

# Recipes

You are not logged in.

Please [Log In](#) for full access to the web site.

Note that this link will take you to an external site (<https://shimmer.mit.edu>) to authenticate, and then you will be redirected back to this page.

## Table of Contents

- [1\) Preparation](#)
- [2\) Introduction](#)
- [3\) Data Structures](#)
- [4\) Transforming Recipes](#)
  - [4.1\) Efficiency and Clarity](#)
- [5\) Finding the Lowest Cost](#)
  - [5.1\) Nonexistent Items](#)
  - [5.2\) Forbidden Items](#)
- [6\) Cheapest Flat Recipe](#)
  - [6.1\) Flat Recipe Helpers](#)
- [7\) All Flat Recipes](#)
  - [7.1\) Generating Combinations of Flat Recipes](#)
- [8\) Summary](#)
- [9\) Code Submission](#)
- [10\) Checkoff](#)

## 1) Preparation

This lab assumes you have Python 3.11+ installed on your machine (3.12 recommended).

The following file contains code and other resources as a starting point for this lab: [recipes.zip](#)

Your raw score for this lab will be counted out of 5 points. Your score for the lab is based on:

- correctly answering the questions throughout this page (.5 points),
- passing the tests in `test.py` (3.5 points), and
- a checkoff conversation (1 point).

### Reminder: Academic Integrity

Please also review the [academic integrity policies](#) before continuing. In particular, **note that you are not allowed to use any code other than that which you have written yourself, including code from online sources and code produced by advanced code-completion tools.**

## 2) Introduction

The goal of this lab is to begin to familiarize you with recursion and different ways it can be applied by building a tool that can keep track of a set of recipes, each of which is comprised of a number of different ingredients. The challenge is that some of the ingredients for your recipes may themselves be made up of multiple ingredients! Furthermore, those ingredients might be made up of other ingredients, and so on.

For this type of nested data, recursion gives us an elegant way to design algorithms prepared for arbitrary nesting levels. For example, if we are trying to figure out if a particular recipe can be made, we can first recursively check if the component ingredients can be made and then decide for the recipe in question.

## 3) Data Structures

Our initial data structure for this assignment will be a list of food items. Some of these food items are "atomic" (i.e., they are not comprised of other items), and some are "compound" (i.e., they are created by combining other food items).

We will represent an atomic food item as a tuple of three elements: the word "atomic", a name, and a cost. For example, the following represents an atomic food item:

```
('atomic', 'jalapeño pepper', 0.15)
```

A compound food item is also represented by a tuple, but its first element is the word "compound," and its last element is an ingredient list rather than a cost. The ingredient list will be a list of tuples, where each tuple contains an ingredient (a food item, which could be either atomic or compound) and an integer quantity telling us how much of that ingredient is required to prepare the recipe. Note the ingredient lists we provide contain no cycles (meaning there are no compound items whose ingredients contain themselves.)

An example recipe is:

```
('compound', 'spicy chili', [('can of beans', 3), ('jalapeño pepper', 10), ('chili powder', 1), ('cornbread', 2)])
```

Throughout this lab, we will be working with a database of recipes, which we'll represent as a list of food items (which may contain both compound and atomic items). Note that there may be multiple ways of creating the same compound food item. In those cases, we will simply have multiple entries in the food database that have the same name, and it should be assumed that all of the associated ingredient lists represent valid ways to make that item. Here is an example recipe database:

```
example_recipes = [  
    ('compound', 'chili', [('beans', 3), ('cheese', 10), ('chili powder',
```

```

1), ('cornbread', 2), ('protein', 1)]),
  ('atomic', 'beans', 5),
  ('compound', 'cornbread', [('cornmeal', 3), ('milk', 1), ('butter',
5), ('salt', 1), ('flour', 2)]),
  ('atomic', 'cornmeal', 7.5),
  ('compound', 'burger', [('bread', 2), ('cheese', 1), ('lettuce', 1),
('protein', 1), ('ketchup', 1)]),
  ('compound', 'burger', [('bread', 2), ('cheese', 2), ('lettuce', 1),
('protein', 2),]),
  ('atomic', 'lettuce', 2),
  ('compound', 'butter', [('milk', 1), ('butter churn', 1)]),
  ('atomic', 'butter churn', 50),
  ('compound', 'milk', [('cow', 1), ('milking stool', 1)]),
  ('compound', 'cheese', [('milk', 1), ('time', 1)]),
  ('compound', 'cheese', [('cutting-edge laboratory', 11)]),
  ('atomic', 'salt', 1),
  ('compound', 'bread', [('yeast', 1), ('salt', 1), ('flour', 2)]),
  ('compound', 'protein', [('cow', 1)]),
  ('atomic', 'flour', 3),
  ('compound', 'ketchup', [('tomato', 30), ('vinegar', 5)]),
  ('atomic', 'chili powder', 1),
  ('compound', 'ketchup', [('tomato', 30), ('vinegar', 3), ('salt', 1),
('sugar', 2), ('cinnamon', 1)]), # the fancy ketchup
  ('atomic', 'cow', 100),
  ('atomic', 'milking stool', 5),
  ('atomic', 'cutting-edge laboratory', 1000),
  ('atomic', 'yeast', 2),
  ('atomic', 'time', 10000),
  ('atomic', 'vinegar', 20),
  ('atomic', 'sugar', 1),
  ('atomic', 'cinnamon', 7),
  ('atomic', 'tomato', 13),
]

```

Throughout the lab, it is safe to assume that each food item is either compound or atomic (i.e., we won't have both an atomic food item and a compound food item that share the same name), and that each atomic food item will only occur once (though compound food items may appear more than once, to represent multiple combinations of ingredients that could be used to make that food item).

Before we dive in, answer the following questions about this data structure (if looking at the structure on the page is too tedious, you can try using Python to help you out!).

How many atomic food items exist in the `example_recipes` database?

How many *distinct* compound food items exist in the `example_recipes` database?

## 4) Transforming Recipes

Before we dive into recursion, let's take a look at our recipes data structure. Each recipe list can contain both compound and atomic food items. Additionally, each compound food item can have one or more different ingredient lists associated with it. In this structure, if we wanted to look up all the different ways to make a specific compound food item, we'd have to loop over the whole list.

If this is reminding you of the `raw_data` structure from the Bacon Number lab, then your intuition is correct! In the Bacon Number lab, we transformed the initial `raw_data` to make it easier to find all the actors a particular person acted with. At this point, you might be wondering if there's a similar way to reorganize the recipes database. Like we did in the Bacon lab, let's consider what questions we might want to ask about our data:

- Given the name of a food item, can we quickly determine whether it is atomic or compound?
- Given an **atomic food item**, can we efficiently determine its cost? (This will help us determine the total price of a recipe.)
- Given a **compound food item**, what are all the different ingredient lists we can use to make this item?

If you're thinking that these questions sound like something a dictionary could easily answer, you're right! Because in later parts of the lab we'll be asking these questions a lot, your first task for this lab is to implement the following functions in `lab.py`:

- `atomic_ingredient_costs(recipes)` also takes in a list of recipes and returns a dictionary of atomic food item names mapped to their costs.
- `compound_ingredient_possibilities(recipes)` takes in a list of recipes, as described in the previous section, and returns a dictionary that maps compound food item names to lists of associated ingredient lists.

As a reminder, an **ingredient list** is a list of tuples, where each tuple consists of a (food\_item, quantity).

The questions below refer to `example_recipes` from the previous section.

Using `atomic_ingredient_costs(example_recipes)`, what is the total cost of buying 1 of every atomic food item?

Using `compound_ingredient_possibilities(example_recipes)`, how many compound food items can be made multiple ways?

## 4.1) Efficiency and Clarity

Using `compound_ingredient_possibilities` and `atomic_ingredient_costs` in the other parts of the lab is highly encouraged but optional. Note that unlike the Bacon Number lab, most of the recursive functions we're asking you to write below make use of the original recipes data structure. How can we use these helper functions efficiently when we're recursively calling the functions that use them? Generally speaking, you can write these functions in a way that behaves similarly to what we saw in earlier labs using one of a couple of different approaches:

One approach involves using the main function (e.g., `lowest_cost`, which we'll describe in the next section) to first create our dictionaries and then call a separate "helper" function that takes in the dictionaries as inputs and performs the recursive process. The following sketch illustrates this idea:

```
def my_main_function(input1, input2, ...):
    # set up some new data structures
    improved_input1 = ...
    return my_recursive_helper(improved_input1, input2, ...)

def my_recursive_helper(better_input1, input2, ...):
    # perform the recursive process
    return the_solution
```

A twist on this approach involves defining the recursive helper function *inside* our main function. The neat thing about closures is that the inner function has access to whatever variables are defined in its enclosing frame, so we do not need to pass them around as arguments:

```
def my_main_function(input1, input2, ...):
    # set up some new data structures
    improved_data_structure_1 = ...

    def my_recursive_helper():
        # perform the recursive process
        # note that within this function we can access all the variables
        # the enclosing frame, e.g., improved_data_structure_1

    return the_solution
```

```
return my_recursive_helper()
```

Look for opportunities to apply these patterns as you work on the remainder of the lab, as doing so can often help improve both the efficiency and clarity of your code. Remember, readable code is often easier to debug when issues arise!

## 5) Finding the Lowest Cost

Now let's move on to our first recursive task. In this section, we'll write a function that will tell us how much the cheapest way to make a particular food item costs.

In your `lab.py` file, implement the function `lowest_cost(recipes, food_item)`, which takes in a recipes list (in the form described in section 3 above) and the name of a particular food item. It should return a number representing the minimum cost required to make that food item (by purchasing all necessary atomic items).

For now, we will assume that the database is complete, i.e., that every recipe in the database can be created using some combination of the atomic food items in the database.

As a small example, consider the following:

```
dairy_recipes = [
    ('compound', 'milk', [('cow', 2), ('milking stool', 1)]),
    ('compound', 'cheese', [('milk', 1), ('time', 1)]),
    ('compound', 'cheese', [('cutting-edge laboratory', 11)]),
    ('atomic', 'milking stool', 5),
    ('atomic', 'cutting-edge laboratory', 1000),
    ('atomic', 'time', 10000),
    ('atomic', 'cow', 100),
]
```

Given the recipes above, what is the result of `lowest_cost(dairy_recipes, 'cheese')`?

Calculating the total cost of an ingredient list requires summing the result of multiplying each ingredient's quantity by its calculated cost. For this and later parts of the lab, we'll assume that all quantities of an ingredient come from the same recipe. For example, consider the following recipe list:

```
cookie_recipes = [
    ('compound', 'cookie sandwich', [('cookie', 2), ('ice cream scoop',
3)]),
    ('compound', 'cookie', [('chocolate chips', 3)]),
```

```
[
    ('compound', 'cookie', [('sugar', 10)]),
    ('atomic', 'chocolate chips', 200),
    ('atomic', 'sugar', 5),
    ('compound', 'ice cream scoop', [('vanilla ice cream', 1)]),
    ('compound', 'ice cream scoop', [('chocolate ice cream', 1)]),
    ('atomic', 'vanilla ice cream', 20),
    ('atomic', 'chocolate ice cream', 30),
]
```

As written, a cookie sandwich must contain two of the same cookies and three of the same ice cream scoops. So in this case there are four possible ways we could make a cookie sandwich:

- 6 chocolate chips and 3 vanilla ice cream
- 6 chocolate chips and 3 chocolate ice cream
- 20 sugar and 3 vanilla ice cream
- 20 sugar and 3 chocolate ice cream

Note that we aren't considering possibilities that fulfill the 3-ice-cream-scoop requirement by having 2 chocolate ice cream and 1 vanilla ice cream, or any other combination. If we wanted to allow for mixing and matching ice cream scoops within the cookie sandwich recipe, we could have included the following ingredient list instead:

```
('compound', 'cookie sandwich', [('cookie', 1), ('cookie', 1), ('ice
cream scoop', 1), ('ice cream scoop', 1), ('ice cream scoop', 1)])
```

Using the original definition of `cookie_recipes` from above, what is the result of running `lowest_cost(cookie_recipes, 'cookie sandwich')`?

Once you have a plan in mind, go ahead and implement `lowest_cost` in `lab.py`. When you have this function working (and before moving on) you should pass the first several test cases for `lowest_cost` (the ones with `all_included` in the name).

## 5.1) Nonexistent Items

Now let's start to think of some potential failure modes of this approach. For one, if the database came from a local grocery store, you can imagine that it might run out of stock of particular items sometimes. Since we can't make a recipe if one (or more) of the ingredients is missing, we should probably take this scenario into account.

Update `lowest_cost` so that it returns `None` if it is given the name of a food item that is not present in the database. Additionally, consider any recipe that includes a missing food item to be impossible to complete.

For example, let's consider our `dairy_recipes` database again:

```
dairy_recipes = [
    ('compound', 'milk', [('cow', 2), ('milking stool', 1)]),
    ('compound', 'cheese', [('milk', 1), ('time', 1)]),
    ('compound', 'cheese', [('cutting-edge laboratory', 11)]),
    ('atomic', 'milking stool', 5),
    ('atomic', 'cutting-edge laboratory', 1000),
    ('atomic', 'time', 10000),
    ('atomic', 'cow', 100),
]
```

Now let's imagine that the entry for `'cow'` was no longer present:

```
dairy_recipes_2 = [
    ('compound', 'milk', [('cow', 2), ('milking stool', 1)]),
    ('compound', 'cheese', [('milk', 1), ('time', 1)]),
    ('compound', 'cheese', [('cutting-edge laboratory', 11)]),
    ('atomic', 'milking stool', 5),
    ('atomic', 'cutting-edge laboratory', 1000),
    ('atomic', 'time', 10000),
]
```

In this case, calling `lowest_cost(dairy_recipes_2, 'cow')` should return `None`. Calling `lowest_cost(dairy_recipes_2, 'milk')` should also return `None` because in this case `'milk'` cannot be made without `'cow'`.

Given the example recipes above, what is the result of `lowest_cost(dairy_recipes_2, 'cheese')`?

Note that if we went further and also removed `'cutting-edge laboratory'`, then `lowest_cost` would return `None` if we tried to calculate the cost of `'cheese'`, since there is no way to make it using the remaining atomic items.

Update your `lowest_cost` function to handle these situations. After doing so, your code should pass the test cases for `lowest_cost` containing the word "excluded" that do not include the word "forbidden".

## 5.2) Forbidden Items

Sometimes when preparing food, we want to avoid using certain ingredients, even if they are technically available. For example, consider making food for someone with a dietary restriction; they may ask you not to include a particular food item even if it is available from the store!



We will make this change by adding an *optional parameter* to the `lowest_cost` function, so that we can optionally provide an iterable of food item names to ignore.

After implementing this change, calling `lowest_cost(dairy_recipes_2, 'cheese', ["cutting-edge laboratory"])` should provide the same results as if we removed `'cutting-edge laboratory'` from the database, but `lowest_cost(dairy_recipes_2, 'cheese')` should work as before.

If you're not familiar with optional parameters, we have [a small primer you can use to get started](#). Feel free to come to open lab hours if you are having trouble!

After adding support for this optional parameter, your code should pass all the test cases for `lowest_cost`!

## 6) Cheapest Flat Recipe

Now that we've written `lowest_cost`, we can figure out how much money to take to the grocery store, taking into account their stock and things like dietary restrictions or foods that you or your friends don't like. But this only tells us how much money we need to take to the store; it doesn't actually tell us what to buy! We'll remedy that in this section by making a new function, `cheapest_flat_recipe`, which will tell us all of the atomic food items needed to create a given food item in the least-expensive way.

**Before you start working on this function, read through this whole section.** We're requiring you to implement a couple of helper functions (`scaled_flat_recipe` and `add_flat_recipes`) that will likely be helpful as you work on `cheapest_flat_recipe`.

`cheapest_flat_recipe` will take the same arguments as `lowest_cost` (including the optional parameter representing forbidden food items), but unlike `lowest_cost`, it should return a "flat recipe": a dictionary mapping the necessary atomic food items to their quantities. Importantly, a flat recipe dictionary should *only* contain atomic food items, and it should contain the right number to build the original food item completely (possibly by creating other intermediate food items along the way).

For example, let's return to our `dairy_recipes` example from earlier:

```
dairy_recipes = [
    ('compound', 'milk', [('cow', 2), ('milking stool', 1)]),
    ('compound', 'cheese', [('milk', 1), ('time', 1)]),
    ('compound', 'cheese', [('cutting-edge laboratory', 11)]),
    ('atomic', 'milking stool', 5),
    ('atomic', 'cutting-edge laboratory', 1000),
    ('atomic', 'time', 10000),
    ('atomic', 'cow', 100),
]
```

Calling `cheapest_flat_recipe(dairy_recipes, 'cheese')` should return:

```
{'cow': 2, 'milking stool': 1, 'time': 1}
```

And calling `cheapest_flat_recipe(dairy_recipes, 'cheese', ['cow'])` should return:

```
{'cutting-edge laboratory': 11}
```

If the same minimal cost could be achieved by multiple different recipes, it is fine for your function to return any of those recipes.

## 6.1) Flat Recipe Helpers

Before you start implementing `cheapest_flat_recipe`, it's worth asking yourself what are the similarities and differences between it and the `lowest_cost` function you wrote in the previous section?

Although on the surface these functions seem similar (both functions seek to minimize cost after all), constructing the flat recipe dictionary is more complex than calculating the cost. As such, we're asking you to write a couple of helper functions that will be useful in `cheapest_flat_recipe` and beyond.

These functions are:

- `scaled_flat_recipe(flat_recipe, n)` - takes a flat-recipe dictionary as described in the previous section and returns a new flat recipe with all values scaled by `n`. Does not mutate input.
- `add_flat_recipes(flat_recipes)` - takes a list of flat recipes and returns a new flat-recipe dictionary that maps keys to the sums of the corresponding values across all the dictionaries. Does not mutate the input.

Let's give these helper functions a try on some example recipes:<sup>1</sup>

Given the recipe for some soup below:

```
soup = {"carrots": 5, "celery": 3, "broth": 2, "noodles": 1,  
        "chicken": 3, "salt": 10}
```

what is the result of `scaled_flat_recipe(soup, 3)`?

Say you wanted to make only one serving of the soup from the previous question, along with a carrot cake and some bread, using the following recipes:

```
carrot_cake = {"carrots": 5, "flour": 8, "sugar": 10, "oil": 5,
               "eggs": 4, "salt": 3}
bread = {"flour": 10, "sugar": 3, "oil": 3, "yeast": 15, "salt": 5}
grocery_list = [soup, carrot_cake, bread]
```

If you went to the grocery store to buy everything needed for these three recipes, how much would you need of each ingredient? Enter your solution as a dictionary in the same format as the recipes. (Hint: `add_flat_recipes` might be helpful for this question!)



Now that we have some additional helper functions available to us, it's time to implement `cheapest_flat_recipe`! Can you see where these helper functions (or others) could fit into your plan for `cheapest_flat_recipe`? Note that using these helper functions is suggested but not required (there are many valid solutions.)

Once you have implemented `cheapest_flat_recipe`, you should be passing more test cases, including all the ones that start with `test_cheapest`.

## 7) All Flat Recipes

It's great to know the least-expensive way to create a given piece of food, but in some cases (maybe a celebration of some kind!), we may want to consider other ways of creating that food item as well.

For the final section of the lab, we would like to write a couple of functions that can help us find *all* of the ways we could create a given food item.

To this end, we'll implement `all_flat_recipes` and a helper explained a bit later called `combined_flat_recipes`. `all_flat_recipes` should take the same arguments as `cheapest_flat_recipe` (including the optional parameter of foods to avoid), and it should return a list of dictionaries, representing each possible flat recipe that can be constructed from the given recipe list.

**Before you start working on this function, read through this whole section.**

For example, in our original `example_recipes` list there are several different ways to make a 'burger', depending on which way we chose to make the burger, how we made the cheese, and whether we chose the fancy ketchup:

```
>>> for i in all_flat_recipes(example_recipes, 'burger'):
...     print(i)

{'yeast': 2, 'salt': 2, 'flour': 4, 'cutting-edge laboratory': 22,
 'lettuce': 1, 'cow': 2}
{'yeast': 2, 'salt': 2, 'flour': 4, 'cow': 4, 'milking stool': 2, 'time':
 2, 'lettuce': 1}
{'yeast': 2, 'salt': 2, 'flour': 4, 'cutting-edge laboratory': 11,
 'lettuce': 1, 'cow': 1, 'tomato': 30, 'vinegar': 5}
{'yeast': 2, 'salt': 3, 'flour': 4, 'cutting-edge laboratory': 11,
 'lettuce': 1, 'cow': 1, 'tomato': 30, 'vinegar': 3, 'sugar': 2,
 'cinnamon': 1}
{'yeast': 2, 'salt': 2, 'flour': 4, 'cow': 2, 'milking stool': 1, 'time':
 1, 'lettuce': 1, 'tomato': 30, 'vinegar': 5}
{'yeast': 2, 'salt': 3, 'flour': 4, 'cow': 2, 'milking stool': 1, 'time':
 1, 'lettuce': 1, 'tomato': 30, 'vinegar': 3, 'sugar': 2, 'cinnamon': 1}
```

If we don't allow 'milk', we end up with a smaller list of possibilities:

```
>>> for i in all_flat_recipes(example_recipes, 'burger', ('milk',)):
...     print(i)

{'yeast': 2, 'salt': 3, 'flour': 4, 'cutting-edge laboratory': 11,
 'lettuce': 1, 'cow': 1, 'tomato': 30, 'vinegar': 3, 'sugar': 2,
 'cinnamon': 1}
{'yeast': 2, 'salt': 2, 'flour': 4, 'cutting-edge laboratory': 11,
 'lettuce': 1, 'cow': 1, 'tomato': 30, 'vinegar': 5}
{'yeast': 2, 'salt': 2, 'flour': 4, 'cutting-edge laboratory': 22,
 'lettuce': 1, 'cow': 2}
```

The list returned from your function may contain the appropriate elements in any order, and it is OK if it contains duplicates.

This function is going to be complex! But we can help ourselves out by carefully breaking the problem down before we dive in too deeply. Once again, we should ask ourselves: what is similar between this function and what we wrote previously, and what is different?

In this case, one huge difference is that the *type* of the output is changing: we should always be returning a

**list** of possible recipes. This affects not only what we return from our recursive case(s) but from our base case(s) as well; everything that this function returns should always be a list. And we'll need to think carefully about how to construct those lists. As an example, let's return to one of our earlier recipe databases and answer a few questions about it:

```
cookie_recipes = [  
    ('compound', 'cookie sandwich', [('cookie', 2), ('ice cream scoop',  
3)]),  
    ('compound', 'cookie', [('chocolate chips', 3)]),  
    ('compound', 'cookie', [('sugar', 10)]),  
    ('atomic', 'chocolate chips', 200),  
    ('atomic', 'sugar', 5),  
    ('compound', 'ice cream scoop', [('vanilla ice cream', 1)]),  
    ('compound', 'ice cream scoop', [('chocolate ice cream', 1)]),  
    ('atomic', 'vanilla ice cream', 20),  
    ('atomic', 'chocolate ice cream', 30),  
]
```

Given the definition above, what should be the result of calling `all_flat_recipes(cookie_recipes, 'cookie sandwich')`? Enter your answer as a Python list below:

What should be the result of calling `all_flat_recipes(cookie_recipes, 'sugar')`? Enter your answer as a Python list below:

What should be the result of calling `all_flat_recipes(cookie_recipes, 'cookie sandwich', ('sugar', 'chocolate ice cream'))`? Enter your answer as a Python list below:

What should be the result of calling `all_flat_recipes(cookie_recipes, 'cookie sandwich', ('cookie',))`? Enter your answer as a Python list below:

## 7.1) Generating Combinations of Flat Recipes

One tricky piece of implementing `all_flat_recipes` is that it requires us to generate combinations of different recipes for each of the subingredients of a compound ingredient. We're going to break this problem down by extracting out this combination task into a helper function called `combined_flat_recipes(flat_recipes)`.

`combined_flat_recipes` should take in a list of lists of flat recipes, where each list inside `flat_recipes` represents all the flat recipes for a single ingredient, and it should return every possible combination of recipes, picking one recipe from each list in `flat_recipes`. Importantly, this function should not mutate its input.

If this is challenging to think about, don't worry! Combinations (and similarly, recursion) can be a little perplexing even to experienced programmers, but going over some examples should help.

For example, say we had a recipe book of spreads:

```
book = {'savory': [[('peanut butter', 1)], [('almond butter', 1)]],
'sweet': [[('jelly', 2)]]}
```

To find all combinations of savory and sweet spreads, we could flatten those ingredients and then compute the following:

```
combined_flat_recipes(  
    [{ 'peanut butter': 1}, { 'almond butter': 1}],  
    [{ 'jelly': 2}])
```

which would return a list of flat\_recipes containing:

```
[{ 'peanut butter': 1, 'jelly': 2}, { 'almond butter': 1, 'jelly': 2}]
```

which represents all possible combinations of savory and sweet spreads.

Or say we're trying to combine together a cake with some icing and some toppings. Let's start with just the cake, which has two possible flat recipes, regular and gluten-free.

If we have

```
cake_recipes = [{"cake": 1}, {"gluten free cake": 1}]
```

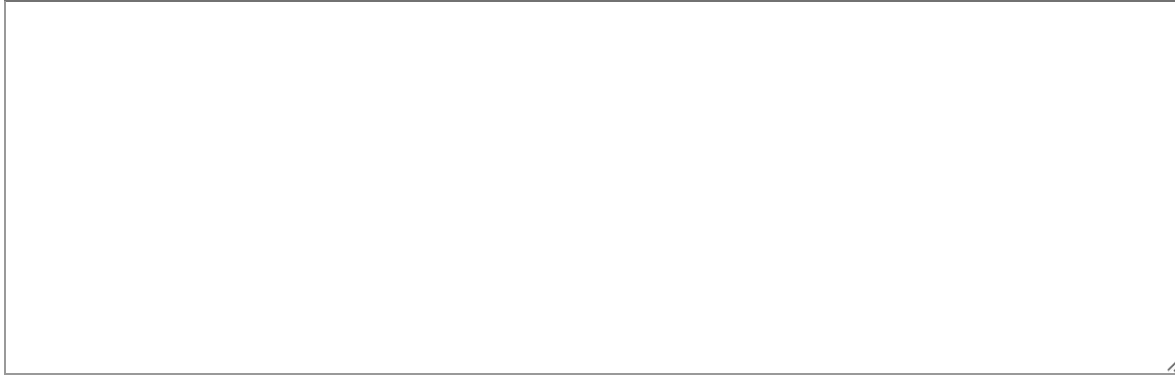
what should the result of `combined_flat_recipes([cake_recipes])` be?

Now, let's make it interesting and add some icing with two different flavors.

If we now have

```
icing_recipes = [{"vanilla icing": 1}, {"cream cheese icing": 1}]
```

what should the result of `combined_flat_recipes([cake_recipes, icing_recipes])` be? If this is tricky, remember we're trying to get all possible combinations of cake and icing.

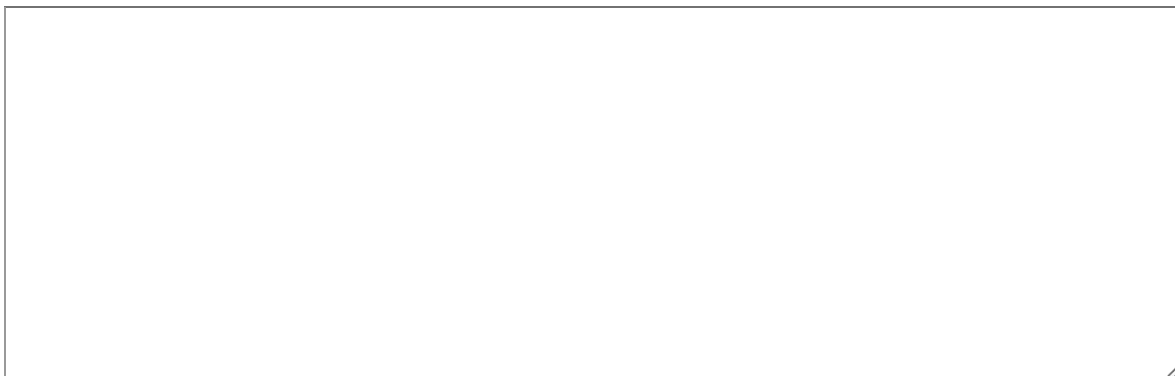


Every cake deserves a topping! Unfortunately, all we have is sprinkles.

If we add

```
topping_recipes = [{"sprinkles": 20}]
```

what should the result of `combined_flat_recipes([cake_recipes, icing_recipes, topping_recipes])` be?



Now it's time to go ahead and implement the `combined_flat_recipes` and `all_flat_recipes` functions in your `lab.py` file. As you do so, don't forget about the `scaled_flat_recipe` and `add_flat_recipes` functions we wrote earlier! Go back to the [Flat Recipe Helpers](#) section if you forgot what they do. Again, while we suggest that using the helper functions may be useful, you are free to implement `all_flat_recipes` however you wish.



At this point, your code should pass all of the tests in `test.py`! Congrats on finishing the first 6.101 recursion lab! Nice work!

## 8) Summary

In this lab, we have been working with data structures representing recipes. These structures have an interesting property: they are *nested* to an arbitrary depth (recipes may depend on other recipes, which may depend on other recipes, to an arbitrary degree). Recursion gives us an elegant way to deal with this kind of nested data. We hope that this lab has given you some useful practice with recursion, and we'll get more practice with it over the next several weeks as well!

## 9) Code Submission

When you have tested your code sufficiently on your own machine, submit your modified `lab.py` using the `6.101-submit` script.

The following command should submit the lab, assuming that the last argument `/path/to/lab.py` is replaced by the location of your `lab.py` file:

```
$ 6.101-submit -a recipes /path/to/lab.py
```

Running that script should submit your file to be checked. After submitting your file, information about the checking process can be found below:

When this page was loaded, you had not yet made any submissions.

If you make a submission, results should show up here automatically; or you may click [here](#) or reload the page to see updated results.

## 10) Checkoff

Once you are finished with the code, you will need to come (in-person) to any open lab time and add yourself to the [queue](#) asking for a checkoff in order to receive credit for the lab. **You must be ready to discuss your code in detail before asking for a checkoff.**

You should be prepared to demonstrate your code (which should be well-commented, should avoid repetition, and should make good use of helper functions; see the notes on [style](#) for more information). In particular, be prepared to discuss:

- Discuss how you implemented the `lowest_cost` function.
- Discuss how you implemented the `cheapest_flat_recipe` function.
- Discuss how you implemented the `combined_flat_recipes` function.
- Discuss how you implemented the `all_flat_recipes` function.

*You have not yet received this checkoff. When you have completed this checkoff, you will see a grade here.*

---

## Footnotes

<sup>1</sup> These recipes are only for example purposes. 6.101 isn't a cooking class, so please look up some real recipes before cooking.