Chapter 4: Going on a picnic: Working with lists

Writing code makes me hungry! Let's write a program to consider some tasty foods we'd like to eat. So far we've handled *one* of something like a name to say "hello" to or a nautical-themed object to point out. In this program, we want to eat one or more foods which we will store in a `list`, a variable that can hold any number of items. We use lists all the time in 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 exercise, we're going on a picnic, and we want to print a list of items to bring. 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 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 put "and" in 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 will separate the items with commas:

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

There's one other twist. You will also need to accept a `--sorted` argument that will require you to sort the items before you print them, but we'll deal with that in a bit. So, your Python program must:

- Store one or more positional arguments in a `list`
- Count the number of arguments
- Possibly modify the list like maybe to 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?

Starting the program

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

```
$ cd picnic
$ new.py picnic.py
Done, see new script "picnic.py."
```

Now 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 is complaining about the fact 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')                 1
>       assert out.strip() == 'You are bringing chips.' 2
E       assert 'str_arg = ""...nal = "chips"' == 'You are bringing chips.'
E         + You are bringing chips.                     3
E         - str_arg = ""                                4
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 =======================
```

1. The program is being run with the argument "chips."
2. This is the line that is causing the error. The output is tested to see if it is equal (`==`) to the string "You are bringing chips."
3. The line starting with a `+` sign is show what was expected.
4. The lines starting with the `-` sign is showing what was returned by the program.

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 have the *correct* arguments, just some examples, so the first thing we need to do is to fix the `get_args` function. Here is what your program should print a usage statement if given *no arguments*:

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

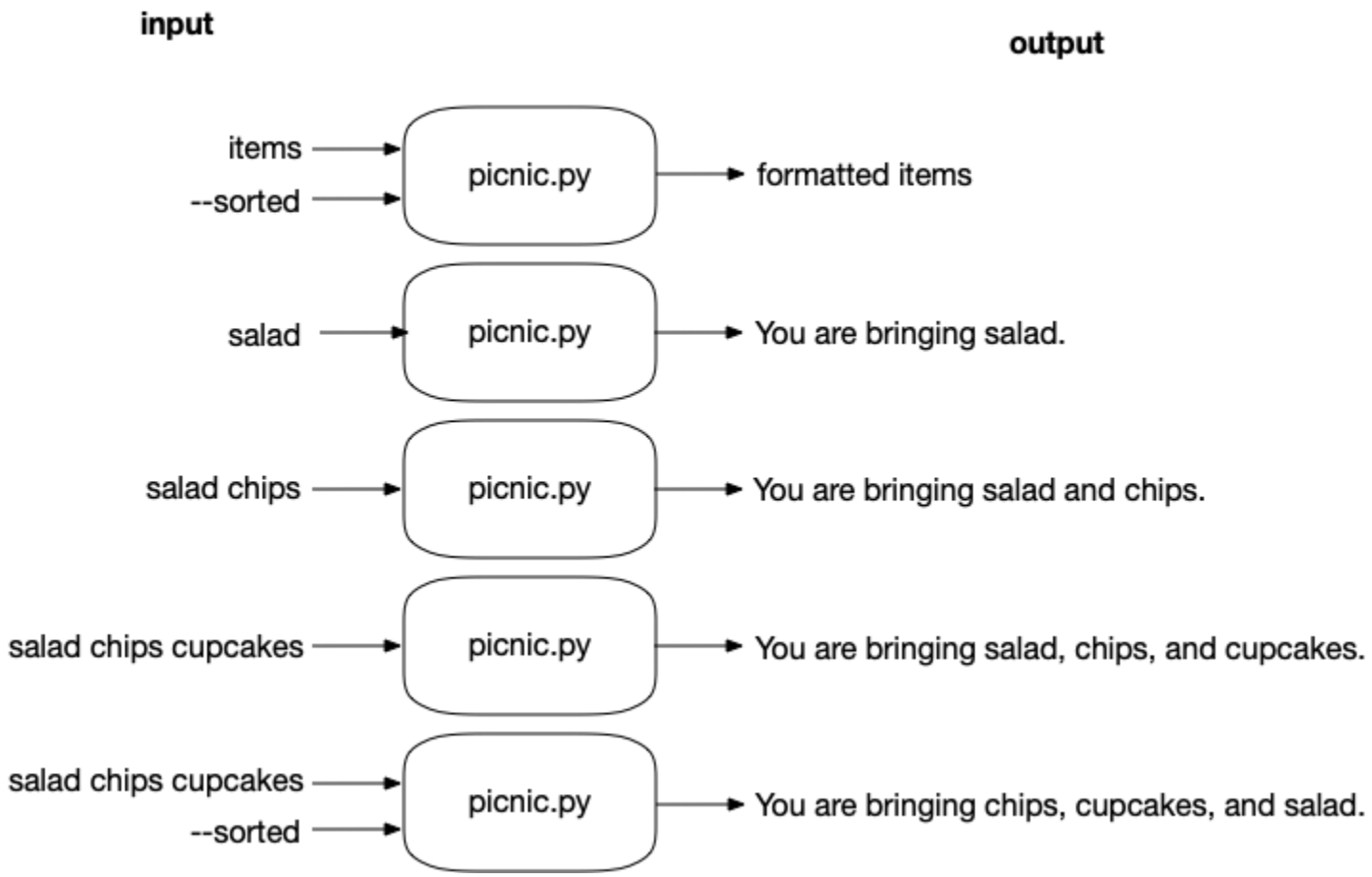
And here is the usage for the `-h` or `--help` flags:

```
$ ./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 a single positional argument and an optional flag called `--sorted`. Modify your `get_args` until it produces the above output. Note that there should be one or more of the `item` parameter, so you should define it with `nargs='+'`. Refer to the section "One or more of the same positional arguments" in the chapter 2.

Writing `picnic.py`

Here is a tasty diagram of the inputs and ouputs for the `picnic.py` program we'll write:



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

1. If one item, state the item:

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

2. If two items, put "and" in between the items. Note that "potato chips" is just *one string* that happens to contain *two words*. If we leave out the quotes, then there would be three arguments to the program. Note that it doesn't matter here if we use single or double quotes:

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

3. If three or more items, place a comma and space between each item 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 if given the `-s` or `--sorted` flag:

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

In order 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 in order to solve this problem.

Stuff about lists

We briefly touched on the idea of a `list` in the `hello.py` program when we looked at `sys.argv`, the "argument vector" for a program. If we run this:

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

Then the arguments `salad chips cupcakes` would be available in `sys.argv` as the `list` of the strings

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

Note that they would be 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 tasty foods we plan to bring on our picnic. I really want you to type these commands yourself, too, 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, we can either use the `list()` function:

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

Or 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. Like 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`.

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 the documentation about the string *methods*, the functions that belong to every `str` in Python. Here I want you to use `help(list)` to find the `list` methods:

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

You'll see lots of "double-under" methods like `__len__`. Skip over those, and the first method we can find is `append`, which we can use to add items to the end of the list. If we evaluate our `items`, we see that the empty brackets tell us that it's empty:

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

Let's add "sammiches" to the end:

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

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

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

Hooray! That worked. In the spirit of testing, use the `assert` method to verify that the length is `1`. The fact that nothing happens is good. When as assertion fails, it triggers an exception that results in a lot of messages. Here, no news is good news:

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

If you type `items<Enter>` in the REPL, Python will show you a the contents:

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

Cool, we added one element.

Adding many elements to a list

Let's try to add "chips" and "ice cream" to the `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. We see that `append() takes exactly one argument`, and we gave it two. If you look at the `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 3 `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
```

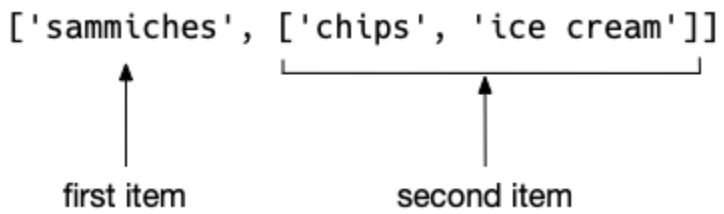
Another exception because `len(items)` is not 3! What is the length?

```
>>> 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. We asked `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*.



Let's reset `items` so we can fix this. We can either use the `del` command to delete the element at index `1`:

```
>>> del items[1]
```

Or reassign it a new value:

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

If you read further into the `help`, you will find the `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 `append` and `extend` methods add new elements to the *end* of a given `list`. The `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']
```

In addition to `help(list)`, you can also find lots of great documentation here:

https://docs.python.org/3/tutorial/datastructures.html

I recommend you read over all the `list` functions so you get an idea of just how powerful this data structure is!

Indexing lists

So now we have a list of `items`. We know how to use `len` to find how many `items` there are, 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`. (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
```

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:

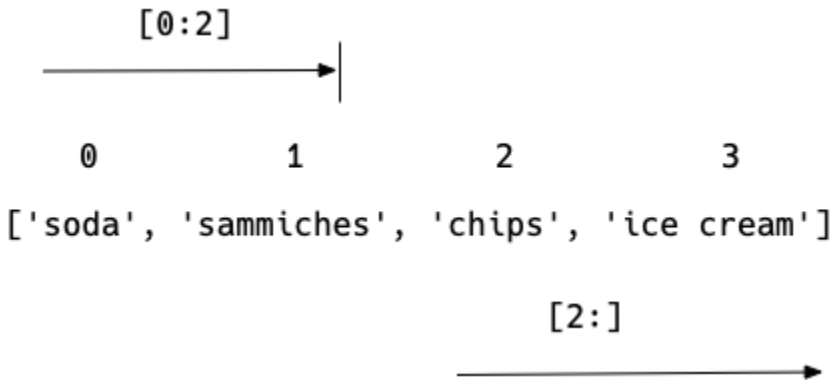
```
>>> items[10]
Traceback (most recent call last):
  File "<stdin>", line 1, in <module>
IndexError: list index out of range
```

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

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

Slicing lists

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




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

If you leave out `start`, it will be `0`, so this 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. Here I 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 our exercise, you're going to need to get the word "and" into the list if there are three or more elements. Could you use a list index to do that?

Finding elements in a list

Did we remember to pack the chips?! Often we want to know if some items is in a `list`. The `index` method will return the location of an element in a `list`:

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

`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 the 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 `index` unless you have first verified that an element is present. The `x in y` that we used in "Crow's Nest" to see if a letter was in the list 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 returns `False` if it is not present:

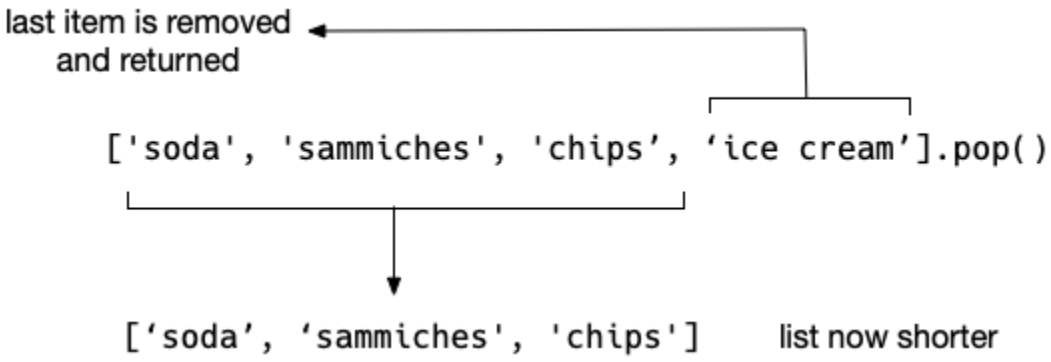
```
>>> 'fog machine' in items
False
```

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

Removing elements from a list

We've seen that we can use the `del` function to delete a `list` element by index. The `list.pop` method will remove *and return* the element at the index. By default it will remove the *last* item (`-1`):

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

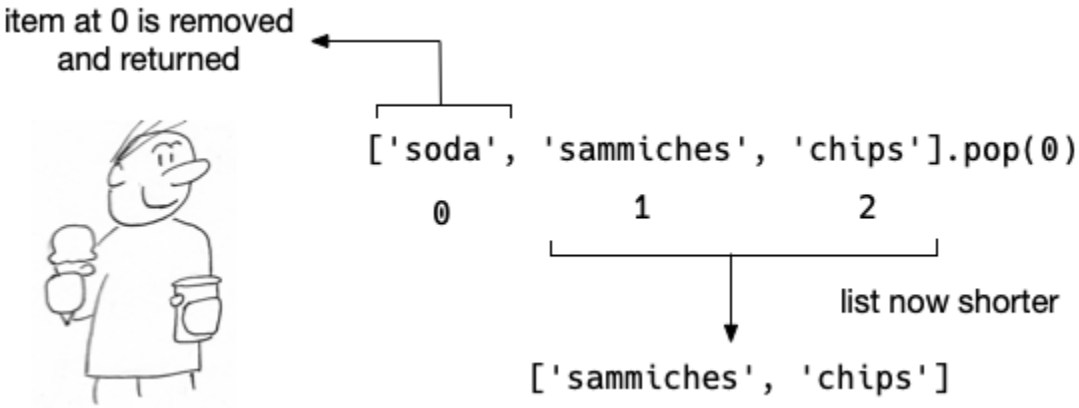


If we look at the list, we will see it's shorter by one:

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

You can use an item's index to remove an element at a particular location. For instance, we can use `0` to remove the first element:

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



And now our `list` is shorter still:

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

You can also use the `list.remove` method to remove the first occurrence of a given item:

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

The `list.remove` will cause an exception if the element is not present.

If we try to `remove` the chips again, we 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:

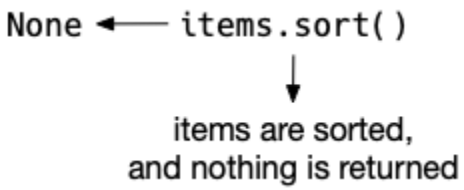
```
item = 'chips'
if item in items:
    items.remove(item)
```

Sorting and reversing a list

If the `--sorted` flag is present, 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 `sort` method will return nothing:




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

But if you inspect `items`, you will see they have been sorted alphabetically:

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

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]
```

As with `list.sort`, we see nothing on 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 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 `list`:

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

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 the `help(list.sort)`:

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

So we could also `sort` the `items` in `reverse` like so:

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

And 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 just as soon as we actually need the elements. We can do that 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` itself remain 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 your `items` equal to the sorted list of `items` like so:

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

What is in `items` now? If you print the `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 called `items.sort()` method that works on the `list` *in-place* and returns `None`. I would note that I tend to not use the `sort`/`reverse` methods because I don't generally like to mutate my data. I would tend to do something like this:

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

Now I have explictly named a `sorted_items` list, and the original `items` has not been altered.

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?

Lists are mutable

As we'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 make our picnic slightly healthier by changing out the chips for apples:

```
>>> items
['soda', 'sammiches', 'chips', 'ice cream']
>>> if 'chips' in items:             1
...     idx = items.index('chips')   2
...     items[idx] = 'apples'        3
...
```

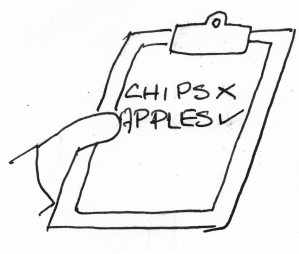
1. See if the string `'chips'` is in the list of `items`.
2. Assign the index of `'chips'` to the variable `idx`.
3. Use the index `idx` to change the element to `'apples'`.

Let's look at `items` to verify:

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

We can also write a couple of tests:

```
>>> assert 'chips' not in items  1
>>> assert 'apples' in items     2
```

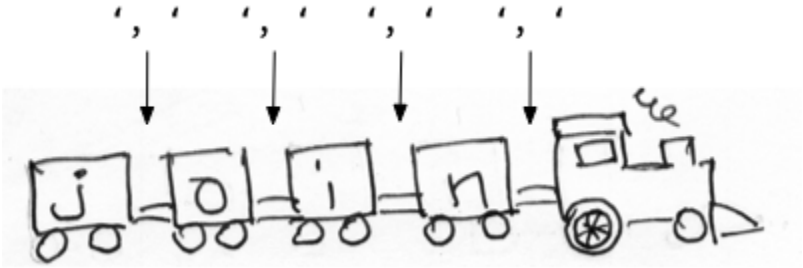


1. Make sure "chips" are no longer on the menu.
2. Check that we now have some "apples."

Joining a `list`

In our exercise, you'll need to print a string based on the number of elements in the given list. The string will intersperse some other string like `', '` in between all the elements of the list. Oddly, this the syntax to join a list on the string made of the comma and a space:

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



Here we use the `str.join` method and pass 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']
```

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

Conditional branching with if/elif/else

You need to use the conditional branching based on the number of items to correctly format the output. In the "Crow's Nest" exercise, there were two conditions (a "binary" choice) — 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). For instance, we want to classify someone by their age by three options:

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

Here is how I 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 to figure out how to write the three options for `picnic.py`. That is, 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*.

Now go write the program yourself before you continue to look at my solution.

Hints:

- Go into your `picnic` directory and run `new.py picnic.py` to start your program. Then run `make test` (or `pytest -xv test.py`) and you should pass the first two tests.
- Next work on getting your `--help` usage looking like the above. It's very important to define your arguments correctly. For the `items` argument, look at `nargs` in `argparse` as discussed in chapter 1's "One or more of the same positional arguments" section.
- If you use `new.py` to start your program, be sure to leave the "boolean flag" and modify it for your `sorted` flag.
- Solve the tests in order! First handle one item, then handle two items, then handle three. Then handle the sorted items.

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

Solution

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

import argparse


# --------------------------------------------------
def get_args():  1
    """Get command-line arguments"""

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

    parser.add_argument('item',  2
                        metavar='str',
                        nargs='+',
                        help='Item(s) to bring')

    parser.add_argument('-s',    3
                        '--sorted',
                        action='store_true',
                        help='Sort the items')

    return parser.parse_args()   4


# --------------------------------------------------
def main():  5
    """Make a jazz noise here"""

    args = get_args()  6
    items = args.item  7
    num = len(items)   8

    if args.sorted:    9
        items.sort()   10

    bringing = ''                        11
    if num == 1:                         12
        bringing = items[0]              13
    elif num == 2:                       14
        bringing = ' and '.join(items)   15
    else:                                16
        items[-1] = 'and ' + items[-1]   17
        bringing = ', '.join(items)      18

    print('You are bringing {}.'.format(bringing))  19


# --------------------------------------------------
if __name__ == '__main__':  20
    main()                  21
```

1 The `get_args` function is placed first so that 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, the reader.

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

3 The dashes in the short (`-s`) and long (`--sort`) 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 (or `False`).

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

5 The `main` function is where the program will start.

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

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

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

9 The `args.sorted` value with either be `True` or `False` because it was defined as a "flag."

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

11 Initialize the variable `bringing` with an empty string. We'll put the items we're bringing into this.

12 If the number of items is 1...

13 Then we will assign the one item to `bringing` .

14 If the number of items is 2...

15 Put the string `' and '` in between the `items` .

16 Otherwise...

17 Alter the last element in `items` to append the string `'and '` before whatever is already there.

18 Join the `items` on the string `', '` .

19 Print the output string, using the `str.format` method to interpolate the `bringing` variable.

20 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.

21 If we are, call the `main` function to make the program begin.

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 perfectly fine if yours is really different from mine, just as long as you pass the tests!

Defining the arguments

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

```
parser.add_argument('item',                    1
                    metavar='str',             2
                    nargs='+',                 3
                    help='Item(s) to bring')   4
```

1 A single, required, positional (because no dashes in name) argument called `item` .

2 An indicator to the user in the usage that this should be a string.

3 The number of arguments where `'+'` means *one or more*.

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

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

```
parser.add_argument('-s',                  1
                    '--sorted',            2
                    action='store_true',   3
                    help='Sort the items') 4
```

1 The short flag name.

2 The long flag name.

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

4 The longer help description.

Assigning and sorting the items

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

```
args = get_args()
items = args.item
```

If `args.sorted` is `True`, then I need to sort my `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 the output.

Formatting the items

I suggested you solve the tests in order. There are 4 conditions we need to solve:

1. Zero items
2. One item
3. Two items
4. Three or more items

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

```
$ ./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 = ''                          1
if num == 1:                           2
    bringing = items[0]                3
elif num == 2:                         4
    bringing = ' and '.join(items)     5
else:                                  6
    items[-1] = 'and ' + items[-1]     7
    bringing = ', '.join(items)        8
```

1. Initialize a variable for what we are `bringing`.
2. Check if the number of items is one.
3. If there is one item, then `bringing` is the one item.
4. Check if the number of items is two.
5. If two items, join the `items` on the string `' and '`.
6. Otherwise...
7. Insert the string `'and '` before the last item
8. Join all the `items` on the string `', '`.

Can you come up with any other ways?

Printing the items

Finally to `print` the output, I can use a format string where the `{}` indicates a placeholder for some value like so:

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

Or, if you prefer, you can use an `f''`-string:

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

They both get the job done, so whichever you prefer.

Testing

For this exercise, I've written the tests for you. Can you see how this is similar to the `hello.py` where we created a `greet` function inside the program and added a `test_greet` function to test it? If I were writing this code for myself, I would do the same by creating a function to format the items and placing the unit test for that inside the program itself.

I believe there is an art to testing. It's up to you to figure out how best to test your own code. Try copying some of the relevant `test_` functions from `test.py` into your `picnic.py` and then running `pytest picnic.py` to see how it works. What do you prefer?

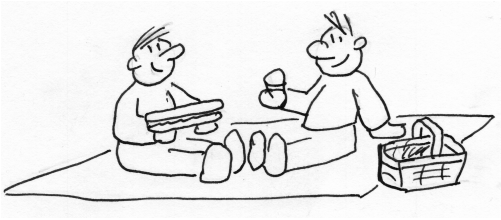
Summary

- Python lists are ordered sequences of other Python data types such as strings and numbers.
- There are methods like `append` and `extend` to add elements to a `list` and `pop` and `remove` to remove them.

- You can use `x in y` to ask if element `x` is in the list `y`. You could 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.

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 some character passed in by the user (like a semicolon if the list of items needed to contain commas).



Last updated 2020-01-18 10:15:21 -0700