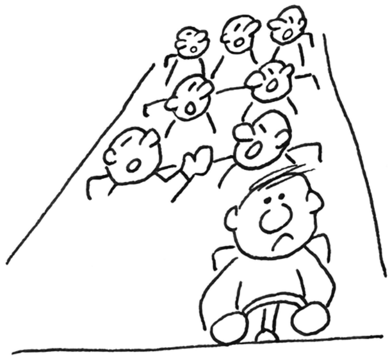
11 Bottles of Beer Song: Writing and testing functions



Few songs are as annoying as “99 Bottles of Beer on the Wall.” Hopefully you’ve never had to ride for hours in a van with middle school boys who like to sing this. I have. It’s a fairly simple song that we can write an algorithm to generate. This will give us an opportunity to play with counting up and down, formatting strings, and—new to this exercise—writing functions and tests for those functions!

Our program will be called bottles.py and will take one option, `-n` or `--num`, which must be a *positive* `int` (the default will be `10`). The program should print all the verses from `--num` down to 1. There should be two newlines between each verse to visually separate them, but there must be only one newline after the last verse (for one bottle), which should print “No more bottles of beer on the wall” rather than “0 bottles”:

```
$ ./bottles.py -n 3
3 bottles of beer on the wall,
3 bottles of beer,
Take one down, pass it around,
2 bottles of beer on the wall!

2 bottles of beer on the wall,
2 bottles of beer,
Take one down, pass it around,
1 bottle of beer on the wall!
```

```
1 bottle of beer on the wall,
1 bottle of beer,
Take one down, pass it around,
No more bottles of beer on the wall!
```

In this exercise, you will

- Learn how to produce a list of numbers decreasing in value
- Write a function to create a verse of the song, using a test to verify when the verse is correct
- Explore how `for` loops can be written as list comprehensions, which in turn can be written with the `map()` function

11.1 Writing bottles.py

We'll be working in the 11_bottles_of_beer directory. Start off by copying template.py or using new.py to create your bottles.py program there. Then modify the `get_args()` function until your usage matches the following usage statement. You need to define only the `--num` option with `type=int` and `default=10`:

```
$ ./bottles.py -h
usage: bottles.py [-h] [-n number]

Bottles of beer song

optional arguments:
  -h, --help            show this help message and exit
  -n number, --num number
                        How many bottles (default: 10)
```

If the `--num` argument is not an `int` value, your program should print an error message and exit with an error value. This should happen automatically if you define your parameter to `argparse` properly:

```
$ ./bottles.py -n foo
usage: bottles.py [-h] [-n number]
bottles.py: error: argument -n/--num: invalid int value: 'foo'
$ ./bottles.py -n 2.4
usage: bottles.py [-h] [-n number]
bottles.py: error: argument -n/--num: invalid int value: '2.4'
```

Since we can't sing zero or fewer verses, we'll need to check if `--num` is less than `1`. To handle this, I suggest you use `parser.error()` inside the `get_args()` function, as in previous exercises:

```
$ ./bottles.py -n 0
usage: bottles.py [-h] [-n number]
bottles.py: error: --num "0" must be greater than 0
```

Figure 11.1 shows a string diagram of the inputs and outputs.

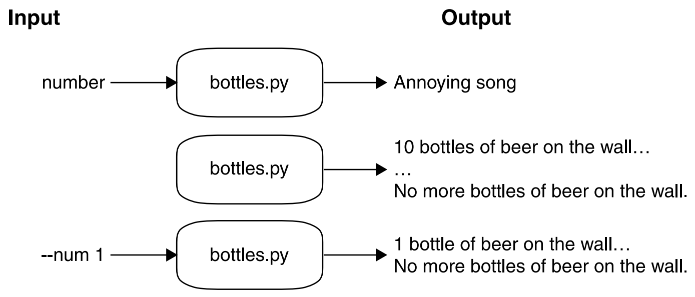
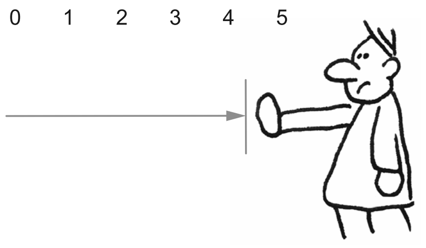

Input
Output
number
bottles.py
Annoying song
bottles.py
10 bottles of beer on the wall…
…
No more bottles of beer on the wall.
--num 1
bottles.py
1 bottle of beer on the wall…
No more bottles of beer on the wall.

Figure 11.1 The bottles program may take a number for the verse to start, or it will sing the song starting at 10.

11.1.1 Counting down

The song starts at the given `--num` value, like 10, and needs to count down to 9, 8, 7, and so forth. How can we do that in Python? We've seen how to use `range(start, stop)` to get a list of integers that go *up* in value. If you give it just one number, that will be considered the `stop`, and it will assume `0` as the `start`:

```
>>> list(range(5))
[0, 1, 2, 3, 4]
```



Because this is a lazy function, we must use the `list()` function in the REPL to force it to produce the numbers. Remember that the `stop` value is never included in the output, so the preceding output stopped at 4, not 5.

If you give `range()` two numbers, they are considered to be `start` and `stop`:

```
>>> list(range(1, 5))
[1, 2, 3, 4]
```

To reverse this sequence, you might be tempted to swap the `start` and `stop` values. Unfortunately, if `start` is greater than `stop`, you get an empty list:

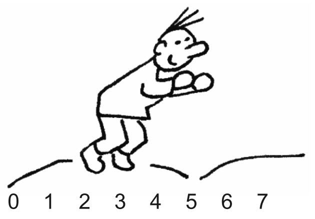
```
>>> list(range(5, 1))
[]
```

You saw in chapter 3 that we can use the `reversed()` function to reverse a `list`. This is another lazy function, so again I'll use the `list()` function to force the values in the REPL:

```
>>> list(reversed(range(1, 5)))
[4, 3, 2, 1]
```

The range() function can also take an optional third argument for a step value. For instance, you could use this to count by fives:

```
>>> list(range(0, 50, 5))
[0, 5, 10, 15, 20, 25, 30, 35, 40, 45]
```



Another way to count down is to swap the start and stop and use -1 for the step:

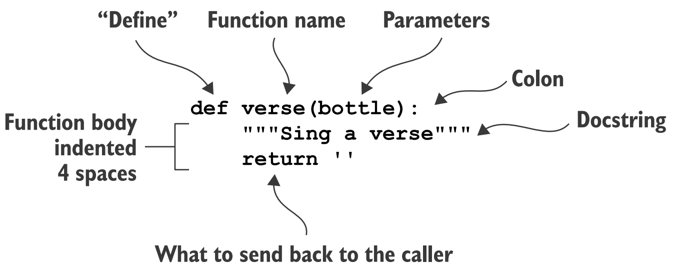
```
>>> list(range(5, 0, -1))
[5, 4, 3, 2, 1]
```

So you have couple of ways to count in reverse.

11.1.2 Writing a function

Up to this point, I've suggested that all your code go into the main() function. This is the first exercise where I suggest you write a function. I would like you to consider how to write the code to sing *just one verse*. The function could take the number of the verse and return the text for that verse.

You can start off with something like the example in figure 11.2. The def keyword "defines" a function, and the name of the function follows. Function names should contain only letters, numbers, and underscores and cannot start with a number. After the name comes parentheses, which describe any parameters that the function accepts. Here our function will be called verse(), and it has the parameter bottle (or number or whatever you want to call it). After the parameters comes a colon to indicate the end of the def line. The function body comes next, with all lines being indented at least four spaces.


"Define"
Function name
Parameters
Colon
Docstring
Function body indented 4 spaces
What to send back to the caller

```
def verse(bottle):
    """Sing a verse"""
    return ''
```

Figure 11.2 The elements of a function definition in Python

The docstring in figure 11.2 is a string just after the function definition. It will show up in the help for your function.

You can enter this function into the REPL:

```
>>> def verse(bottle):
...     """Sing a verse"""
...     return ''
...
>>> help(verse)
```

When you do, you will see this:

```
Help on function verse in module __main__:

verse(bottle)
    Sing a verse
```

The `return` statement tells Python what to send back from the function. It's not very interesting right now because it will just send back the empty string:

```
>>> verse(10)
''
```

It's also common practice to use the `pass` statement for the body of a dummy function. The `pass` will do nothing, and the function will return `None` instead of the empty string, as we have done here. When you start writing your own functions and tests, you might like to use `pass` when you stub out a new function, until you decide what the function will do.

11.1.3 Writing a test for verse()

In the spirit of *test-driven development*, let's write a test for `verse()` before we go any further. The following listing shows a test you can use. Add this code into your bottles.py program just after your `main()` function :

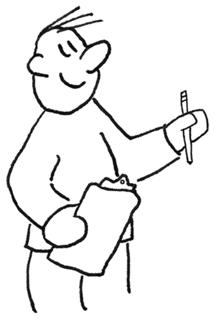
```
def verse(bottle):
    """Sing a verse"""

    return ''


def test_verse():
    """Test verse"""

    last_verse = verse(1)
    assert last_verse == '\n'.join([
        '1 bottle of beer on the wall,', '1 bottle of beer,',
        'Take one down, pass it around,',
        'No more bottles of beer on the wall!'
    ])

    two_bottles = verse(2)
    assert two_bottles == '\n'.join([
        '2 bottles of beer on the wall,', '2 bottles of beer,',
        'Take one down, pass it around,', '1 bottle of beer on the wall!'
    ])
```



There are many, many ways you could write this program. I have in mind that my `verse()` function will produce a single verse of the song, returning a new `str` value that is the lines of the verse joined on newlines.

You don't have to write your program this way, but I'd like you to consider what it means to write a function and a *unit test.* If you read about software testing, you'll find that there are different definitions of what a "unit" of code is. In this book, I consider a *function* to be a *unit,* so my unit tests are tests of individual functions.

Even though the song has potentially hundreds of verses, these two tests should cover everything you need to check. It may help to look at the musical notation in figure 11.3 for the song, as this does a nice job of graphically showing the structure of the song and, hence, our program.

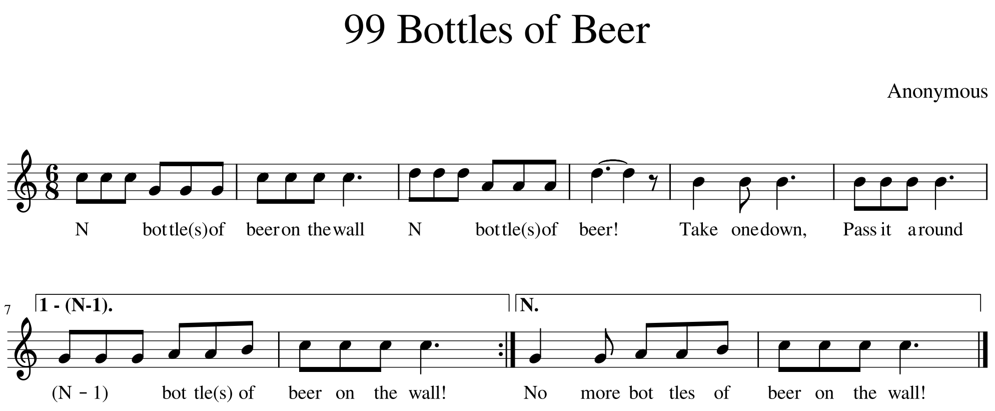

99 Bottles of Beer
Anonymous
N bottle(s) of beer on the wall N bottle(s) of beer! Take one down, Pass it around
7
1 - (N-1).
N.
(N - 1) bot tle(s) of beer on the wall! No more bot tles of beer on the wall!

Figure 11.3 The musical notation for the song shows there are two cases to handle: one for all the verses up to the last, and then the last one.

I've taken a few liberties with the notation by mixing in some programming ideas. If you don't know how to read music, let me briefly explain the important parts. The `N` is the current *number,* like "99" so that `(N - 1)` would be "98." The endings are noted `1 - (N - 1)`, which is a bit confusing because we're using the hyphen to indicate both a range and subtraction in the same "equation." Still, the first ending is used for the first time through the penultimate repeat. The colon before the bar lines in the first ending means to repeat the song from the beginning. Then the `N` ending is taken on the last repeat, and the double bar indicates the end of the song/program.

What we can see from the music is that there are only two cases we need to handle: the last verse, and all the other verses. So first we check the last verse. We're looking for "1 bottle" (singular) and not "1 bottles" (plural). We also need to check that the last line says "No more bottles" instead of "0 bottles." The second test, for "2 bottles of beer," is making sure that the numbers are "2 bottles" and then "1 bottle." If we managed to pass these two tests, our program ought to be able to handle all the verses.

I wrote `test_verse()` to test just the `verse()` function. The name of the function matters because I am using the `pytest` module to find all the functions in my code that start with `test_` and run them. If your bottles.py program has the preceding functions for `verse()` and `test_verse()`, you can run `pytest bottles.py`.

Try it, and you should see something like this:

```
$ pytest bottles.py
============================= test session starts ==============================
...
collected 1 item

bottles.py F                                                             [100%]

=================================== FAILURES ===================================
__________________________________ test_verse __________________________________

    def test_verse():
        """Test verse"""

        last_verse = verse(1)
>       assert last_verse == '\n'.join([
            '1 bottle of beer on the wall,', '1 bottle of beer,',
            'Take one down, pass it around,',
            'No more bottles of beer on the wall!'
        ])
E       AssertionError: assert '' == '1 bottle of beer on the wal...ottles of
     beer on the wall!'
E         + 1 bottle of beer on the wall,
E         + 1 bottle of beer,
E         + Take one down, pass it around,
E         + No more bottles of beer on the wall!

bottles.py:49: AssertionError
=========================== 1 failed in 0.10 seconds ===========================
```

Call the verse() function with the argument 1 to get the last verse of the song.

The > at the beginning of this line indicates this is the source of the error. The test checks if the value of last_verse is equal to an expected str value. Since it's not, this line throws an exception, causing the assertion to fail.

The "E" lines show the difference between what was received and what was expected. The value of last_verse is the empty string (''), which does not match the expected string "1 bottle of beer..." and so on.

To pass the first test, you could copy the code for the expected value of `last_verse` directly from the test. Change your `verse()` function to match this:

```
def verse(bottle):
    """Sing a verse"""

    return '\n'.join([
        '1 bottle of beer on the wall,', '1 bottle of beer,',
        'Take one down, pass it around,',
        'No more bottles of beer on the wall!'
    ])
```

Now run your test again. The first test should pass, and the second one should fail. Here are the relevant error lines:

```
=================================== FAILURES ===================================
__________________________________ test_verse __________________________________

    def test_verse() -> None:
        """Test verse"""

        last_verse = verse(1)
        assert last_verse == '\n'.join([
            '1 bottle of beer on the wall,', '1 bottle of beer,',
            'Take one down, pass it around,',
            'No more bottles of beer on the wall!'
        ])
```

This test now passes.

Call verse() with the value of 2 to get the “Two bottles…” verse.

Assert that this verse is equal to the expected string.

```
        two_bottles = verse(2)
>       assert two_bottles == '\n'.join([
            '2 bottles of beer on the wall,', '2 bottles of beer,',
            'Take one down, pass it around,', '1 bottle of beer on the wall!'
        ])
E       AssertionError: assert '1 bottle of ... on the wall!' == '2 bottles of
     ... on the wall!'
E         - 1 bottle of beer on the wall,
E         ? ^
E         + 2 bottles of beer on the wall,
E         ? ^       +
E         - 1 bottle of beer,
E         ? ^
E         + 2 bottles of beer,...
E
E         ...Full output truncated (7 lines hidden), use '-vv' to show
```

These E lines are showing you the problem. The verse() function returned '1 bottle' but the test expected '2 bottles', etc.

Go back and look at your `verse()` definition. Look at figure 11.4 and think about which parts need to change—the first, second, and fourth lines. The third line is always the same. You’re given a value for `bottle` that needs to be used in the first two lines, along with either “bottle” or “bottles,” depending on the value of `bottle`. (Hint: It’s only singular for the value `1`; otherwise, it’s plural.) The fourth line needs the value of `bottle - 1` and, again, the proper singular or plural depending on that value. Can you figure out how to write this?

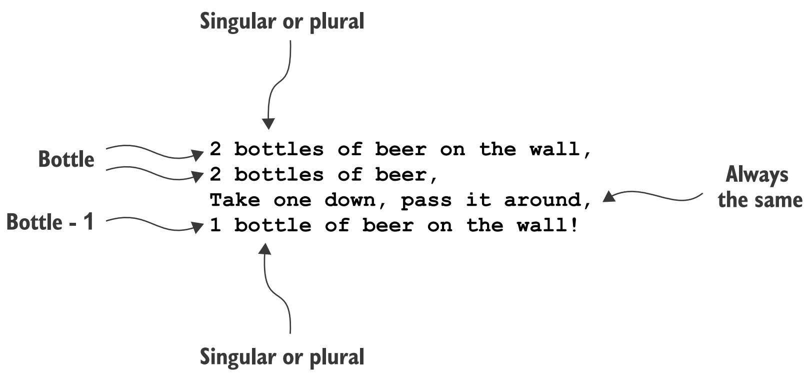

Singular or plural
Bottle
Bottle - 1
2 bottles of beer on the wall,
2 bottles of beer,
Take one down, pass it around,
1 bottle of beer on the wall!
Always the same
Singular or plural

Figure 11.4 Each verse has four lines, where the first two and last are very similar. The third line is always the same. Find the parts that vary.

Focus on passing those two tests before you move to the next stage of printing the whole song. That is, do not attempt anything until you see this:

```
$ pytest bottles.py
============================= test session starts ==============================
...
collected 1 item

bottles.py .                                                             [100%]

=========================== 1 passed in 0.05 seconds ===========================
```

11.1.4 Using the verse() function

At this point, you know

- That the `--num` value is a valid integer value greater than 0
- How to count from that `--num` value backwards down to 0
- That the `verse()` function will print any one verse properly

Now you need to put them together. I suggest you start by using a `for` loop with the `range()` function to count down. Use each value from that to produce a `verse()`. There should be two newlines after every verse except for the last.

You will use `pytest -xv test.py` (or `make test`) to test the program at this point. In the parlance of testing, test.py is an *integration test* because it checks that the program *as a whole* is working. From this point on, we'll focus on how to write *unit* tests to check individual functions in addition to integration tests to ensure that all the functions work together.

Once you can pass the test suite using a `for` loop, try to rewrite it using either a list comprehension or a `map()`. Rather than starting again from scratch, I suggest you comment out your working code by adding # to the beginnings of the lines, and then try other ways to write the algorithm. Use the tests to verify that your code still passes. If it is at all motivating, my solution is one line long. Can you write a single line of code that combines the `range()` and `verse()` functions to produce the expected output?

Here are a few hints:

- Define the `--num` argument as an `int` with a default value of `10`.
- Use `parser.error()` to get `argparse` to print an error message for a negative `--num` value.
- Write the `verse()` function. Use the `test_verse()` function and Pytest to make that work properly.
- Combine the `verse()` function with `range()` to create all the verses.

Do try your best to write the program before reading the solution. Also feel free to solve the problem in a completely different way, even writing your own unit tests.

11.2 Solution

I've decided to show you a slightly fancy-pants version that uses `map()`. Later I'll show you how to write it using a `for` loop and a list comprehension.

```
#!/usr/bin/env python3
"""Bottles of beer song"""

import argparse


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

    parser = argparse.ArgumentParser(
        description='Bottles of beer song',
        formatter_class=argparse.ArgumentDefaultsHelpFormatter)

    parser.add_argument('-n',
                        '--num',
                        metavar='number',
                        type=int,
                        default=10,
                        help='How many bottles')
```

Define the --num argument as an int with a default value of 10.

```
    args = parser.parse_args()
```

Parse the command-line argument into the variable args.

```
    if args.num < 1:
        parser.error(f'--num "{args.num}" must be greater than 0')

    return args
```

If args.num is less than 1, use parser.error() to display an error message and exit the program with an error value.

The map() function expects a function as the first argument and some iterable as the second argument. Here I feed the descending numbers from the range() function to my verse() function. The result from map() is a new list of verses that can be joined on two newlines.

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

    args = get_args()
    print('\n\n'.join(map(verse, range(args.num, 0, -1))))
```

Define a function that can create a single verse().

Define a next_bottle that is one less than the current bottle.

Define an s1 (the first "s") that is either the character 's' or the empty string, depending on the value of current bottle.

Do the same for s2 (the second "s"), depending on the value of next_bottle.

Define a value for next_num depending on whether the next value is 0 or not.

Create a return string by joining the four lines of text on the newline. Substitute in the variables to create the correct verse.

```
# --------------------------------------------------
def verse(bottle):
    """Sing a verse"""

    next_bottle = bottle - 1
    s1 = '' if bottle == 1 else 's'
    s2 = '' if next_bottle == 1 else 's'
    num_next = 'No more' if next_bottle == 0 else next_bottle
    return '\n'.join([
        f'{bottle} bottle{s1} of beer on the wall,',
        f'{bottle} bottle{s1} of beer,',
        f'Take one down, pass it around,',
        f'{num_next} bottle{s2} of beer on the wall!',
    ])
```

Define a unit test called test_verse() for the verse() function. The test_ prefix means that the pytest module will find this function and execute it.

Test the last verse() with the value 1.

Test a verse() with the value 2.

```
# --------------------------------------------------
def test_verse():
    """Test verse"""

    last_verse = verse(1)
    assert last_verse == '\n'.join([
        '1 bottle of beer on the wall,', '1 bottle of beer,',
        'Take one down, pass it around,',
        'No more bottles of beer on the wall!'
    ])

    two_bottles = verse(2)
    assert two_bottles == '\n'.join([
        '2 bottles of beer on the wall,', '2 bottles of beer,',
        'Take one down, pass it around,', '1 bottle of beer on the wall!'
    ])


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

11.3 Discussion

There isn't anything new in the `get_args()` function in this program. By this point, you have had several opportunities to define an optional integer parameter with a default argument and to use `parser.error()` to halt your program if the user provides a bad argument. By relying on `argparse` to handle so much busy work, you are saving yourself loads of time as well as ensuring that you have good data to work with. Let's move on to the new stuff!

11.3.1 Counting down

You know how to count down from the given `--num`, and you know you can use a `for` loop to iterate:

```
>>> for n in range(3, 0, -1):
...     print(f'{n} bottles of beer')
...
3 bottles of beer
2 bottles of beer
1 bottles of beer
```

Instead of directly creating each verse inside the `for` loop, I suggested that you could create a function called `verse()` to create any given verse and use that with the `range()` of numbers. Up to this point, we've been doing all our work in the `main()` function. As you grow as a programmer, though, your programs will become longer—hundreds to even thousands of lines of code (LOC). Long programs and functions can get very difficult to test and maintain, so you should try to break ideas into small, functional units that you can understand and test. Ideally, functions should do *one* thing. If you understand and trust your small, simple *functions*, then you know you can safely compose them into longer, more complicated *programs*.

11.3.2 Test-driven development

I wanted you to add a `test_verse()` function to your program to use with Pytest to create a working `verse()` function. This idea follows the principles described by Kent Beck in his book, *Test-Driven Development* (Addison-Wesley Professional, 2002):

1. Add a new test for an unimplemented unit of functionality.
2. Run all previously written tests and see the newly added test fails.
3. Write code that implements the new functionality.
4. Run all tests and see them succeed.
5. Refactor (rewrite to improve readability or structure).
6. Start at the beginning (repeat).

For instance, suppose we want a function that adds 1 to any given number. We'll called it `add1()` and define the function body as `pass` to tell Python "nothing to see here":

```
def add1(n):
    pass
```

Now write a `test_add1()` function where you pass some arguments to the function, and use `assert` to verify that you get back the value that you expect:

```
def test_add1():
    assert add1(0) = 1
    assert add1(1) = 2
    assert add1(-1) = 0
```

Run pytest (or whatever testing framework you like) and verify that the function *does not work* (of course it won't, because it just executes `pass`). Then go fill in some function code that *does* work (`return n + 1` instead of `pass`). Pass all manner of arguments you can imagine, including nothing, one thing, and many things.[1]

11.3.3 The verse() function

I provided you with a `test_verse()` function that shows you exactly what is expected for the arguments of 1 and 2. What I like about writing my tests first is that it gives me an opportunity to think about how I'd like to use the code, what I'd like to give as arguments, and what I expect to get back in return. For instance, what *should* the function `add1()` return if given

- No arguments
- More than one argument
- The value `None`
- Anything other than a numeric type (`int`, `float`, or `complex`) like a `str` value or a `dict`

You can write tests to pass both good and bad values and decide how you want your code to behave under both favorable and adverse conditions.

Here's the `verse()` function I wrote, which passes the `test_verse()` function:

```
def verse(bottle):
    """Sing a verse"""

    next_bottle = bottle - 1
    s1 = '' if bottle == 1 else 's'
    s2 = '' if next_bottle == 1 else 's'
    num_next = 'No more' if next_bottle == 0 else next_bottle
    return '\n'.join([
        f'{bottle} bottle{s1} of beer on the wall,',
        f'{bottle} bottle{s1} of beer,',
        f'Take one down, pass it around,',
        f'{num_next} bottle{s2} of beer on the wall!',
    ])
```

[1] A CS professor once told me in office hours to handle the cases of 0, 1, and *n* (infinity), and that has always stuck with me.

This code is annotated in section 11.2, but I essentially isolate all the parts of the return string that change, and I create variables to substitute into those places. I use `bottle` and `next_bottle` to decide if there should be an “s” or not after the “bottle” strings. I also need to figure out whether to print the next bottle as a number, or if I should print the string “No more” (when `next_bottle` is `0`). Choosing the values for `s1`, `s2`, and `num_next` all involve *binary* decisions, meaning they are a choice between *two* values, so I find it best to use an `if` expression.

This function passes `test_verse()`, so I can move on to using it to generate the song.

11.3.4 Iterating through the verses

I could use a `for` loop to count down and `print()` each `verse()`:

```
>>> for n in range(3, 0, -1):
...     print(verse(n))
...
3 bottles of beer on the wall,
3 bottles of beer,
Take one down, pass it around,
2 bottles of beer on the wall!
2 bottles of beer on the wall,
2 bottles of beer,
Take one down, pass it around,
1 bottle of beer on the wall!
1 bottle of beer on the wall,
1 bottle of beer,
Take one down, pass it around,
No more bottles of beer on the wall!
```

That’s *almost* correct, but we need two newlines in between all the verses. I could use the `end` option to `print` to include two newlines for all values greater than 1:

```
>>> for n in range(3, 0, -1):
...     print(verse(n), end='\n' * (2 if n > 1 else 1))
...
3 bottles of beer on the wall,
3 bottles of beer,
Take one down, pass it around,
2 bottles of beer on the wall!

2 bottles of beer on the wall,
2 bottles of beer,
Take one down, pass it around,
1 bottle of beer on the wall!

1 bottle of beer on the wall,
1 bottle of beer,
Take one down, pass it around,
No more bottles of beer on the wall!
```

I would rather use the `str.join()` method to put two newlines in between items in a `list`. My items are the verses, and I can turn a `for` loop into a list comprehension as shown in figure 11.5.

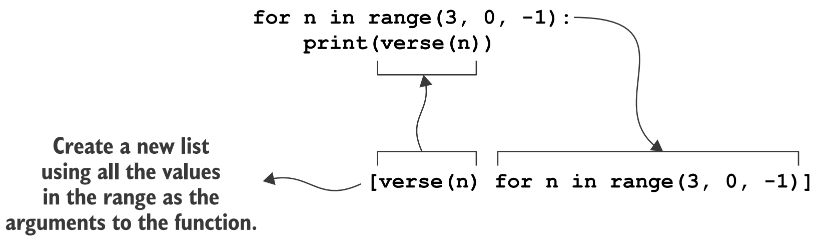

for n in range(3, 0, -1):
print(verse(n))
Create a new list using all the values in the range as the arguments to the function.
[verse(n) for n in range(3, 0, -1)]

Figure 11.5 A `for` loop compared to a list comprehension

```
>>> verses = [verse(n) for n in range(3, 0, -1)]
>>> print('\n\n'.join(verses))
3 bottles of beer on the wall,
3 bottles of beer,
Take one down, pass it around,
2 bottles of beer on the wall!

2 bottles of beer on the wall,
2 bottles of beer,
Take one down, pass it around,
1 bottle of beer on the wall!

1 bottle of beer on the wall,
1 bottle of beer,
Take one down, pass it around,
No more bottles of beer on the wall!
```

That is a fine solution, but I would like you to start noticing a pattern we will see repeatedly: applying a function to every element of a sequence, which is exactly what `map()` does! As shown in figure 11.6, our list comprehension can be rewritten very concisely using `map()`.

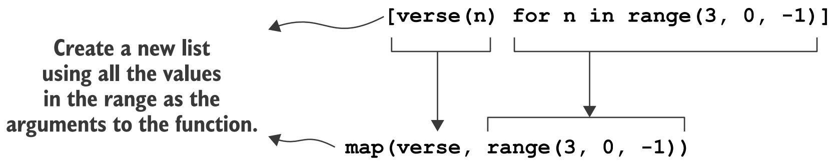

[verse(n) for n in range(3, 0, -1)]
Create a new list using all the values in the range as the arguments to the function.
map(verse, range(3, 0, -1))

Figure 11.6 A list comprehension can be replaced with `map()`. They both return a new `list`.

In our case, our sequence is a descending `range()` of numbers, and we want to apply our `verse()` function to each number and collect the resulting verses. It's like the paint booth idea in chapter 8, where the function "painted" the cars "blue" by adding the word "blue" to the start of the string. When we want to apply a function to every element in a sequence, we might consider refactoring the code using `map()`:

```
>>> verses = map(verse, range(3, 0, -1))
>>> print('\n\n'.join(verses))
3 bottles of beer on the wall,
3 bottles of beer,
Take one down, pass it around,
2 bottles of beer on the wall!

2 bottles of beer on the wall,
2 bottles of beer,
Take one down, pass it around,
1 bottle of beer on the wall!

1 bottle of beer on the wall,
1 bottle of beer,
Take one down, pass it around,
No more bottles of beer on the wall!
```

Whenever I need to transform some sequence of items with some function, I like to start off by thinking about how I'll handle just *one* of the items. I find it's much easier to write and test one function with one input rather than some possibly huge list of operations. List comprehensions are often considered more "Pythonic," but I tend to favor `map()` because it usually involves shorter code. If you search the internet for "python list comprehension map," you'll find that some people think list comprehensions are easier to read than `map()`, but `map()` might possibly be somewhat faster. I wouldn't say either approach is better than the other. It really comes down to taste or perhaps a discussion with your teammates.

If you want to use `map()`, remember that it wants a *function* as the first argument and then a sequence of elements that will become arguments to the function. The `verse()` function (which you've tested!) is the first argument, and the `range()` provides the `list`. The `map()` function will pass each element of the `range()` as an argument to the `verse()` function, as shown in figure 11.7. The result is a new `list` with the return values from all those function calls. Many are the `for` loops that can be better written as mapping a function over a list of arguments!

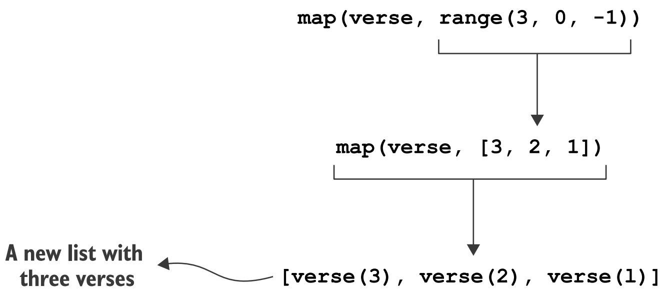

map(verse, range(3, 0, -1))
map(verse, [3, 2, 1])
[verse(3), verse(2), verse(1)]
A new list with three verses

Figure 11.7 The `map()` function will call the `verse()` function with each element produced by the `range()` function. It's functions all the way down.

11.3.5 1,500 other solutions

There are literally hundreds of ways to solve this problem. The "99 Bottles of Beer" website (www.99-bottles-of-beer.net) claims to have 1,500 variations in various languages. Compare your solution to others there. Trivial as the program may be, it has allowed us to explore some really interesting ideas in Python, testing, and algorithms.

11.4 Going further

- Replace the Arabic numbers (1, 2, 3) with text (one, two, three).
- Add a `--step` option (positive `int`, default `1`) that allows the user to skip numbers, like by twos or fives.
- Add a `--reverse` flag to reverse the order of the verses, counting up instead of down.

Summary

- Test-driven development (TDD) is central to developing dependable, reproducible code. Tests also give you the freedom to refactor your code (reorganize and improve it for speed or clarity), knowing that you can always verify your new version still works properly. As you write your code, always write tests!
- The `range()` function will count backwards if you swap `start` and `stop` and supply the optional third `step` value of `-1`.
- A `for` loop can often be replaced with a list comprehension or a `map()` for shorter, more concise code.