Discussion Activity: Pantries and Cooking

Group Names and Roles

- David (driver)
- Lauren (proposer)
- Rashi (reviewer)

In this activity, we'll create a class for managing a pantry of food ingredients. This class will interact with the `recipe` class from a previous worksheet, allowing us to check whether we have enough ingredients to make the desired recipe. We'll also be able to add ingredients to the pantry, representing "going shopping." Schematically:

$$\text{go shopping} \implies \text{ingredients in pantry} \implies \text{cook recipes}.$$

Part A

A `Pantry` is a subclass of `dict` that supports addition (with `dict` s) and subtraction (with `recipe` s). The code below implements a simple `Pantry` class with entrywise addition. If this code looks a bit familiar, that's because it is! This is just a rebranded `ArithmeticDict` from the first lecture on inheritance (https://nbviewer.jupyter.org/github/PhilChodrow/PIC16A/blob/master/content/object_oriented_programming/inheritance_I.ipynb).

Run this block.

Note: *In a more thorough implementation of `__add__()` and subsequent methods, we would do input checking to ensure that we are dealing with dictionaries with integer or float values. Because we've already practiced input checking when we wrote the `recipe` class, we're not going to worry about that again here.*

In [33]:
```
# run this block

# used for warning for low ingredients (Part E)
import warnings

class Pantry(dict):
    """
    A dictionary class that supports entrywise addition.
    """

    # supplied to students
    def __add__(self, to_add):
        """
        Add the contents of a dictionary to_add to self entrywise.
        Keys present in to_add but not in self are treated as though
        they are present in self with value 0.
        Similarly, keys present in self but not in to_add are treated
        as though they are present in to_add with value 0.
        """
        new = {}
        keys1 = set(self.keys())
        keys2 = set(to_add.keys())
        all_keys = keys1.union(keys2)

        for key in all_keys:
            new.update({key : self.get(key,0) + to_add.get(key,0)})

        return Pantry(new)

    # implement subtraction in Part B here
    def __sub__(self, recipe):
        new = {}
        keys1 = set(self.keys())
        keys2 = set(recipe.ingredients.keys())
        all_keys = keys1.union(keys2)

        for key in all_keys:
            new.update({key : self.get(key,0) - recipe.ingredients.get(key,0)})

        return Pantry(new)
```

Let's say that we'd like to make some delicious chocolate chip cookies. But wait -- we don't have any chocolate chips in our pantry! (Run this block):

In [35]:
```
my_pantry = Pantry({"flour (grams)"  : 2000,
                    "sugar (grams)"  : 1000,
                    "butter (grams)" : 500,
                    "salt (grams)"   : 1000})
```

In the code cell below, use addition to add to your pantry. To do so, first make a `dict` called `grocery_trip` in which you buy:

- 1000 grams of flour
- 500 grams of butter
- 500 grams of chocolate chips
- 2 onions

The format should be the same as `my_pantry`. For example, `grocery_trip` might begin like this:

```
grocery_trip = {
    "flour (grams)" : 1000,
    ...
}
```

Then, add the contents of `grocery_trip` to `my_pantry`. Check the result to ensure that it makes sense.

In [36]:
```
# your solution here
grocery_trip = Pantry({"flour (grams)"  : 1000,
                       "butter (grams)" : 500,
                       "chocolate chips (grams)" : 500,
                       "onions" : 2})
print(my_pantry)

my_pantry += grocery_trip
print(my_pantry)
```

```
{'flour (grams)': 2000, 'sugar (grams)': 1000, 'butter (grams)': 500, 'salt (grams)': 1000}
{'sugar (grams)': 1000, 'salt (grams)': 1000, 'onions': 2, 'flour (grams)': 3000, 'butter (grams)': 1000, 'choc
olate chips (grams)': 500}
```

Part B

Here is solution code for the `Recipe` class from last time. To simplify the code, we have removed the input checking in the `__init__` method, as well as the `__str__` method.

In [37]:
```
class Recipe:

    def __init__(self, title, ingredients, directions):
        self.title       = title
        self.ingredients = ingredients
        self.directions  = directions

    def __rmul__(self, multiplier):
        multiplied_ingredients = {key : multiplier*val for key, val in self.ingredients.items()}
        return recipe(self.title, multiplied_ingredients, self.directions)
```

Now, implement **subtraction** in which the first argument is a `Pantry` and the second argument is a `Recipe`. The relevant magic method for this is called `__sub__()`, and should be implemented in the `Pantry` class. Here's how subtraction `my_pantry - my_recipe` should work:

1. You may assume that `my_pantry` and `my_recipe` are valid instances of their class. In particular, all quantities of ingredients are positive numbers (`int`s or `float`s).
2. If all keys from `my_recipe.ingredients` are present in `my_pantry`, and if they all have values smaller than their values in `my_pantry`, then the result of `my_pantry - my_recipe` is a new `Pantry` object in which the values corresponding to the keys have been reduced by the quantity in `my_recipe`.
3. If a key is present in `my_pantry` but not in `my_recipe`, then it is treated as though it is present in `my_pantry` with value `0`.

For now, you can assume that the conditions of clause 2. are met, and that subtraction should therefore "work." That is, you can assume that you have enough of all ingredients in the pantry to make the recipe. For example, with `my_pantry` from Part A,

```
title = "cookies"
ingredients = {
    "flour (grams)"  : 400,
    "butter (grams)" : 200,
    "salt (grams)"   : 10,
    "sugar (grams)"  : 100
}

# Great British Baking Show-style directions
directions = ["make the cookies"]

cookies = Recipe(title, ingredients, directions)
my_pantry - cookies
```

```
{'salt (grams)': 990,
 'flour (grams)': 2600,
 'butter (grams)': 800,
 'chocolate chips (grams)': 500,
 'sugar (grams)': 900,
 'onions': 2}
```

You can implement subtraction by modifying the code block in Part A -- no need to copy/paste your class.

Hint: *Dictionary comprehensions* provide a convenient way to make new dictionaries from old ones. Their syntax is related to list comprehensions. For example:

```
d = {"shortbread cookie" : 2, "chocolate chip cookie" : 1}
{"tasty " + key : val for key, val in d.items()}
```

Hint: The method `dict.get()` will let you specify a "default" value in a dictionary, returned when a key is not found. For example,

```
{"cinnamon cookie" : 1, "florentine cookie" : 1}.get("brownie", 0)
```

will return value 0 because the key `"brownie"` is not found.

In [38]:
```
# test your solution here
title = "cookies"
ingredients = {
    "flour (grams)"  : 400,
    "butter (grams)" : 200,
    "salt (grams)"   : 10,
    "sugar (grams)"  : 100
}

# Great British Baking Show-style directions
directions = ["make the cookies"]

cookies = Recipe(title, ingredients, directions)
print(cookies.ingredients)
print(my_pantry)
my_pantry -= cookies

print(my_pantry)
```

```
{'flour (grams)': 400, 'butter (grams)': 200, 'salt (grams)': 10, 'sugar (grams)': 100}
{'sugar (grams)': 1000, 'salt (grams)': 1000, 'onions': 2, 'flour (grams)': 3000, 'butter (grams)': 1000, 'choc
olate chips (grams)': 500}
{'sugar (grams)': 900, 'salt (grams)': 990, 'onions': 2, 'flour (grams)': 2600, 'butter (grams)': 800, 'chocola
te chips (grams)': 500}
```

Part D

Now, handle the case in which `my_recipe` contains a key not contained in `my_pantry` , or in which case the associated value in `my_recipe` is larger. This models the situation in which your recipe requires an ingredient that you don't have, or that you don't have in sufficient quantity.

In this case, `my_pantry - my_recipe` should raise an informative `ValueError` , stating which ingredients need to be added in order to make the recipe. For example, here's `my_pantry` as it was at the end of Part A.

```
my_pantry = Pantry({'salt (grams)': 1000,
                    'flour (grams)': 4000,
                    'butter (grams)': 1500,
                    'chocolate chips (grams)': 1000,
                    'sugar (grams)': 1000,
                    'onions': 2})

title = "oatmeal cookies"
ingredients = {
    "oatmeal (grams)" : 200,
    "flour (grams)"   : 500,
    "sugar (grams)"   : 100,
    "raisins (grams)" : 200,
    "butter (grams)"  : 200
}
directions = ["make the cookies"]

my_recipe = Recipe(title, ingredients, directions)

my_pantry - my_recipe
```

In this case, an informative `ValueError` should be raised indicating which ingredients are ndeed, and in what quantities.

Hints

Implementing this behavior efficiently can be a bit challenging. Here are some suggestions:

1. The command `set(L1).union(set(L2))` will create a set of all items in either `L1` or `L2`, where `L1` and `L2` are lists or other iterables.
2. I found it helpful to first create a dictionary containing the full state of the pantry after subtraction, with negative numbers allowed. I then checked to see whether there were any negative numbers in this dictionary, raising the `ValueError` if so.

In [ ]:

Part E

If you have completed Parts A-D and still have some time, do one of two things:

1. Implement an `__str__()` method for your class to enable attractive printing.
2. Modify the subtraction method so that a *warning* is shown when subtraction results in an ingredient running low. For the purposes of today, let's say that "low" means that there are fewer than 100g of the ingredient in the pantry.

To issue warnings, you need to `import` the `warnings` module and then call `warnings.warn` with the text of the warning.

```
import warnings
# ... other code
warnings.warn("Uh oh! These ingredients are running low.: ... ")
```

In [ ]:
```
# test your new features here
```