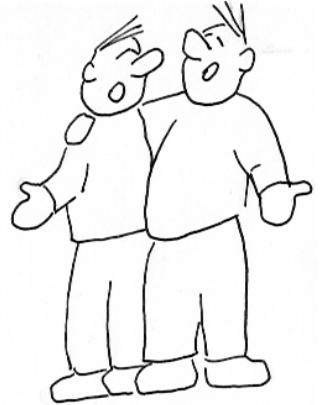


1. Twelve Days of Christmas: Algorithm design

Perhaps one of the worst songs of all time and the one that is sure to ruin my Christmas spirit is "The Twelve Days of Christmas." WILL IT EVER STOP!? AND WHAT IS WITH ALL THE BIRDS?! Still, it's pretty interesting to write an algorithm to generate the song starting from any given day because you have to count *up* as you add each verse (day) and then count *down* inside the verses (recapitulating the previous days' gifts). We'll be able to build off what you learned writing the "99 Bottles of Beer" song.



Our program will be called `twelve_days.py` and will generate the "Twelve Days of Christmas" song up to a given `-n` or `--num` argument (default `12`). Note that there should be two newlines in between each verse but only one at the end:

```
$ ./twelve_days.py -n 3
On the first day of Christmas,
My true love gave to me,
A partridge in a pear tree.

On the second day of Christmas,
My true love gave to me,
Two turtle doves,
And a partridge in a pear tree.

On the third day of Christmas,
My true love gave to me,
Three French hens,
Two turtle doves,
And a partridge in a pear tree.
```

The text will be printed to `STDOUT` unless there is a `-o` or `--outfile` argument, in which case the text should be placed inside a file by that name. Note there should be 113 lines of text for the entire song:

```
$ ./twelve_days.py -o song.txt
$ wc -l song.txt
  113 song.txt
```

In this exercise, you will:

- Create an algorithm to generate "The Twelve Days of Christmas" from any given day in the

range 1-12.

- Reverse a list
- Use the `range` function
- Write text to a file or to `STDOUT`

1.1. Writing `twelve_days.py`

As always, I suggest you start by running `new.py twelve_days.py` in the `twelve` directory or by copying the `template/template.py` file. Your program should will take two options:

1. `-n` or `--num`: an `int` with a default of 12
2. `-o` or `--outfile`: a `str` with a default of nothing/`STDOUT`

When run with the `-h` or `--help` flag, the program should print a usage:

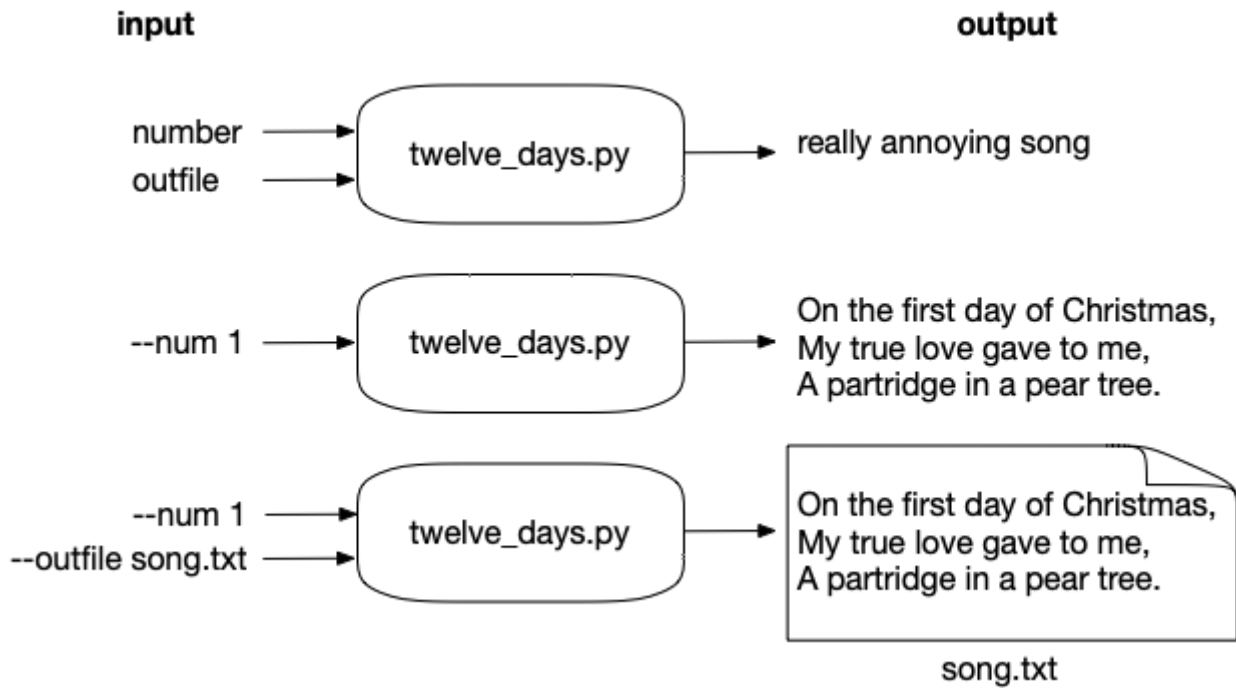
```
$ ./twelve_days.py -h
usage: twelve_days.py [-h] [-n int] [-o str]

Twelve Days of Christmas

optional arguments:
  -h, --help            show this help message and exit
  -n int, --num int      Number of days to sing (default: 12)
  -o str, --outfile str  Outfile (STDOUT) (default: )
```

Before trying to write the song, make your usage match the above. At this point, you should pass the first two tests when you run the test suite.

Here is a holly, jolly string diagram to get you in the mood for writing:



The program should complain if the `--num` value is not in the range 1-12. I suggest you check this inside the `get_args` function and use `parser.error` to halt with an error and usage:

```
$ ./twelve_days.py -n 21
usage: twelve_days.py [-h] [-n int] [-o str]
twelve_days.py: error: --num "21" must be between 1 and 12
```

Once you've handled the bad `--num`, you should pass the first three tests.

1.1.1. Counting

In the "99 Bottles of Beer" song, we need to count down from a given number. Here we need to count up to the `--num`. The `range` function will give us what we need, but we must remember to start at 1 (the default is 0) and that *the upper bound is not included*:

```
>>> num = 3
>>> list(range(1, num))
[1, 2]
```

You need to add 1 to whatever you're given for `--num`:

```
>>> list(range(1, num + 1))
[1, 2, 3]
```

Let's start by printing something like the first line of each verse:

```
>>> for day in range(1, num + 1):
...     print(f'On the {day} day of Christmas,')
...
On the 1 day of Christmas,
On the 2 day of Christmas,
On the 3 day of Christmas,
```

At this point, I'm starting to think about how we wrote "99 Bottles of Beer." There we ended up creating a `verse` function that would generate any *one* verse. Then we used `str.join` to put them all together with two newlines. I suggest we try the same approach here.

```
def verse(day):
    """Create a verse"""
    return f'On the {day} day of Christmas,'
```

Let's see how we can use it:

```
>>> for day in range(1, num + 1):
...     print(verse(day))
...
On the 1 day of Christmas,
On the 2 day of Christmas,
On the 3 day of Christmas,
```

Notice that the function will not `print` the string but will `return` the verse so that we can test it:

```
>>> assert verse(1) == 'On the 1 day of Christmas,'
```

Here's a simple (but incorrect) `test_verse` function we could start off with:

```
def test_verse():
    """ Test verse """
    assert verse(1) == 'On the 1 day of Christmas,'
    assert verse(2) == 'On the 2 day of Christmas,'
```

Add the the `verse` and `test_verse` functions to your `twelve_days.py` program and then run `pytest twelve_days.py` to verify your code.

1.1.2. Creating the ordinal value

Maybe the first thing to do is to change the numeric value to its ordinal position, that is "1" to "first," "2" to "second." You could use a dictionary like we used in "Jump The Five" to associate each `int` value 1-12 with its `str` value. That is, you might create a new `dict` called `ordinal`:

```
>>> ordinal = {} # what goes here?
```

So that you can do:

```
>>> ordinal[1]
'first'
>>> ordinal[2]
'second'
```

You could also use a **list** if you think about how you could use the each **day** in the **range** to index into a **list** of the ordinal strings.

```
>>> ordinal = [] # what goes here?
```

How would you use the **int** value of a given number to access one of the values in the **list** called **ordinal**?

Now your **verse** might look something like:

```
def verse(day):
    """Create a verse"""
    ordinal = [] # something here!
    return f'On the {ordinal[day]} of Christmas,'
```

You can update your test with your expectations:

```
def test_verse():
    """ Test verse """
    assert verse(1) == 'On the first day of Christmas,'
    assert verse(2) == 'On the second day of Christmas,'
```

Once you have this working, you should be able to replicate something like this:

```
>>> for day in range(1, num + 1):
...     print(verse(day))
...
On the day first day of Christmas
On the day second day of Christmas
On the day third day of Christmas
```

If you put the **test_verse** function inside your **twelve_days.py** program, you can verify that your **verse** function works by running **pytest twelve_days.py**. The **pytest** module will run any function that has a name starting with **test_**.

Shadowing

You might be tempted to use the variable name `ord`, and you would be allowed by Python to do this. The problem is that Python has function called `ord` that returns "the Unicode code point for a one-character string":

```
>>> ord('a')
97
```

Python will not complain if you define a variable or another function with the name `ord`:

```
>>> ord = {}
```

Such that you could do this:

```
>>> ord[1]
'first'
```

But then it overwrites the actual `ord` function, and so breaks a function call:

```
>>> ord('a')
Traceback (most recent call last):
  File "<stdin>", line 1, in <module>
TypeError: 'dict' object is not callable
```

This is called "shadowing," and it's quite dangerous. Any code in the scope of the shadowing would be affected by the change.

Tools like `pylint` can help you find problems like this in your programs. Assume the following code:

```
$ cat shadow.py
#!/usr/bin/env python3

ord = {}
print(ord('a'))
```

Here is what `pylint` has to say:

```
$ pylint shadow.py
***** Module shadow
shadow.py:3:0: W0622: Redefining built-in 'ord' (redefined-builtin)
shadow.py:1:0: C0111: Missing module docstring (missing-docstring)
shadow.py:4:6: E1102: ord is not callable (not-callable)

-----
Your code has been rated at -25.00/10
```

It's good to double-check your code with tools like `pylint`, `flake8`, and `mypy`!

1.1.3. Making the verses

Now that we have the basic structure of the program, let's focus on creating the *correct* output. We'll update our `test_verse` with the actual values for the first two verses. You can, of course, add more tests, but presumably if we can manage the first two, then we can handle all the other days:

```
1 def test_verse():
2     """Test verse"""
3
4     assert verse(1) == '\n'.join([
5         'On the first day of Christmas,', 'My true love gave to me,',
6         'A partridge in a pear tree.'
7     ])
8
9     assert verse(2) == '\n'.join([
10        'On the second day of Christmas,', 'My true love gave to me,',
11        'Two turtle doves,', 'And a partridge in a pear tree.'
12    ])
```

If you add this to your `twelve_days.py` program, you can run `pytest twelve_days.py` to see how your `verse` function is failing:

```

===== FAILURES =====
----- test_verse -----

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

>     assert verse(1) == '\n'.join([ ①
        'On the first day of Christmas,', 'My true love gave to me,',
        'A partridge in a pear tree.'
    ])
E     AssertionError: assert 'On the first...of Christmas,' == 'On the first ... a
pear tree.'
E         - On the first day of Christmas, ②
E         + On the first day of Christmas, ③
E         ?                               +
E         + My true love gave to me,
E         + A partridge in a pear tree.

twelve_days.py:88: AssertionError
===== 1 failed in 0.11 seconds =====

```

- ① The leading `>` shows this is the code that is creating an exception. We are running `verse(1)` and asking if it's equal to the expected verse.
- ② This is the text that `verse(1)` actually produced, which is only the first line of the verse.
- ③ The lines following are what was expected.

Now we need to supply the rest of the lines for each verse. They all start off the same:

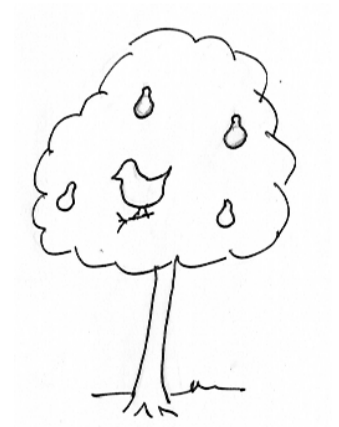
```

On the {ordinal[day]} day of Christmas,
My true love gave to me,

```


And then we need to add these gifts for each day:

1. A partridge in a pear tree
2. Two turtle doves
3. Three French hens
4. Four calling birds
5. Five gold rings
6. Six geese a laying
7. Seven swans a swimming
8. Eight maids a milking
9. Nine ladies dancing
10. Ten lords a leaping
11. Eleven pipers piping
12. Twelve drummers drumming



Note that for every day greater than 1, the last line changes "A partridge..." to "*And a* partridge in a pear tree."

Each verse needs to count backwards from the given **day**. For example, if the **day** is 3, then:

3. Three French hens
2. Two turtle doves
1. And a partridge in a pear tree

We've talked before in the "Picnic" chapter about how to reverse a **list** either with the **list.reverse** method or the **reversed** function, and we also used these ideas in "99 Bottles," so this code should not be unfamiliar:

```
>>> day = 3
>>> for n in reversed(range(1, day + 1)):
...     print(n)
...
3
2
1
```

Try to make the function return the first two lines and then the count down of the days:

```
>>> print(verse(3))
On the third day of Christmas,
My true love gave to me,
3
2
1
```

And then, instead of **3 2 1**, add the actual gifts:

```
>>> print(verse(3))
On the third day of Christmas,
My true love gave to me,
Three French hens,
Two turtle doves,
And a partridge in a pear tree.
```

If you can get that to work, then you ought to be able to pass the `test_verse` test.

1.1.4. Using the `verse` function

Once you have that working, think about a final structure that calls your `verse`. It could be a `for` loop:

```
verses = []
for day in range(1, args.num + 1):
    verses.append(verse(day))
```

A list comprehension:

```
verses = [verse(day) for day in range(1, args.num + 1)]
```

Or a `map`:

```
verses = map(verse, range(1, args.num + 1))
```



1.1.5. Printing

Once you have all the verses, you can use the `str.join` method to print the output. The default is to print this to "standard out" (`STDOUT`), but the program will also take an optional `--outfile` that names a file to write the output. I suggest you go back to the "Howler" solution for how to handle that. You ought to be able to copy the relevant code exactly to determine the proper file handle. If you do need to write to an `--outfile`, be sure to `open` it with the proper permissions. Again, see the "Howler" exercise for a review of that.

1.1.6. Time to write

It's not at all mandatory that you solve the problem the way that I describe. The "correct" solution is the one that you write and understand which passes the test suite. It's fine if you like the idea of creating a functions for `verse` and using the provided test. It's fine if you want to go another way, but do try to think of writing small functions *and tests* to solve small parts of your problem and combining them to solve the larger problem.

If you need more than one sitting or even several days, take your time. Sometimes a good walk or a nap can do wonders for solving problems. Don't neglect your hammock or a nice cup of tea.

1.2. Solution

```
1 #!/usr/bin/env python3
2 """Twelve Days of Christmas"""
3
4 import argparse
5 import sys
6
7
8 # -----
9 def get_args():
10     """Get command-line arguments"""
11
12     parser = argparse.ArgumentParser(
13         description='Twelve Days of Christmas',
14         formatter_class=argparse.ArgumentDefaultsHelpFormatter)
15
16     parser.add_argument('-n', ①
17                         '--num',
18                         help='Number of days to sing',
19                         metavar='int',
20                         type=int,
21                         default=12)
22
23     parser.add_argument('-o', ②
24                         '--outfile',
25                         help='Outfile (STDOUT)',
26                         metavar='str',
27                         type=str,
28                         default='')
29
30     args = parser.parse_args() ③
31
32     if args.num not in range(1, 13): ④
33         parser.error(f'--num "{args.num}" must be between 1 and 12') ⑤
34
35     return args
36
37
38 # -----
39 def main():
40     """Make a jazz noise here"""
41
42     args = get_args() ⑥
43     out_fh = open(args.outfile, 'wt') if args.outfile else sys.stdout ⑦
44     out_fh.write('\n\n'.join(map(verse, range(1, args.num + 1))) + '\n') ⑧
45     out_fh.close() ⑨
46
47
48 # -----
```

```

49 def verse(day): ⑩
50     """Create a verse"""
51
52     ordinal = [ ⑪
53         'first', 'second', 'third', 'fourth', 'fifth', 'sixth', 'seventh',
54         'eighth', 'ninth', 'tenth', 'eleventh', 'twelfth'
55     ]
56
57     gifts = [ ⑫
58         'A partridge in a pear tree.',
59         'Two turtle doves,',
60         'Three French hens,',
61         'Four calling birds,',
62         'Five gold rings,',
63         'Six geese a laying,',
64         'Seven swans a swimming,',
65         'Eight maids a milking,',
66         'Nine ladies dancing,',
67         'Ten lords a leaping,',
68         'Eleven pipers piping,',
69         'Twelve drummers drumming,',
70     ]
71
72     lines = [ ⑬
73         f'On the {ordinal[day - 1]} day of Christmas,',
74         'My true love gave to me,'
75     ]
76
77     lines.extend(reversed(gifts[:day])) ⑭
78
79     if day > 1: ⑮
80         lines[-1] = 'And ' + lines[-1].lower() ⑯
81
82     return '\n'.join(lines) ⑰
83
84
85 # -----
86 def test_verse(): ⑱
87     """Test verse"""
88
89     assert verse(1) == '\n'.join([
90         'On the first day of Christmas,', 'My true love gave to me,',
91         'A partridge in a pear tree.'
92     ])
93
94     assert verse(2) == '\n'.join([
95         'On the second day of Christmas,', 'My true love gave to me,',
96         'Two turtle doves,', 'And a partridge in a pear tree.'
97     ])
98
99

```

```

100 # -----
101 if __name__ == '__main__':
102     main()

```

- ① The `--num` option is an `int` with a default of 12.
- ② The `--outfile` option is a `str` with a default of either `None` or the empty string.
- ③ Capture the results of parsing the command-line arguments (`parser.parse_args`) into the `args` variable.
- ④ Check that the given `args.num` is in the allowed range (1-12, inclusive).
- ⑤ If `args.num` is invalid, use `parser.error` to print a short usage and the error message to `STDERR` and exit the program with an error value. Note that the error message includes the bad value for the user and explicitly states that a good value should be in the range 1-12.
- ⑥ Get the command-line arguments. Remember that all argument validation happens inside `get_args`. If this call succeeds, then we have good arguments from the user.
- ⑦ The output file handle (`out_fh`) is either the result of opening the given `args.outfile` for writing or is `sys.stdout`.
- ⑧ Use `map` to generate all the verses using the `range` function to generate a list of the days up to the given number. The verses are joined on two newlines. We need to included a final (single) newline because the `fh.write` method does not automatically add a newline (unlike `print`, which does).
- ⑨ Close the output file handle. This is also the last statement when the program runs.
- ⑩ Define a function to create any one verse from a given number 1-12.
- ⑪ The `ordinal` values for the numbers are a `list` of `str` values.
- ⑫ The `gifts` for the days is a `list` of `str` values.
- ⑬ The `lines` of each verse start off the same, substituting in the ordinal value of the given `day`.
- ⑭ I use the `list.extend` method to add the `gifts`, which are a slice from the given `day` and then `reversed`.
- ⑮ Check if this is for a `day` greater than 1.
- ⑯ Change the last of the `lines` to add "And " to the beginning appended to the lowercased version of the line.
- ⑰ Return the `lines` joined on the newline.
- ⑱ The unit test for the `verse` function.

1.3. Discussion

1.3.1. Defining the arguments

Nothing in the `get_args` is new, so we'll throw a sidelong, cursory glance. The `--num` option is an `int` value with a default value of 12. The `--outfile` option is a `str` with a default of either `None` or the empty string because both will evalute to `False` when we check later. If the `--num` value is not in `range(1, 13)` (remembering that 13 is not included, so just the numbers 1-12), I use `parser.error` to

show an error message and the short usage and exit with a non-zero exit value to indicate a failure. Notice that the error message provides feedback in showing the bad value and explaining exactly what is allowed.

1.3.2. Defining the output handle

If the user provides a value for `--outfile`, we'll use that as the name of a file to open. If they do not, we want to use `sys.stdout`. Because this is a binary decision (there are two options), I prefer to use an `if` expression. I always name variables that are file handles with `fh` somewhere in the name, so here the output file handle is `out_fh`.

```
out_fh = open(args.outfile, 'wt') if args.outfile else sys.stdout
```

Note that, if we `open` the output file, we must use `'wt'` for "write" and "text" modes. This is exactly the same code as from "Howler," so I would recommend you review that discussion.

1.3.3. Making one verse

I chose to make a function called `verse` to create any one verse given an `int` value of the `day`:

```
1 def verse(day):
2     """Create a verse"""
```

ordinal = [index	day
'first'	0	1
'second'	1	2
'third'	2	3
'fourth'	3	4
'fifth'	4	5
'sixth'	5	6
'seventh'	6	7
'eighth'	7	8
'ninth'	8	9
'tenth'	9	10
'eleventh'	10	11
'twelfth'	11	12

]

I decided to use a `list` to represent the `ordinal` value of the `day`:

```
1 ordinal = [
2     'first', 'second', 'third', 'fourth', 'fifth', 'sixth', 'seventh',
3     'eighth', 'ninth', 'tenth', 'eleventh', 'twelfth'
4 ]
```

Since the `day` is based on counting from 1 but Python lists start from 0, I have to subtract 1:

```
>>> day = 3
>>> ordinal[day - 1]
'third'
```

I could have just as easily used a `dict`:

```
1 ordinal = {
2     1: 'first', 2: 'second', 3: 'third', 4: 'fourth',
3     5: 'fifth', 6: 'sixth', 7: 'seventh', 8: 'eighth',
4     9: 'ninth', 10: 'tenth', 11: 'eleventh', 12: 'twelfth',
5 }
```

Then you don't have to subtract 1:

```
>>> ordinal[3]
'third'
```

Whatever works for you. I also used a `list` for the `gifts`:


```
1 gifts = [
2     'A partridge in a pear tree.',
3     'Two turtle doves,',
4     'Three French hens,',
5     'Four calling birds,',
6     'Five gold rings,',
7     'Six geese a laying,',
8     'Seven swans a swimming,',
9     'Eight maids a milking,',
10    'Nine ladies dancing,',
11    'Ten lords a leaping,',
12    'Eleven pipers piping,',
13    'Twelve drummers drumming,',
14 ]
```



```

                                gifts[:3]
['A partridge in a pear tree.', 0
 'Two turtle doves,',          1
 'Three French hens,',         2
 'Four calling birds,',        3
 'Five gold rings,',           4
 'Six geese a laying,',         5
 'Seven swans a swimming,',     6
 'Eight maids a milking,',      7
 'Nine ladies dancing,',        8
 'Ten lords a leaping,',        9
 'Eleven pipers piping,',       10
 'Twelve drummers drumming,']  11

```



This makes a bit more sense as I can use a list slice to get the **gifts** for a given **day**:

```

>>> gifts[:3]
['A partridge in a pear tree.',
 'Two turtle doves,',
 'Three French hens,']

```

But I want them in reverse order. The **reversed** function is lazy, so I need to use the **list** function to coerce the values in the REPL:

```

>>> list(reversed(gifts[:3]))
['Three French hens,',
 'Two turtle doves,',
 'A partridge in a pear tree.']

```

The first two lines of any verse are the same, substituting in the **ordinal** value for the **day**.

```

1 lines = [
2     f'On the {ordinal[day - 1]} day of Christmas,',
3     'My true love gave to me,',
4 ]

```

I need to put these two **lines** together with the **gifts**. Since each verse is made of some number of lines, I think it will make sense to use a **list** to represent the entire verse. So I need to add the **gifts** to the **lines**, and I can use the **list.extend** method to do that:

```

>>> lines.extend(reversed(gifts[:day]))

```

And now there are 5 **lines**:

```
>>> lines
['On the third day of Christmas,',
 'My true love gave to me,',
 'Three French hens,',
 'Two turtle doves,',
 'A partridge in a pear tree.']
>>> assert len(lines) == 5
```

Note that I cannot use the `list.append` method. It's easy to confuse these two methods. The `list.extend` method takes another `list` as the argument, expands it, and adds all of the individual elements to the original `list`. The `list.append` method is meant to add one element to the `list`, so, if you give it a `list`, it will tack that entire `list` on to the end of the original list.

Here the `reversed` iterator will be added the end of `lines` such that it would have three elements rather than the desired five:

```
>>> lines.append(reversed(gifts[:day]))
>>> lines
['On the third day of Christmas,',
 'My true love gave to me,',
 <list_reverseiterator object at 0x105bc8588>]
```

Maybe you're thinking you could coerce the `reversed` with a `list`? Thinking you are, young Jedi, but, alas, that will still add a new `list` to the end:

```
>>> lines.append(list(reversed(gifts[:day])))
>>> lines
['On the third day of Christmas,',
 'My true love gave to me,',
 ['Three French hens,', 'Two turtle doves,', 'A partridge in a pear tree.']]
```

And we still have 3 `lines` rather than 5:

```
>>> len(lines)
3
```

If the `day` is greater than 1, I need to change the last line to say "And a" instead of "A":

```
if day > 1:
    lines[-1] = 'And ' + lines[-1].lower()
```

Note that this is another good reason to represent the `lines` as a `list` because the elements of a `list` are *mutable*. I could have represented the `lines` as a `str`, but strings are *immutable*, so it would be much harder to change the last line.

I want to return a single `str` value from the function, so I join the `lines` on a newline:

```
>>> print('\n'.join(lines))
On the third day of Christmas,
My true love gave to me,
Three French hens,
Two turtle doves,
A partridge in a pear tree.
```

My function returns the joined `lines` and will pass the `test_verse` function I provided.

1.3.4. Generating the verses

Given the `verse` function, I can create all the needed verses by iterating from 1 to the given `--num`. I could collect them in `list` of `verses`:

```
1 day = 3
2 verses = []
3 for n in range(1, day + 1):
4     verses.append(verse(n))
```

I can check that I have the right number of verses:

```
>>> assert len(verses) == day
```

Whenever you see this pattern of creating an empty `str` or `list` and then using a `for` loop to add to it, consider instead using a list comprehension:

```
>>> verses = [verse(n) for n in range(1, day + 1)]
>>> assert len(verses) == day
```

I personally prefer using `map` over list comprehensions. I will use the `list` function to coerce the lazy `map` function here, but it's not necessary in the program code:

```
>>> verses = list(map(verse, range(1, day + 1)))
>>> assert len(verses) == day
```

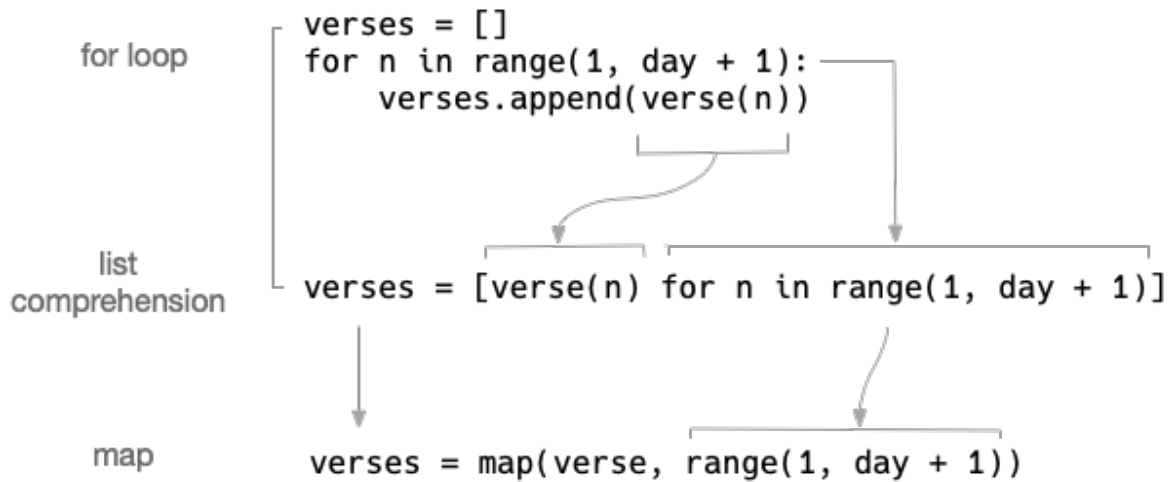


Figure 13. 1. Building a `list` using an `for` loop, a list comprehension, and `map`.

All of these methods will produce the correct number of verses. Choose whichever one makes the most sense to you.

1.3.5. Printing the verses

Just like with the "99 Bottles of Beer," I want to `print` them with two newlines in between. The `str.join` method is a good choice:

```
>>> print('\n\n'.join(verses))
On the first day of Christmas,
My true love gave to me,
A partridge in a pear tree.

On the second day of Christmas,
My true love gave to me,
Two turtle doves,
And a partridge in a pear tree.

On the third day of Christmas,
My true love gave to me,
Three French hens,
Two turtle doves,
And a partridge in a pear tree.
```

You can use the `print` method with the optional `file` argument to put the text into an open filehandle. The `out_fh` will be either the file indicated by the user or `sys.stdout`:

```
1 print('\n\n'.join(map(verse, range(1, args.num + 1))), file=out_fh)
```

Or you can use the `out_fh.write` method, but you need to remember to add the trailing newline that `print` adds for you:

```
1 out_fh.write('\n\n'.join(map(verse, range(1, args.num + 1))) + '\n')
```

Finally, you need to close the file handle:

```
1 out_fh.close()
```

There are dozens to hundreds of ways to write this algorithm just as there are for "99 Bottles of Beer." If you came up with an entirely different approach which passed the test, that's terrific! Please share it with me. I wanted to stress the idea of how to write, test, and use a single `verse` function, but I'd love to see other approaches!



1.4. Review

- There are many ways to encode algorithms to perform repetitive tasks. In my version, I wrote and tested a function to handle one task and then mapped a range of input values over that.
- The `range` function will return `int` values between a given start and stop value, the latter of which is not included.
- You can use the `reversed` function to reverse the values returned by `range`.
- To `open` a file for writing, you must include the `'w'` flag. We were writing text (not binary data), so we also included `'t'` to make the open mode `'wt'`.
- The `sys.stdout` file handle is always open and available for writing.
- Modeling the `gifts` as a `list` allowed me to use a list slice to get all the gifts for a given day. I used the `reversed` function to put them into the right order for the song.
- I modeled the `lines` as a `list` because a `list` is mutable which I needed in order to change the last line when the day is greater than 1.
- Shadowing a variable or function is reusing an existing variable or function name. If, for instance, you create a variable with the name of an existing function, then the function is effectively hidden because of the shadow. Avoid shadowing by using tools like `pylint` to find these and many other common coding problems.

1.5. Going Further

- Install the `emoji` module (<https://pypi.org/project/emoji/>) and print various emojis for the gifts rather than text. For instance, you could use `':bird:'` to print 🐦 for every "bird" like a hen or

dove. I also used `':man:'`, `':woman:'`, and `':drum:'`, but you can use whatever you like:

```
On the twelfth day of Christmas,  
My true love gave to me,  
Twelve ♀s drumming,  
Eleven ♀s piping,  
Ten ♀s a leaping,  
Nine ♀s dancing,  
Eight ♀s a milking,  
Seven ♀s a swimming,  
Six ♀s a laying,  
Five gold ♀s,  
Four calling ♀s,  
Three French ♀s,  
Two turtle ♀s,  
And a ♂ in a pear tree.
```

- Write the "Wheels on the Bus" song. Play "golf" with your code. In the actual game of golf, you want to score as low as possible. Each time you swing the club is a stroke you count, so when you "golf" your code, you try to use the fewest number of characters. For fun, search the Internet for "Python golf Tetris" to see how other programmers can implement a working game of Tetris in astoundingly small programs.