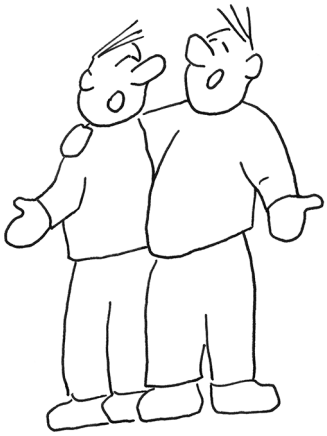
13 Twelve Days of Christmas: Algorithm design



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). You'll be able to build on what you learned writing the program for "99 Bottles of Beer."

Our program in this chapter will be called twelve_days.py, and it will generate the "Twelve Days of Christmas" song up to a given day, specified by the `-n` or `--num` argument (default `12`). Note that there should be two newlines between verses 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 an `-o` or `--outfile` argument, in which case the text should be placed inside a file with the given name. Note that 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`

13.1 Writing twelve_days.py

As always, I suggest you create your program by running new.py or by copying the template/template.py file. This one must be called twelve_days.py and live in the 13_twelve_days directory.

Your program should take two options:

- `-n` or `--num`—An `int` with a default of 12
- `-o` or `--outfile`—An optional filename for writing the output

For the second option, you can go back to chapter 5 to see how we handled this in the Howler solution. That program writes its blistering output to the given filename if one is supplied, and otherwise writes to `sys.stdout`. For this program, I suggest you declare the `--outfile` using `type=argparse.FileType('wt')` to indicate that `argparse` will require an argument to name a *writable text* file. If the user supplies a valid argument, `args.outfile` will be an *open, writable file handle*. If you also use a default of `sys.stdout`, you'll have quickly handled both options of writing to a text file or `STDOUT`!

The only downside to this approach is that the usage statement for the program looks a little funny in describing the default for the `--outfile` parameter:

```
$ ./twelve_days.py -h
usage: twelve_days.py [-h] [-n days] [-o FILE]

Twelve Days of Christmas

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

```
  -n days, --num days   Number of days to sing (default: 12)
  -o FILE, --outfile FILE
                        Outfile (default: <_io.TextIOWrapper name='<stdout>'
                        mode='w' encoding='utf-8'>)
```

Once you've completed the usage, your program should pass the first two tests.

Figure 13.1 shows a holly, jolly string diagram to get you in the mood for writing the rest of the program.

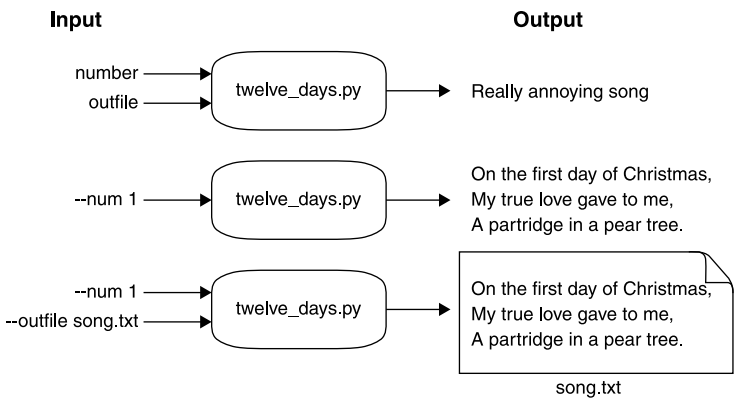


Figure 13.1 The twelve_days.py program takes options for which day to start on and an output file to write.

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 message:

```
$ ./twelve_days.py -n 21
usage: twelve_days.py [-h] [-n days] [-o FILE]
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.

13.1.1 Counting

In the "99 Bottles of Beer" song, we needed to count down from a given number. Here we need to count up to `--num` and then count back down through the gifts. The `range()` function will give us what we need, but we must remember to start at `1` because we don't start singing "On the zeroth day of Christmas." Keep in mind that the upper bound is not included:

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

You'll 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, so I'll move the code inside the `for` loop into its own function:

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

Let's see how we can use this `verse()` function:

```
>>> 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,
```

Here's a simple `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,'
```

This is incorrect, of course, because it should say "On the *first* day" or the "*second* day," not "1 day" or "2 day." Still, it's a place to start. Add the `verse()` and `test_verse()` functions to your twelve_days.py program, and then run `pytest twelve_days.py` to verify this much works.

13.1.2 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?
```

Then you could do this:

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

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

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

Your `verse()` function might look something like this now:

```
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 a 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 that 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 you have 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 and Flake8!

13.1.3 Making the verses

Now that we have the basic structure of the program, let's focus on creating the *correct* output. We'll update `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 days, we can handle all the other days:

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

    assert verse(1) == '\n'.join([
        'On the first day of Christmas,', 'My true love gave to me,',
        'A partridge in a pear tree.'
    ])

    assert verse(2) == '\n'.join([
        'On the second day of Christmas,', 'My true love gave to me,',
        'Two turtle doves,', 'And a partridge in a pear tree.'
    ])
```

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([
            '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,
E         + On the first day of Christmas,
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 ==========================
```

The leading > shows that this is the code that is creating an exception. We are running verse(1) and asking if it's equal to the expected verse.

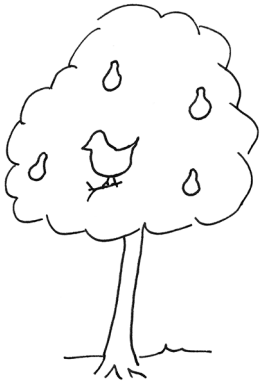
This is the text that verse(1) actually produced, which is only the first line of the verse.

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,
```

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 the verse lists

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

We talked in chapter 3 about how you can reverse a `list`, either with the `list.reverse()` method or the `reversed()` function. We also used these ideas in chapter 11 to get the bottles of beer off the wall, 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 countdown of the days:

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

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, you ought to be able to pass the `test_verse()` test.

13.1.4 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))
```

Since we're trying to create a `list` of the verses, a list comprehension is a better choice:

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

Or it could be a `map()`:

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

13.1.5 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 to. You can copy exactly what we did in chapter 5, but it's really worth your time to learn how to declare output files using `type=argparse.FileType('wt')`. You can even set the default to `sys.stdout` so that you'll never have to `open()` the output file yourself!

13.1.6 Time to write

It's not at all mandatory that you solve the problem the way that I describe. The "correct" solution is one that you write and understand and that passes the test suite. It's fine if you like the idea of creating a function for `verse()` and using the provided test. It's also 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 then combining them to solve the larger problem.

If you need more than one sitting or even several days to pass the tests, take your time. Sometimes a good walk or a nap can do wonders for solving problems. Don't neglect your hammock[1] or a nice cup of tea.

[1] Search the internet for the talk "Hammock Driven Development" by Rich Hickey, the creator of the Clojure language.

13.2 Solution

A person would receive almost 200 birds in this song! Anyway, here is a solution that uses `map()`. After that you'll see versions that use `for` and list comprehensions.

```
#!/usr/bin/env python3
"""Twelve Days of Christmas"""

import argparse
import sys


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

    parser = argparse.ArgumentParser(
        description='Twelve Days of Christmas',
        formatter_class=argparse.ArgumentDefaultsHelpFormatter)

    parser.add_argument('-n',
                        '--num',
                        help='Number of days to sing',
                        metavar='days',
                        type=int,
                        default=12)

    parser.add_argument('-o',
                        '--outfile',
                        help='Outfile',
                        metavar='FILE',
                        type=argparse.FileType('wt'),
                        default=sys.stdout)

    args = parser.parse_args()

    if args.num not in range(1, 13):
        parser.error(f'--num "{args.num}" must be between 1 and 12')

    return args


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

    args = get_args()
    verses = map(verse, range(1, args.num + 1))
    print('\n\n'.join(verses), file=args.outfile)


# --------------------------------------------------
def verse(day):
    """Create a verse"""
```

If args.num is invalid, use parser.error() to print a short usage statement 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.

The --num option is an int with a default of 12.

The --outfile option is a type=argparse.FileType('wt') with a default of sys.stdout. If the user supplies a value, it must be the name of a writable file, in which case argparse will open the file for writing.

Check that the given args.num is in the allowed range1–12, inclusive.

Capture the results of parsing the command-line arguments into the args variable.

Generate the verses for the given args.num of days.

Get the command-line arguments. Remember that all argument validation happens inside get_args(). If this call succeeds, we have good arguments from the user.

Join the verses on two newlines and print to args.outfile, which is an open file handle, or sys.stdout.

Define a function to create any one verse from a given number.

The ordinal values is a list of str values.

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

The gifts for the days is a list of str values.

```
    gifts = [
        'A partridge in a pear tree.',
        'Two turtle doves,',
        'Three French hens,',
        'Four calling birds,',
        'Five gold rings,',
        'Six geese a laying,',
        'Seven swans a swimming,',
        'Eight maids a milking,',
        'Nine ladies dancing,',
        'Ten lords a leaping,',
        'Eleven pipers piping,',
        'Twelve drummers drumming,',
    ]
```

The lines of each verse start off the same, substituting in the ordinal value of the given day.

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

Use the list.extend() method to add the gifts, which are a slice from the given day and then reversed().

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

Check if this is for a day greater than 1.

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

Change the last of the lines to add "And " to the beginning, appended to the lowercased version of the line.

Return the lines joined on the newline.

```
    return '\n'.join(lines)


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

The unit test for the verse() function

```
    assert verse(1) == '\n'.join([
        'On the first day of Christmas,', 'My true love gave to me,',
        'A partridge in a pear tree.'
    ])

    assert verse(2) == '\n'.join([
        'On the second day of Christmas,', 'My true love gave to me,',
        'Two turtle doves,', 'And a partridge in a pear tree.'
    ])


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

13.3 Discussion

Not much in `get_args()` is new, so we'll throw it a sidelong, cursory glance. The `--num` option is an `int` value with a default value of `12`, and we use `parser.error()` to halt the program if the user provides a bad value. The `--outfile` option is a bit different, though, as we're declaring it with `type=argparse.FileType('wt')` to indicate the value must be a writable file. This means the value we get from `argparse` will be an open, writable file. We set the default to `sys.stdout`, which is also an open, writable file, so we've handled the two output options entirely through `argparse`, which is a real time saver!

13.3.1 Making one verse

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

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

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

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

Since the `day` is based on counting from 1, but Python lists start from 0 (see figure 13.2), I have to subtract 1:

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

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
]		

Figure 13.2 Our days start counting from 1, but Python indexes from 0.

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

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

In this case you don't have to subtract 1. Whatever works for you:

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

I also used a `list` for the `gifts`:

```
gifts = [
    'A partridge in a pear tree.',
    'Two turtle doves,',
    'Three French hens,',
    'Four calling birds,',
    'Five gold rings,',
    'Six geese a laying,',
    'Seven swans a swimming,',
    'Eight maids a milking,',
    'Nine ladies dancing,',
    'Ten lords a leaping,',
    'Eleven pipers piping,',
    'Twelve drummers drumming,',
]
```

This makes a bit more sense, as I can use a list slice to get the `gifts` for a given `day` (see figure 13.3):

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

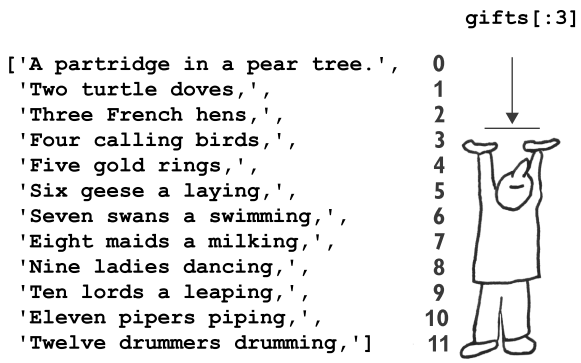


Figure 13.3 The gifts are listed by their days in ascending order.

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

```
>>> 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`:

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

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.

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]))
```

Now there are five `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 it with the `list.extend()` method, which takes another `list` as its argument, expands it, and adds all of the individual elements to the original `list`. The `list.append()` method is meant to add *just one* element to a `list`, so if you give it a `list`, it will tack that entire `list` onto the end of the original list!

Here the `reversed()` iterator will be added to 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 `reversed()` with the `list()` function? 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 three `lines` rather than five:

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

If `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.

13.3.2 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 a `list` of `verses`:

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

I can test 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. See figure 13.4 to review how the three methods fit together. I need to use the `list()` function to coerce the lazy `map()` function in the REPL, but it's not necessary in the program code:

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

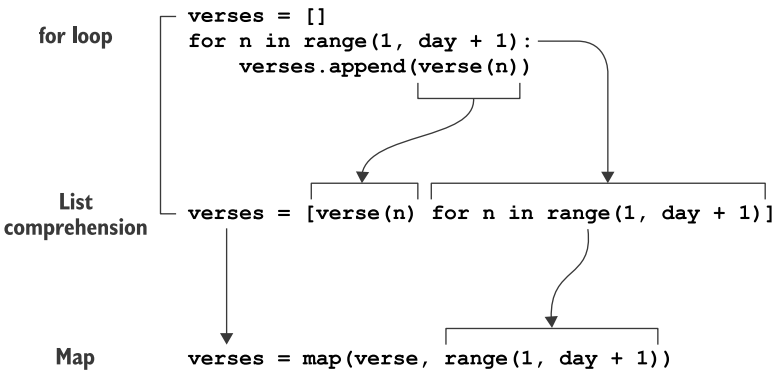


Figure 13.4 Building a `list` using a `for` loop, a list comprehension, and `map()`.

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

13.3.3 Printing the verses

Just like with "99 Bottles of Beer" in chapter 11, I want to `print()` the verses 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()` function with the optional `file` argument to put the text into an open file handle. The `args.outfile` value will be either the file indicated by the user or `sys.stdout`:

```
print('\n\n'.join(verses), file=args.outfile)
```

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

```
args.outfile.write('\n\n'.join(verses) + '\n')
```

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 that 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!

13.4 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.
```

Summary

- 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 given start and stop values, the latter of which is not included.
- You can use the `reversed()` function to reverse the values returned by `range()`.

- If you use `type=argparse.FileType('wt')` to define an argument with `argparse`, you get a file handle that is open for writing text.
- The `sys.stdout` file handle is always open and available for writing.
- Modeling `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 `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, that function is effectively hidden because of the shadow. Avoid shadowing by using tools like Pylint to find these and many other common coding problems.