introduce even more problems because the code is entangled with other code).

14.3.4 Creating rhyming strings

I decided that my `stemmer()` function would always return a 2-tuple of the `(start, rest)` for any given word. As such, I can unpack the two values into two 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` that 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}"')
```

14.3.5 Writing stemmer() without regular expressions

It is certainly possible to write a solution that does not use regular expressions. We could start by finding the first position of a vowel in the given string. If one is present, we could use a list slice to return the portion of the string up to that position and the portion starting at that position:

Lowercase the given word to avoid dealing with uppercase letters.

Filter the vowels 'aeiou' to find those in word, and then map the present vowels to word.index to find their positions. This is one of the rare instances when we need to use the list() function to coerce Python into evaluating the lazy map() function because the next if statement needs a concrete value.

```
def stemmer(word):
    """Return leading consonants (if any), and 'stem' of word"""
    word = word.lower()
    vowel_pos = list(map(word.index, filter(lambda v: v in word, 'aeiou')))

    if vowel_pos:
        first_vowel = min(vowel_pos)
        return (word[:first_vowel], word[first_vowel:])
    else:
        return (word, '')
```

Check if there are any vowels present in the word.

Find the index of the first vowel by taking the minimum (min) value from the positions.

Return a tuple of a slice of the word up to the first vowel, and another starting at the first vowel.

Otherwise, no vowels were found in the word.

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. In my mind, the `stemmer()` function 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, 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.

14.4 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 program in chapter 6 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 finds all unique consonant sounds in a dictionary of English words. (I have included inputs/words.txt.zip, which is a compressed version of the dictionary from my machine. Unzip the file to use inputs/words.txt.) Print the output in alphabetical order and use those to expand this program's consonants.
- Alter your program to only emit words that are found in the system dictionary (for example, inputs/words.txt).
- 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, 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."

Summary

- Regular expressions allow you to declare a pattern that you wish to find. The regex *engine* will sort out whether the pattern is found or not. This is a *declarative* approach to programming, in contrast to the *imperative* method of manually seeking out patterns by writing code ourselves.

- You can wrap parts of the pattern in parentheses to "capture" them into groups that you 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 that are deemed "truthy" by the function 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," so any non-falsey value from those types, like the non-empty `str`, `list`, or `dict`, or any numeric value not zero-ish, will be considered "truthy."
- You can break long string literals into shorter adjacent strings in your code to have Python join them into one 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.