# 3 Going on a picnic: Working with lists

Writing code makes me hungry! Let's write a program to list some tasty foods we'd like to eat.

So far we've worked with single variables, like a name to say "hello" to or a nautical-themed object to point out. In this program, we want to keep track of one or more foods that we will store in a `list`, a variable that can hold any number of items. We use lists all the time in real life. Maybe it's your top-five favorite songs, your birthday wish list, or a bucket list of the best types of buckets.

In this chapter, we're going on a picnic, and we want to print a list of items to bring along. You will learn to

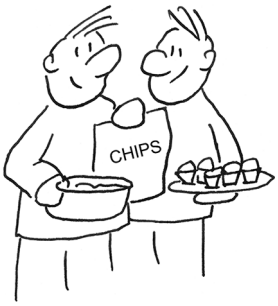
- Write a program that accepts multiple positional arguments
- Use `if`, `elif`, and `else` to handle conditional branching with three or more options
- Find and alter items in a list
- Sort and reverse lists
- Format a list into a new string

The items for the list will be passed as positional arguments. When there is only one item, you'll print that:

```
$ ./picnic.py salad
You are bringing salad.
```


What? Who just brings salad on a picnic? When there are two items, you'll print "and" between them:

```
$ ./picnic.py salad chips
You are bringing salad and chips.
```




Hmm, chips. That's an improvement. When there are three or more items, you'll separate the items with commas:

```
$ ./picnic.py salad chips cupcakes
You are bringing salad, chips, and cupcakes.
```

There's one other twist. The program will also need to accept a `--sorted` argument that will require you to sort the items before you print them. We'll deal with that in a bit.

So, your Python program must do the following:

- Store one or more positional arguments in a `list`
- Count the number of arguments
- Possibly sort the items
- Use the `list` to print a new a string that formats the arguments according to how many items there are

How should we begin?

## 3.1 Starting the program

I will always recommend you start programming by running new.py or by copying template/template.py to the program name. This time the program should be called picnic.py, and you need to create it in the 03_picnic directory.

You can do this using the new.py program from the top level of your repository:

```
$ bin/new.py 03_picnic/picnic.py
Done, see new script "03_picnic/picnic.py."
```

Now go into the 03_picnic directory and run `make test` or `pytest -xv test.py`. You should pass the first two tests (program exists, program creates usage) and fail the third:

```
test.py::test_exists PASSED                                          [ 14%]
test.py::test_usage PASSED                                           [ 28%]
test.py::test_one FAILED                                             [ 42%]
```

The rest of the output complains that the test expected "You are bringing chips" but got something else:

```
================================== FAILURES ==================================
_________________________________ test_one __________________________________

    def test_one():
        """one item"""

        out = getoutput(f'{prg} chips')
>       assert out.strip() == 'You are bringing chips.'
E       assert 'str_arg = ""...nal = "chips"' == 'You are bringing chips.'
E         + You are bringing chips.
E         - str_arg = ""
E         - int_arg = "0"
E         - file_arg = ""
E         - flag_arg = "False"
E         - positional = "chips"

test.py:31: AssertionError
====================== 1 failed, 2 passed in 0.56 seconds ======================
```

**The program is being run with the argument "chips."**

**The line starting with a + sign shows what was expected.**

**The lines starting with the - sign show what was returned by the program.**

**This line is causing the error. The output is tested to see if it is equal (==) to the string "You are bringing chips."**

Let's run the program with the argument "chips" and see what it gets:

```
$ ./picnic.py chips
str_arg = ""
int_arg = "0"
file_arg = ""
flag_arg = "False"
positional = "chips"
```

Right, that's not correct at all! Remember, the template doesn't yet have the *correct* arguments, just some examples, so the first thing we need to do is fix the `get_args()` function. Your program should print a usage statement like the following if given *no arguments*:

```
$ ./picnic.py
usage: picnic.py [-h] [-s] str [str ...]
picnic.py: error: the following arguments are required: str
```

And here is the usage for the `-h` or `--help` flags:

```
$ ./picnic.py -h
usage: picnic.py [-h] [-s] str [str ...]

Picnic game

positional arguments:
  str           Item(s) to bring
```

```
optional arguments:
  -h, --help    show this help message and exit
  -s, --sorted  Sort the items (default: False)
```

We need one or more positional arguments and an optional flag called `--sorted`. Modify your `get_args()` until it produces the preceding output.

Note that there should be one or more of the `item` parameter, so you should define it with `nargs='+'`. Refer to section A.4.5 in the appendix for details.

## 3.2 Writing picnic.py

Figure 3.1 shows a tasty diagram of the inputs and outputs for the picnic.py program we'll write.

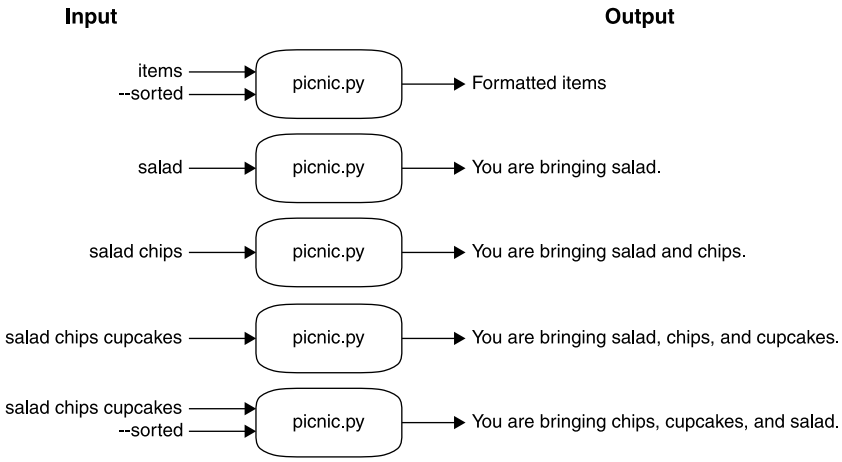



**Figure 3.1 A string diagram of the picnic program showing the various inputs and outputs the program will handle**

The program should accept one or more positional arguments for the items to bring on a picnic as well as an `-s` or `--sorted` *flag* to indicate whether or not to sort the items. The output will be "You are bringing" followed by the list of items formatted according to the following rules:

- If there's one item, state the item:

  ```
  $ ./picnic.py chips
  You are bringing chips.
  ```

- If there are two items, put "and" in between the items. Note that "potato chips" is just *one string* that happens to contain *two words*. If you leave out the quotes,

there would be three arguments to the program. It doesn't matter here whether you use single or double quotes:

```
$ ./picnic.py "potato chips" salad
You are bringing potato chips and salad.
```

- If there are three or more items, place a comma and space between the items and the word "and" before the final element. Don't forget the comma before the "and" (sometimes called the "Oxford comma") because your author was an English lit major and, while I may have finally stopped using two spaces after the end of a sentence, you can pry the Oxford comma from my cold, dead hands:

```
$ ./picnic.py "potato chips" salad soda cupcakes
You are bringing potato chips, salad, soda, and cupcakes.
```

Be sure to sort the items if the `-s` or `--sorted` flag is specified:

```
$ ./picnic.py --sorted salad soda cupcakes
You are bringing cupcakes, salad, and soda.
```

To figure out how many items we have, how to sort and slice them, and how to format the output string, we need to talk about the `list` type in Python.

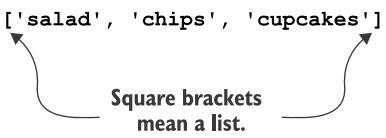
## 3.3 Introducing lists

It's time to learn how to define positional arguments so that they are available as a `list`. That is, if we run the program like this,

```
$ ./picnic.py salad chips cupcakes
```

the arguments `salad chips cupcakes` will be available as a `list` of strings inside the program. If you `print()` a `list` in Python, you'll see something like this:

```
['salad', 'chips', 'cupcakes']
```




The square brackets tell us this is a `list`, and the quotes around the elements tell us they are strings. Note that the items are shown in the same order as they were provided on the command line. Lists always keep their order!

Let's go into the REPL and create a variable called `items` to hold some scrumptious victuals to bring on our picnic. I really want you to type these commands yourself, whether in the `python3` REPL or IPython or a Jupyter Notebook. It's very important to interact in real time with the language.

To create a new, empty `list`, you can use the `list()` function:

```
>>> items = list()
```

Or you can use empty square brackets:

```
>>> items = []
```

Check what Python says for the `type()`. Yep, it's a `list`:

```
>>> type(items)
<class 'list'>
```

One of the first things we need to know is how many `items` we have for our picnic. Just as with a `str`, we can use `len()` (length) to get the number of elements in `items`:

```
>>> len(items)
0
```

The length of an empty `list` is `0`.

### 3.3.1 Adding one element to a list

An empty `list` is not very useful. Let's see how we can add new items. We used `help(str)` in the last chapter to read documentation about the string methods—the functions that belong to every `str` in Python. Here I want you to use `help(list)` to learn about the `list` methods:

```
>>> help(list)
```

Remember that pressing the spacebar or *F* key (or Ctrl-F) will take you forward, and pressing *B* (or Ctrl-B) will take you back. Pressing the / key will let you search for a string.

You'll see lots of "double-under" methods, like `__len__`. Skip over those, and the first method is `list.append()`, which we can use to add items to the end of a `list`.

If we evaluate `items`, the empty brackets will tell us that it's empty:

```
>>> items
[]
```

Let's add "sammiches" to the end:

```
>>> items.append('sammiches')
```

Nothing happened, so how do we know if it worked? Let's check the length. It should be `1`:

```
>>> len(items)
1
```

Hooray! That worked. In the spirit of testing, we'll use the `assert` statement to verify that the length is `1`:

```
>>> assert len(items) == 1
```

The fact that nothing happens is good. When an assertion fails, it triggers an exception that results in a lot of messages.

If you type `items` and press Enter in the REPL, Python will show you the contents:

```
>>> items
['sammiches']
```

Cool, we added one element.

### 3.3.2 Adding many elements to a list

Let's try to add "chips" and "ice cream" to `items`:

```
>>> items.append('chips', 'ice cream')
Traceback (most recent call last):
  File "<stdin>", line 1, in <module>
TypeError: append() takes exactly one argument (2 given)
```

Here is one of those pesky exceptions, and these will cause your programs to *crash*, something we want to avoid at all costs. As you can see, `append() takes exactly one argument`, and we gave it two. If you look at `items`, you'll see that nothing was added:

```
>>> items
['sammiches']
```

OK, so maybe we were supposed to give it a `list` of items to add? Let's try that:

```
>>> items.append(['chips', 'ice cream'])
```

Well, that didn't cause an exception, so maybe it worked? We would expect there to be three `items`, so let's use an assertion to check that:

```
>>> assert len(items) == 3
Traceback (most recent call last):
  File "<stdin>", line 1, in <module>
AssertionError
```

We get another exception, because `len(items)` is not `3`. What is the length?

```
>>> len(items)
2
```

Only `2`? Let's look at `items`:

```
>>> items
['sammiches', ['chips', 'ice cream']]
```

Check that out! Lists can hold any type of data, like strings and numbers and even other lists (see figure 3.2). We asked `items.append()` to add `['chips', 'ice cream']`, which is a `list`, and that's just what it did. Of course, it's not what we wanted.

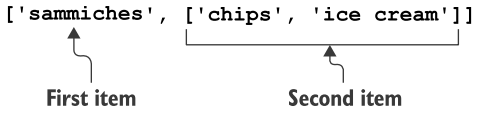



**Figure 3.2 A list can hold any mix of values, such as a string and another list of strings.**

Let's reset `items` so we can fix this:

```
>>> items = ['sammiches']
```

If you read further into the help, you will find the `list.extend()` method:

```
|  extend(self, iterable, /)
|  Extend list by appending elements from the iterable.
```

Let's try that:

```
>>> items.extend('chips', 'ice cream')
Traceback (most recent call last):
  File "<stdin>", line 1, in <module>
TypeError: extend() takes exactly one argument (2 given)
```

Well that's frustrating! Now Python is telling us that `extend() takes exactly one argument`, which, if you refer to the `help`, should be an `iterable`. A `list` is something you can iterate (travel over from beginning to end), so that will work:

```
>>> items.extend(['chips', 'ice cream'])
```

Nothing happened. No exception, so maybe that worked? Let's check the length. It *should* be 3:

```
>>> assert len(items) == 3
```

Yes! Let's look at the items we've added:

```
>>> items
['sammiches', 'chips', 'ice cream']
```

Great! This is sounding like a pretty delicious outing.

If you know everything that will go into the `list`, you can create it like so:

```
>>> items = ['sammiches', 'chips', 'ice cream']
```

The `list.append()` and `list.extend()` methods add new elements to the *end* of a given `list`. The `list.insert()` method allows you to place new items at any position

by specifying the index. I can use the index `0` to put a new element at the beginning of `items`:

```
>>> items.insert(0, 'soda')
>>> items
['soda', 'sammiches', 'chips', 'ice cream']
```

I recommend you read through all the `list` functions so you get an idea of just how powerful this data structure is. In addition to `help(list)`, you can also find lots of great documentation here: https://docs.python.org/3/tutorial/datastructures.html.

### 3.3.3 Indexing lists

We now have a `list` of items. We know how to use `len()` to find how many items there are in the `items` list, and now we need to know how to get parts of the `list` to format.

Indexing a `list` in Python looks exactly the same as indexing a `str` (figure 3.3). (This actually makes me a bit uncomfortable, so I tend to imagine a `str` as a `list` of characters, and then I feel somewhat better.)

```
    0           1            2            3
['soda',  'sammiches',  'chips', 'ice cream']

   -4          -3           -2           -1
```

**Figure 3.3 Indexing lists and strings is the same. For both, you start counting at 0, and you can also use negative numbers to index from the end.**

All indexing in Python is zero-offset, so the first element of `items` is at index `items[0]`:

```
>>> items[0]
'soda'
```

If the index is negative, Python starts counting backwards from the end of the `list`. The index `-1` is the last element of the `list`:

```
>>> items[-1]
'ice cream'
```

You should be very careful when using indexes to reference elements in a `list`. This is unsafe code:

```
>>> items[10]
Traceback (most recent call last):
  File "<stdin>", line 1, in <module>
IndexError: list index out of range
```

**WARNING** Referencing an index that is not present will cause an exception.

You'll soon learn how to safely *iterate*, or travel through, a `list` so that you don't have to use indexes to get at elements.

### 3.3.4 Slicing lists

You can extract "slices" (sub-lists) of a `list` by using `list[start:stop]`. To get the first two elements, you use `[0:2]`. Remember that the `2` is actually the index of the *third* element, but it's not inclusive, as shown in figure 3.4.

```
>>> items[0:2]
['soda', 'sammiches']
```

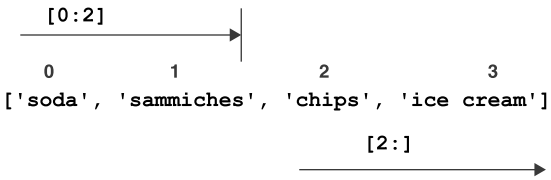



**Figure 3.4 The `stop` value for a list slice is not included. If the `stop` value is omitted, the slice goes to the end of the list.**

If you leave out `start`, it will default to a value of `0`, so the following line does the same thing:

```
>>> items[:2]
['soda', 'sammiches']
```

If you leave out `stop`, it will go to the end of the `list`:

```
>>> items[2:]
['chips', 'ice cream']
```

Oddly, it is completely *safe* for slices to use `list` indexes that don't exist. For example, we can ask for all the elements from index `10` to the end, even though there is nothing at index `10`. Instead of an exception, we get an empty `list`:

```
>>> items[10:]
[]
```

For this chapter's exercise, you're going to need to insert the word "and" into the `list` if there are three or more elements. Could you use a `list` index to do that?

### 3.3.5 Finding elements in a list

Did we remember to pack the chips?

Often you'll want to know if some item is in a `list`. The `index` method will return the location of an element in a `list`:

```
>>> items.index('chips')
2
```

Note that `list.index()` is unsafe code, because it will cause an exception if the argument is not present in the `list`. See what happens if we check for a fog machine:

```
>>> items.index('fog machine')
Traceback (most recent call last):
  File "<stdin>", line 1, in <module>
ValueError: 'fog machine' is not in list
```

You should never use `list.index()` unless you have first verified that an element is present. The `x in y` approach that we used in chapter 2 to see if a letter was in a string of vowels can also be used for lists. We get back a `True` value if `x` is `in` the collection of `y`:

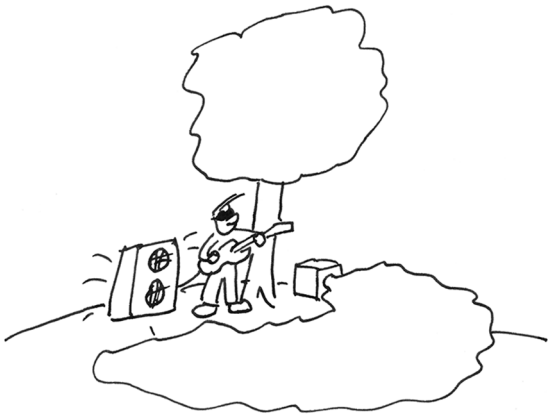
```
>>> 'chips' in items
True
```

I hope they're salt and vinegar chips.

The same code returns `False` if the element is not present:

```
>>> 'fog machine' in items
False
```

We're going to need to talk to the planning committee. What's a picnic without a fog machine?



### 3.3.6 Removing elements from a list

The `list.pop()` method will remove *and return* the element at the index, as shown in figure 3.5. By default it will remove the *last* item (`-1`).

```
>>> items.pop()
'ice cream'
```

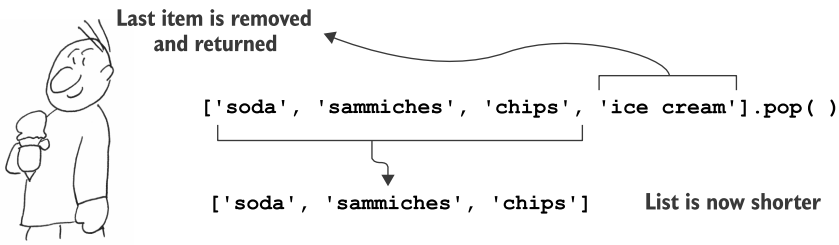



**Figure 3.5** The `list.pop()` method will remove an element from the list.

If we look at `items`, we will see it's now shorter by one:

```
>>> items
['soda', 'sammiches', 'chips']
```

We can use an index value to remove an element at a particular location. For instance, we can use `0` to remove the first element (see figure 3.6):

```
>>> items.pop(0)
'soda'
```

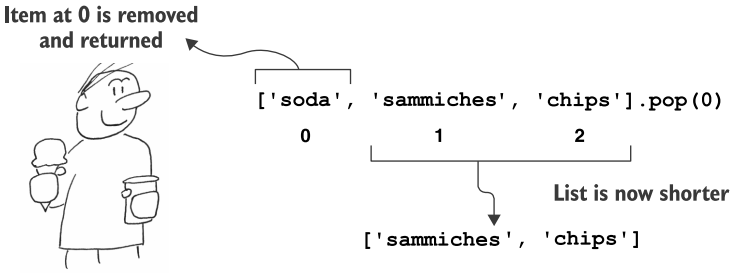



**Figure 3.6 You can specify an index value to `list.pop()` to remove a particular element.**

Now `items` is shorter still:

```
>>> items
['sammiches', 'chips']
```

You can also use the `list.remove()` method to remove the first occurrence of a given item (see figure 3.7):

```
>>> items.remove('chips')
>>> items
['sammiches']
```

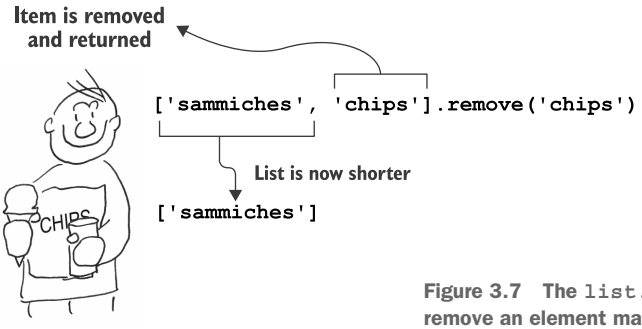



**Figure 3.7 The `list.remove()` method will remove an element matching a given value.**

**WARNING** The `list.remove()` method will cause an exception if the element is not present.

If we try to use `items.remove()` to remove the chips again, we'll get an exception:

```
>>> items.remove('chips')
Traceback (most recent call last):
  File "<stdin>", line 1, in <module>
ValueError: list.remove(x): x not in list
```

So don't use this code unless you've verified that a given element is in the `list`:

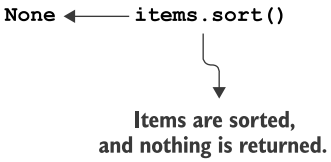
```
item = 'chips'
if item in items:
    items.remove(item)
```

### 3.3.7 Sorting and reversing a list

If the `--sorted` flag is used to call our program, we're going to need to sort the items. You might notice in the help documentation that two methods, `list.reverse()` and `list.sort()`, stress that they work *in place*. That means that the `list` itself will be either reversed or sorted, and nothing will be returned. So, given this `list`,

```
>>> items = ['soda', 'sammiches', 'chips', 'ice cream']
```

the `items.sort()` method will return nothing:




```
>>> items.sort()
```

If you inspect `items`, you will see that the items have been sorted alphabetically:

```
>>> items
['chips', 'ice cream', 'sammiches', 'soda']
```

As with `list.sort()`, nothing is returned from the `list.reverse()` call:

```
>>> items.reverse()
```

But the `items` are now in the opposite order:

```
>>> items
['soda', 'sammiches', 'ice cream', 'chips']
```

The `list.sort()` and `list.reverse()` *methods* are easily confused with the `sorted()` and `reversed()` *functions*. The `sorted()` *function* accepts a `list` as an argument and *returns* a new `list`:

```
>>> items = ['soda', 'sammiches', 'chips', 'ice cream']
>>> sorted(items)
['chips', 'ice cream', 'sammiches', 'soda']
```

It's crucial to note that the `sorted()` function *does not alter* the given `list`:

```
>>> items
['soda', 'sammiches', 'chips', 'ice cream']
```

Note that Python will sort a `list` of numbers *numerically*, so we've got that going for us, which is nice:

```
>>> sorted([4, 2, 10, 3, 1])
[1, 2, 3, 4, 10]
```

**WARNING** Sorting a `list` that mixes strings and numbers will cause an exception!

```
>>> sorted([1, 'two', 3, 'four'])
Traceback (most recent call last):
  File "<stdin>", line 1, in <module>
TypeError: '<' not supported between instances of 'str' and 'int'
```

The `list.sort()` *method* is a function that belongs to the `list`. It can take arguments that affect the way the sorting happens. Let's look at `help(list.sort)`:

```
sort(self, /, *, key=None, reverse=False)
    Stable sort *IN PLACE*.
```

That means we can also `sort()` `items` in reverse, like so:

```
>>> items.sort(reverse=True)
```

Now they look like this:

```
>>> items
['soda', 'sammiches', 'ice cream', 'chips']
```

The `reversed()` function works a bit differently:

```
>>> reversed(items)
<list_reverseiterator object at 0x10e012ef0>
```

I bet you were expecting to see a new `list` with the items in reverse. This is an example of a *lazy* function in Python. The process of reversing a `list` might take a while, so Python is showing that it has generated an *iterator object* that will provide the reversed `list` when we actually need the elements.

We can see the values of our `reversed()` list in the REPL by using the `list()` function to evaluate the iterator:

```
>>> list(reversed(items))
['ice cream', 'chips', 'sammiches', 'soda']
```

As with the `sorted()` function, the original `items` remains unchanged:

```
>>> items
['soda', 'sammiches', 'chips', 'ice cream']
```

If you use the `list.sort()` method instead of the `sorted()` function, you might end up deleting your data. Imagine you wanted to set `items` equal to the sorted list of `items`, like so:

```
>>> items = items.sort()
```

What is in `items` now? If you print `items` in the REPL, you won't see anything useful, so inspect the `type()`:

```
>>> type(items)
<class 'NoneType'>
```

It's no longer a `list`. We set it equal to the result of calling the `items.sort()` method, which changes `items` *in place* and returns `None`.

If the `--sorted` flag is given to your program, you will need to sort your items in order to pass the test. Will you use `list.sort()` or the `sorted()` function?

### 3.3.8 Lists are mutable

As you've seen, we can change a `list` quite easily. The `list.sort()` and `list.reverse()` methods change the whole list, but you can also change any single element by referencing it by index. Maybe we should make our picnic slightly healthier by swapping out the chips for apples:

```
>>> items
['soda', 'sammiches', 'chips', 'ice cream']
>>> if 'chips' in items:
...     idx = items.index('chips')
...     items[idx] = 'apples'
...
```

**See if the string 'chips' is in the list of items.**

**Assign the index of 'chips' to the variable idx.**

**Use the index idx to change the element to 'apples'.**

Let's look at `items` to verify the result:

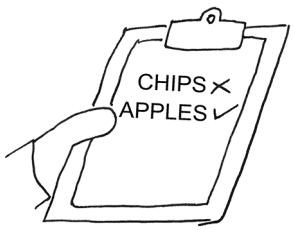
```
>>> items
['soda', 'sammiches', 'apples', 'ice cream']
```

We can also write a couple of tests:

**Make sure "chips" are no longer on the menu.**

```
>>> assert 'chips' not in items
>>> assert 'apples' in items
```

**Check that we now have some "apples."**



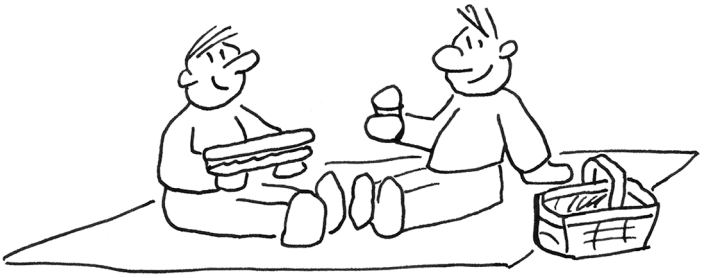
You will need to get the word "and" into your list just before the last element when there are three or more items. Could you use this idea?

### 3.3.9 Joining a list

In this chapter's exercise, you'll need to print a string based on the number of elements in the given list. The string will intersperse other strings like a comma and a space (`', '`) between the elements of the list.

The following syntax will join a list with a string made of a comma and a space:

```
>>> ', '.join(items)
'soda, sammiches, chips, ice cream'
```



The preceding code uses the `str.join()` method and passes the `list` as an argument. It always feels backwards to me, but that's the way it goes.

The result of `str.join()` is a new string:

```
>>> type(', '.join(items))
<class 'str'>
```

The original `list` remains unchanged:

```
>>> items
['soda', 'sammiches', 'chips', 'apples']
```

We can do quite a bit more with Python's `list`, but that should be enough for you to solve this chapter's problem.

## 3.4 Conditional branching with if/elif/else

You need to use conditional branching, based on the number of items, to correctly format the output. In chapter 2's exercise, there were two conditions—either a vowel or not—so we used `if/else` statements. Here we have three options to consider, so you will have to use `elif` (else-if) as well.

For instance, suppose we want to classify someone by their age using three options:

- If their age is greater than `0`, it is valid.
- If their age is less than `18`, they are a minor.
- Otherwise, they are 18 years or older, which means they can vote.

Here is how we could write that code:

```
>>> age = 15
>>> if age < 0:
...     print('You are impossible.')
... elif age < 18:
...     print('You are a minor.')
... else:
...     print('You can vote.')
...
You are a minor.
```

See if you can use that example to figure out how to write the three options for picnic.py. First write the branch that handles one item. Then write the branch that handles two items. Then write the last branch for three or more items. Run the tests *after every change to your program.*

### 3.4.1 Time to write

Now go write the program yourself before you look at my solution. Here are a few hints:

- Go into your 03_picnic directory and run `new.py picnic.py` to create your program. Then run `make test` (or `pytest -xv test.py`). You should pass the first two tests.
- Next work on getting your `--help` usage looking like the example shown earlier in the chapter. It's very important to define your arguments correctly. For the `items` argument, look at `nargs` in `argparse`, as discussed in section A.4.5 of the appendix.
- If you use new.py to start your program, be sure to keep the Boolean flag and modify it for your `sorted` flag.
- Solve the tests in order! First handle one item, then handle two items, and then handle three. Then handle the sorted items.

You'll get the best benefit from this book if you try writing the programs and passing the tests before reading the solutions!

## 3.5 Solution

Here is one way to satisfy the tests. If you wrote something different that passed, that's great!

```
#!/usr/bin/env python3
"""Picnic game"""

import argparse

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

**The get_args() function is placed first so we can easily see what the program accepts when we read it. Note that the function order here is not important to Python, only to us readers.**

```
    parser = argparse.ArgumentParser(
        description='Picnic game',
        formatter_class=argparse.ArgumentDefaultsHelpFormatter)

    parser.add_argument('item',
                        metavar='str',
                        nargs='+',
                        help='Item(s) to bring')

    parser.add_argument('-s',
                        '--sorted',
                        action='store_true',
                        help='Sort the items')

    return parser.parse_args()

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

    args = get_args()
    items = args.item
    num = len(items)

    if args.sorted:
        items.sort()

    bringing = ''
    if num == 1:
        bringing = items[0]
    elif num == 2:
        bringing = ' and '.join(items)
    else:
        items[-1] = 'and ' + items[-1]
        bringing = ', '.join(items)

    print('You are bringing {}.'.format(bringing))

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

The item argument uses nargs='+' so that it will accept one or more positional arguments, which will be strings.

The dashes in the short (-s) and long (--sorted) names make this an option. There is no value associated with this argument. It's either present (in which case it will be True) or absent (False).

Process the command-line arguments and return them to the caller.

The main() function is where the program will start.

Call the get_args() function and put the returned value into the variable args. If there is a problem parsing the arguments, the program will fail before the values are returned.

Copy the item list from args into the new variable items.

Use the length function len() to get the number of items in the list. There can never be zero items because we defined the argument using nargs='+', which always requires at least one value.

The args.sorted value will be either True or False.

If we are supposed to sort the items, call the items.sort() method to sort them in place.

Use an empty string to initialize a variable to hold the items we are bringing.

If the number of items is 1, we will assign the one item to bringing.

If the number of items is 2, put the string ' and ' in between the items.

Join the items on a string of comma and space.

Otherwise, alter the last element in items to append the string 'and ' before whatever is already there.

Print the output string, using the str.format() method to interpolate the bringing variable.

When Python runs the program, it will read all the lines to this point but will not run anything. Here we look to see if we are in the "main" namespace. If we are, we call the main() function to make the program begin.

## 3.6 Discussion

How did it go? Did it take you long to write your version? How different was it from mine? Let's talk about my solution. It's fine if yours is different from mine, just as long as you pass the tests!

### 3.6.1 Defining the arguments

This program can accept a variable number of arguments that are all the same thing (strings). In my `get_args()` method I define an `item` like so:

```
parser.add_argument('item',
                    metavar='str',
                    nargs='+',
                    help='Item(s) to bring')
```

A positional parameter called item

An indication to the user in the usage that this should be a string

The number of arguments, where '+' means one or more

A longer help description that appears for the -h or --help options

This program also accepts `-s` and `--sorted` arguments. They are "flags," which typically means that they are `True` if they are present and `False` if absent. Remember that the leading dashes makes them optional.

```
parser.add_argument('-s',
                    '--sorted',
                    action='store_true',
                    help='Sort the items')
```

The short flag name

The long flag name

If the flag is present, store a True value. The default value will be False.

The longer help description

### 3.6.2 Assigning and sorting the items

In `main()` I call `get_args()` to get the arguments, and I assign them to the `args` variable. Then I create the `items` variable to hold the `args.item` value(s):

```
def main():
    args = get_args()
    items = args.item
```

If `args.sorted` is `True`, I need to sort `items`. I chose the in-place `sort` method here:

```
if args.sorted:
    items.sort()
```

Now I have the items, sorted if needed, and I need to format them for output.

### 3.6.3 Formatting the items

I suggested you solve the tests in order. There are four conditions we need to solve:

- Zero items
- One item

- Two items
- Three or more items

The first test is actually handled by `argparse`—if the user fails to provide any arguments, they get a usage message:

```
$ ./picnic.py
usage: picnic.py [-h] [-s] str [str ...]
picnic.py: error: the following arguments are required: str
```

Since `argparse` handles the case of no arguments, we have to handle the other three conditions. Here's one way to do that:

```
bringing = ''
if num == 1:
    bringing = items[0]
elif num == 2:
    bringing = ' and '.join(items)
else:
    items[-1] = 'and ' + items[-1]
    bringing = ', '.join(items)
```

Initialize a variable for what we are bringing.

Check if the number of items is 1.

If there is one item, bringing is the one item.

Check if the number of items is 2.

If there are two items, we join the items on the string ' and '.

Insert the string 'and ' before the last item.

Otherwise…

Join all the items on the string ', '.

Can you come up with any other ways to do this?

### 3.6.4 Printing the items

Finally, to `print()` the output, I used a format string where the {} indicate a placeholder for a value, like so:

```
>>> print('You are bringing {}.'.format(bringing))
You are bringing salad, soda, and cupcakes.
```

If you prefer, you could use an `f''`-string:

```
>>> print(f'You are bringing {bringing}.')
You are bringing salad, soda, and cupcakes.
```

They both get the job done.

## 3.7 Going further

- Add an option so the user can choose not to print with the Oxford comma (even though that is a morally indefensible option).
- Add an option to separate items with a character passed in by the user (like a semicolon if the list of items needs to contain commas).

Be sure to add tests to the test.py program to ensure your new features are correct!

## Summary

- Python lists are ordered sequences of other Python data types, such as strings and numbers.
- There are methods like `list.append()` and `list.extend()` to add elements to a `list`. Use `list.pop()` and `list.remove()` to remove elements.
- You can use `x in y` to ask if element `x` is in the list `y`. You can also use `list.index()` to find the index of an element, but this will cause an exception if the element is not present.
- Lists can be sorted and reversed, and elements within lists can be modified. Lists are useful when the order of the elements is important.
- Strings and lists share many features, such as using `len()` to find their lengths, using zero-based indexing where `0` is the first element and `-1` is the last, and using slices to extract smaller pieces from the whole.
- The `str.join()` method can be used to make a new `str` from a `list`.
- `if/elif/else` can be used to branch code depending on conditions.