Chapter 12: Bottles of Beer Song: Writing and testing functions

Few songs are as annoying as "100 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. The author has. It's a fairly simple song that we can write an algorithm to generate. This gives 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` (default `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 --num 2
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.

Writing `bottles.py`

As always, the program should respond to `-h` or `--help` with a usage statement:

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

Bottles of beer song

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

Start off by copying the `template.py` or using `new.py` to create your `bottles.py` program. Then modify the `get_args` function until your usage matches the above. Since the `-h` and `--help` flags are automatically handled by `argparse`, you need define only the `--num` option with `type=int` and `default=10`.

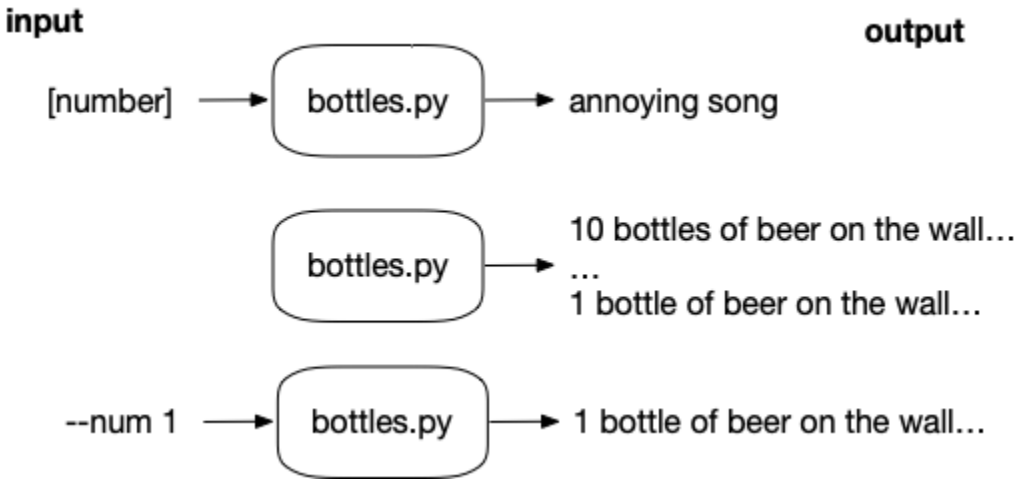
If the `--num` argument is not an `int` value, your program should print an error message and exit with an error value. This should happen automatically if you define your parameter to `argparse` properly:

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

It should do likewise when the `--num` is a negative number. To handle this, I suggest you manually check this and call `parser.error` inside the `get_args` functions as in previous exercises:

```
$ ./bottles.py --num -4
usage: bottles.py [-h] [-n int]
bottles.py: error: --num (-4) must > 0
```

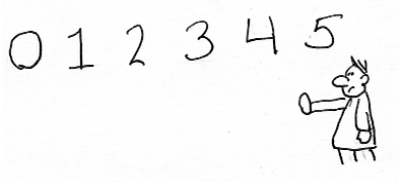
Here is a string diagram of the inputs and outputs:



Counting down

The song starts at the given `--num` value like 10 and needs to count down to 9, 8, 7, and so forth. So how do 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, it will be consider the `stop` and will assume `0` as the `start`. Because this is a lazy function, I must use `list` in the REPL to force it to produce the numbers:

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



Remember that the `stop` value is never included in the output, so the above stopped at 4 and not 5. If you give `range` two numbers, they are considered to be `start` and `stop`:

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

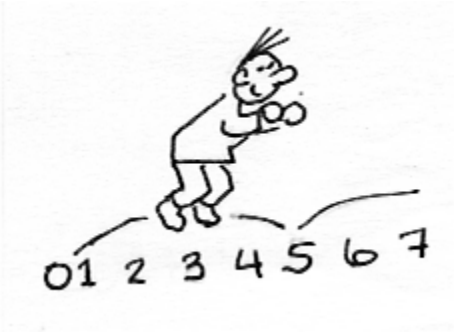
To reverse this sequence, you might be tempted to swap the `start` and `stop` values. Unfortunately, if the `start` is greater than the `stop`, you get an empty list:

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

We saw in "Picnic" that we can use the `reversed` function to reverse a `list`. This is another lazy function, so, again, I 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 5s:



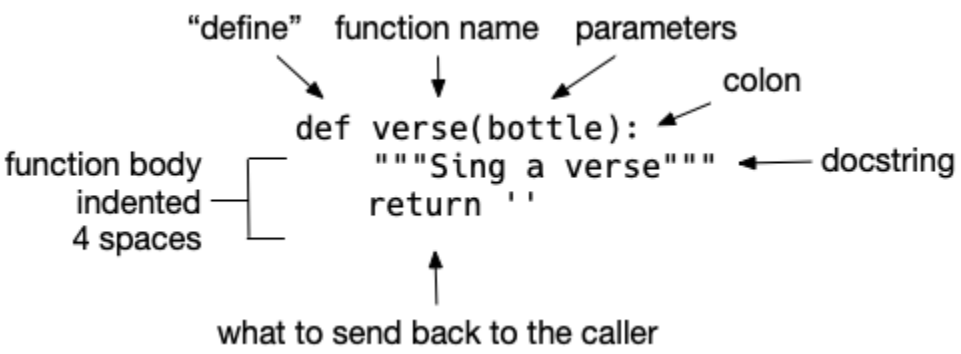

```
>>> list(range(0, 50, 5))
[0, 5, 10, 15, 20, 25, 30, 35, 40, 45]
```

So 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]
```

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 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 diagram. The `def` keyword "defines" a function, and the name of the function follows. Function names should have only letters, numbers and the underscore and cannot start with a number. After the name comes parentheses that hold any parameters that the function accepts. Here our function should be called `verse` and has the parameter `bottle` (or `number` or whatever you want to call it). After the parameters comes a colon to show the end of the `def` line. The function body comes next, with all lines being indented at least 4 spaces.

The "docstring" is a comment just after the function definition and will show up in the `help` for your function. If you enter this into the REPL:

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

Then you will see:

```
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 because it will only ever send back the empty string right now:

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

Writing a test for `verse`

In the spirit of *test-driven development*, let us write a test for `verse` before we go any further. Here is a test you can use. Copy and paste this code as a new function into your `bottles.py` program:

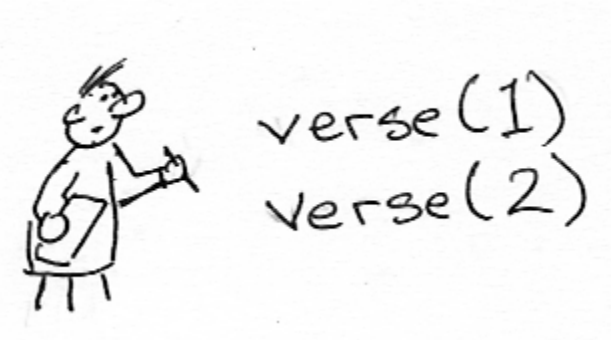
```
# --------------------------------------------------
def test_verse():
    """Test verse"""

    one = verse(1)
    assert one == '\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 = verse(2)
    assert two == '\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 `str` value which is the four lines of the verse joined on newlines. You don't have to write it this way, but I would 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 it. In this book, I consider a *function* to be a *unit*, and so my unit tests are tests of individual functions.

By the way, even though the song has potentially hundreds of verses, these two tests should cover everything you need to check. The first test shows that we are looking for "1 bottle" (singular) and not "1 bottles" (plural). We also check that the last line says "No more bottles" instead of "0 bottles." The test for "2 bottles of beer" is making sure that the numbers are "2 bottles" and then "1 bottle." Presumably if you've managed to pass those two tests, your program ought to be able to handle any value.



Here I've written `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 above 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 ==============================
platform darwin -- Python 3.7.3, pytest-4.6.5, py-1.8.0, pluggy-0.12.0
rootdir: /Users/kyclark/work/manning/tiny_python_projects/bottles_of_beer
plugins: openfiles-0.3.2, arraydiff-0.3, doctestplus-0.3.0, remotedata-0.3.1, cov-2.7.1
collected 1 item

bottles.py F                                                             [100%]

=================================== FAILURES ===================================
_____________________________________ test_verse _____________________________________

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

        one = verse(1)  1
>       assert one == '\n'.join([  2
            '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!'  3
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 ===========================
```

1. Call the `verse` function with the argument `1` to get the `one` verse of the song.
2. The `>` at the beginning of this line indicates this is the source of the error. The test checks if the value of `one` is equal to an expected `str` value. Since it's not, this line throws an exception causing the assertion to fail.
3. The `E` lines show the difference between what was received and what was expected. The value of `one` 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 `one` directly from the test. Change your `verse` function to be 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!'
    ])
```

And run your test again. Now the first test should pass and the second one should fail. I'll only show the relevant error lines:

```
================================== FAILURES ===================================
_____________________________________ test_verse ______________________________________

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

        one = verse(1)
        assert one == '\n'.join([  1
            '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 = verse(2)  2
>       assert two == '\n'.join([  3
            '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!'  4
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
```

1. This assertion passes this time.
2. Call the `verse` with the value of `2`.
3. Assert that `two` is equal to an expected string.
4. These `E` lines are showing you the problem. It got `'1 bottle'` but expected `'2 bottles'`, etc.

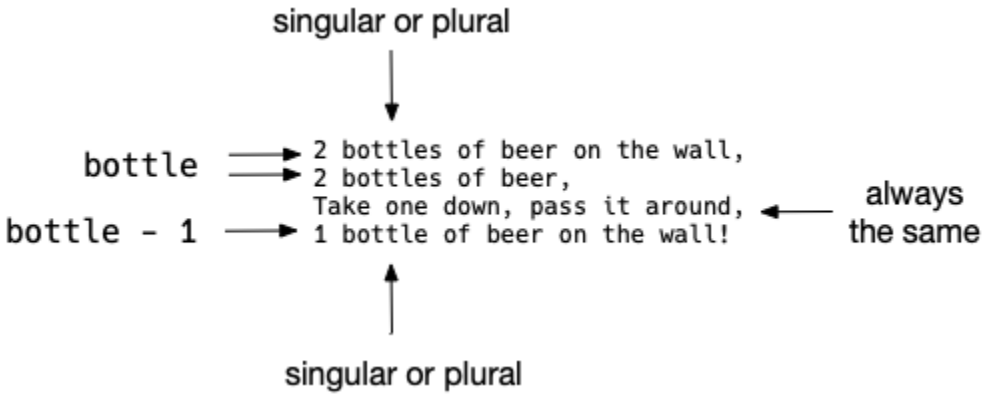
Go back and look at your `verse` definition. Think about which parts need to change — the first, second, and fourth lines. The third line is always the same. You're given a value for `bottle` which needs to be used in the first two lines along with either "bottle" or "bottles," depending on the value 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 how to write this?

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
============================
platform darwin -- Python 3.7.3,
pytest-4.6.5, py-1.8.0, pluggy-
0.12.0
rootdir:
/Users/kyclark/work/manning/tiny_
python_projects/bottles_of_beer
plugins: openfiles-0.3.2,
arraydiff-0.3, doctestplus-0.3.0,
remotedata-0.3.1, cov-2.7.1
collected 1 item

bottles.py .
[100%]

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



Using the `verse` function

At this point, you know:

1. That the `--num` value is a valid integer value greater than 0.
2. How to count from that `--num` value backwards down to 0.
3. 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 2 newlines after every verse except for the last.

You will use the regular `pytest -xv test.py` (or `make test`) to test the program at this point. In the parlance of testing, the `test.py` is an *integration test* because it checks that the program *as a whole* is working. From this point on, we'll focus more on how to write *unit* tests to check individual functions as well as *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`. I would suggest commenting 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 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?

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 the `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!

Solution

```
#!/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',  1
                        '--num',
                        metavar='int',
                        type=int,
                        default=10,
                        help='How many bottles')

    args = parser.parse_args()  2

    if args.num < 1:  3
        parser.error('--num ({}) must > 0'.format(args.num))

    return args


# --------------------------------------------------
def verse(bottle):  4
    """Sing a verse"""

    next_bottle = bottle - 1                                    5
    s1 = '' if bottle == 1 else 's'                             6
    s2 = '' if next_bottle == 1 else 's'                        7
    num_next = 'No more' if next_bottle == 0 else next_bottle   8
    return '\n'.join([                                          9
        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!',
    ])


# --------------------------------------------------
def test_verse():  10
    """Test verse"""

    one = verse(1)  11
    assert one == '\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 = verse(2)  12
    assert two == '\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!'
    ])


```

```
64  # --------------------------------------------------
65  def main():
66      """Make a jazz noise here"""
67
68      args = get_args()
69      print('\n\n'.join(map(verse, range(args.num, 0, -1)))) 13
70
71
72  # --------------------------------------------------
73  if __name__ == '__main__':
74      main()
```

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

2 Parse the command-line argument into the variable `args`.

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

4 Define a function that can create a single `verse`.

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

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

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

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

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

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

11 Test the last `verse` with the value `1`.

12 Test a `verse` with the value `2`.

13 Use a `map` function to send all the values from counting down into the `verse` function. The `map` function returns a new `list` of verses that can be joined on the string `\n\n` (two newlines).

Discussion

Defining the arguments

There isn't anything new with 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 as well as using `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.

Counting down

We know how to count down from the given `--num`, and we know we 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 making up each verse in the `for` loop, I suggested in the introduction that you 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 right in the `main` function. As you grow as a programmer, 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 smaller functions, then you know you can put them together to acheive larger tasks.

Test-Driven Development

I wanted you to add the `test_verse` function to your program to use with `pytest` to create a working `verse` function. This idea follows the principles of *Test-Driven Development* described in that book by Kent Beck (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, assume 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 the 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]

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 is 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 test 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!',
    ])
```

It's annotated above, but essentially I isolated all the parts of the return string that vary and created variables to substitute into those places. I need both the `bottle` and the `next_bottle` which I can then use to decide if there should be an "s" or not after the the "bottle" strings. I also need to figure out whether to print the next bottle as a number or the string "No more" when the `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`, and so we can move on to using it to generate the song.

Iterating through the verses

I can 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. I want an two newlines in between all the verses. I could use the `end` option to `print` to use 2 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!
```

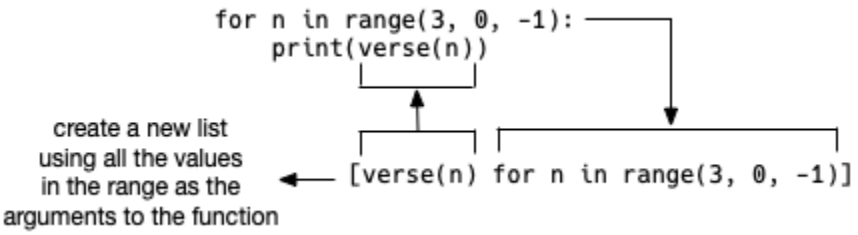


Figure 1. A `for` loop compared to a list comprehension.

I know that I could also the `str.join` method to put 2 newlines in between items in a `list`. My items are the verses, and I know I can turn a `for` loop into a list comprehension:

```
>>> print('\n\n'.join([verse(n) for n in range(3, 0, -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,
0 bottles of beer on the wall!
```

That is a fine solution; however, I would like you to start noticing a pattern we will see repeatedly. That is applying some fuction to every element of a sequence.

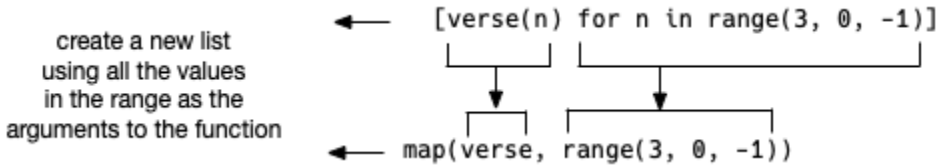


Figure 2. A list comprehension can be replaced with `map`*. They both return a new* `list`*.*

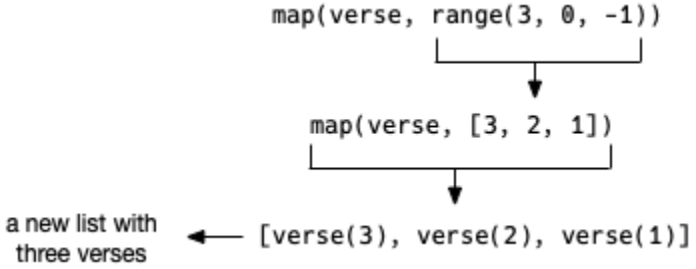
Here we have a descending `range` of numbers, and we want to send each number through the `verse` function to collect the results verses on the other end. It's like the paint booth idea in "Apples and Bananas" where the function "painted" the cars "blue" by adding the word "blue" to the front. When we want to apply a function to every element in a sequence, we might consider refactoring the code using `map`:

```
>>> print('\n\n'.join(map(verse, range(3, 0, -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!
```

Whenever I need to transform some sequence of items by some function, I like to start off by thinking about 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. I tend to favor `map` or list comprehensions because they so naturally favor a *functional* approach — that



is, one using functions. I really like writing and testing small functions, and so this approach works well for me. You may prefer to program differently, and that's fine, too.

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. Our `verse` function (which we've tested!) is the first argument, and the `range` provides the `list`. Map will make each element of the `range` an argument to the `verse` function and will return a new `list` with the results of all those function calls. Many are the `for` multi-line loops that can be better written as mapping a function over a list of arguments!

1500 other solutions

There are literally hundreds of ways to solve this problem. The website http://www.99-bottles-of-beer.net/ claims to have 1500 variations in various languages. Compare your solution to others there. Trivial as the actual program may be, it has allowed us to explore some really interesting ideas in Python, testing, and algorithms!

Review

- Test-Driven Development (TDD) is central to developing dependable, reproducible code. Tests also give you the freedom to refactor (reorganize and improve for speed or clarity) your code knowing that you can always verify your new version still works the same. As you write your code, always write tests!
- The `range` function will count backwards if the 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.

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, like by 2s or 5s.
- Add a `--reverse` flag to reverse the order of the verses, counting up instead of down.

---

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.

Last updated 2020-01-14 20:37:57 -0700