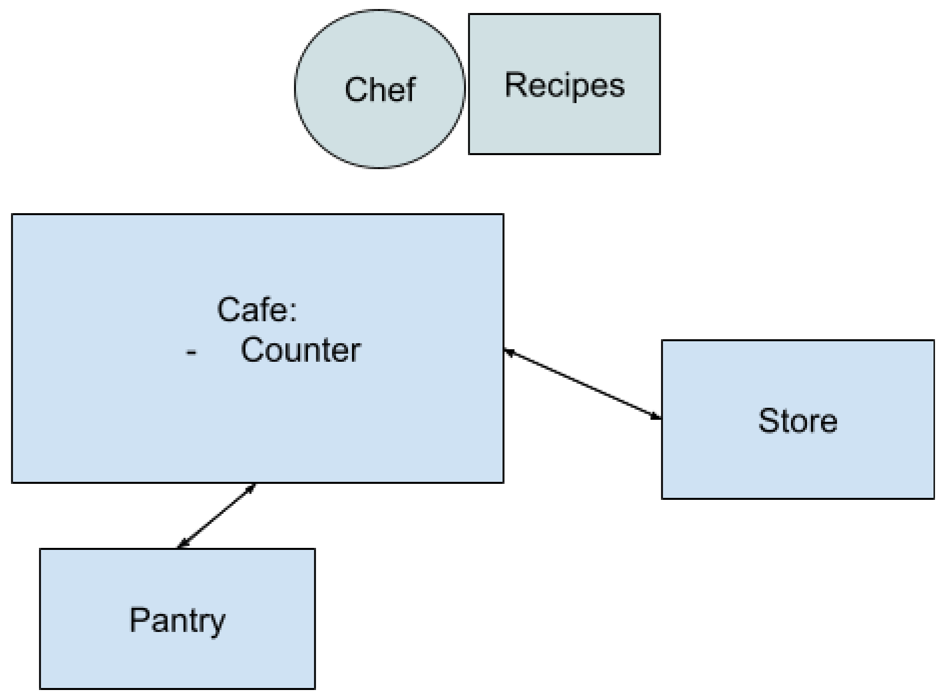
Idea:

The user is a chef and they need to feed customers that come into the restaurant. The chef has a pantry, a recipe book, and they can also go to a store to buy groceries.



Game model:

- Customers come at an interval (default 45 seconds)
- Chef's goal is to serve these customers and earn money
- Currently: game is set to end when the chef has served 5 people

Chef and States and their (un)available actions:

Notes:

- There are several dicts that are used here that are basically bags or multisets. When we remove an item from one of these bags, it just decreases its count or removes it. Similarly, when we add an item to one of these bags, it just increases the count or adds it.
  - Types of bags below:
    - Chef inventory
    - Cafe counter
    - Pantry

- Some actions have invalidity, meaning the user typed an invalid thing after a keyword. It passes all the lexing, but when evaluating, if invalid, then an appropriate message will be printed and nothing will happen
- If “exit” is typed then the program terminates
- Some actions will return a string that represents the game state that the game should be in after the action. I write “goes to [some state]” below. No changing of states will return nothing and I don’t explicitly say it below.
- One unchecked possibility of the game is when the user doesn’t have enough money to buy more ingredients and then they can’t do anything to make more money. When this happens, they should just exit the game.

Help command:

If the user types in “help”, then all commands available at current state will be displayed along with its description.

Chef:

Chef (user) always has these attributes no matter what state they are in; they can do these actions whenever as well:

- Inventory: has capacity of amount of ingredients that can be in it
    - Actions:
        - **ls**: lists inventory on screen; no states changed
- Recipe book: chef always has recipe book and can always look at it
    - Actions:
        - **recipes**: goes to recipe state
- Wallet
    - Actions:
        - **wallet**: checks wallet for money

States:

Notes: These classes extend a parent class called State which contains a method called do() that all these classes implement. This is for dynamic dispatch which is touched on later.

- Cafe:
    - What there is (*useful* instance variables):
        - dict of food on counter and amount (has capacity), list of customers with orders that come in randomly, list of completed foods
        - ‘Making’ related:
            - Food currently making (“” for not making)
            - Making step: keeps track of which step in the recipe file that the user is on
            - list of used ingredients
    - Actions at this state:
        - **make** [food]: starting command to make/start an order

            - After this command, the ‘making’ variable is set to the food
            - Invalid:
                - If food is not in recipe book
        - **serve** [food] **to** [customer]: customer is served and evaluates a score based on time and random taste and gives money
            - Invalid:
                - If food is not made
                - If customer is not in restaurant
                - If customer didn’t order food
        - **orders**: lists the orders (customer name, food they want)
        - **lc**: lists ingredients on the counter (max 10 ingredients (default))
        - **set** [ingredient]: sets ingredient from inventory to counter
            - Invalid:
                - if no space on counter
                - If ingredient is not in inventory
        - **clear** [ingredient]: clears ingredients from counter to inventory
            - Invalid:
                - if no space in inventory
                - If ingredient is not on counter
        - **completed**: lists the completed foods

        - If making some food, here are additional actions:
            - [verb] [ingredient]: the verb is the cooking verb in the recipe step; the ingredient is the part after the cooking verb; more on how recipe files are store are down below
                - Invalid:
                    - If current making step is incorrect
            - **stop**: stops making and resets making step; now these sub-actions cannot be done
        - If the verb is incorrect but the ingredient is one that’s on the counter, then that ingredient will be removed from the counter because you ‘verb’ed it
    - Actions to change states
        - **open pantry**: goes to Pantry state
        - **go to store**: goes to Store state
    - Note: Sort of works alongside Chef — a lot of chef methods and variables used here
- Pantry:
    - What there is (*useful* instance variables):
        - dict of ingredients and amount (infinite space)
    - Actions at this state:
        - **store** [ingredient]: stores ingredient from inventory into pantry
            - Invalid:
                - If ingredient is not in inventory
        - **get** [ingredient]: gets ingredient from pantry to inventory

            - Invalid:
                - If ingredient is not in pantry
                - If no space in inventory
        - **look**: lists the pantry
    - Actions to change states:
        - **close**: simulates closing pantry and automatically goes back to cafe state
    - Note: Pantry will contain defaults if the default file is specified. Set to 999
- Recipe:
    - What there is (*useful* instance variables):
        - List of recipes with number, ingredients, and steps
    - Actions at this state:
        - **flip**: displays next page in the recipe book
        - **back**: displays previous page in the recipe book
    - Actions to change states:
        - **close**: simulates closing recipe book and automatically goes back to the state that opened the book
    - Notes: the recipe is circular so flipping from the end will go to the front and flipping back at the front will go to the end
- Store:
    - What there is:
        - Dict of ingredients available in the store and its price (infinite space in store), dict of shopping cart that contains ingredients and count
    - Actions at this state:
        - **price** [ingredient]: generates a price for new ingredients; that price stays the same throughout the whole game since it's added to the prices dict
        - **take** [ingredient]: adds ingredient to imaginary shopping cart
            - Invalid:
                - If ingredient is not in prices dict
                - If the added ingredient exceeds the inventory capacity after checking out the cart
                - If the added ingredient causes the total amount in the cart to be greater than the wallet
        - **return** [ingredient]: returns an ingredient from cart (basically just removes the ingredient from the cart)
            - Invalid:
                - If the ingredient is not in the cart
        - **prices**: displays all the ingredients available in the store and their prices
        - **cart:** displays items in the cart along with the prices and the total price
    - Actions to change states:
        - **checkout**: simulates checking out and returning to restaurant; goes back to restaurant state with all ingredients now in inventory; wallet is decremented as well
    - Clarification Note: The 'take' action automatically checks if the ingredient to be added is 'viable.' This makes it so that checking out is always good, meaning

you'll never checkout a cart that has too many ingredients for your inventory or that is too expensive

Other Classes:

Cafegame Class:

- Wraps all the classes mentioned above.
- Responsible for performing actions by these classes depending on the state through dynamic dispatch. Gets the return values and switches states if needed

Customer Class:

Useful instance variables: name, order, arrival

When a customer is created, it will need to call functions to create its name and order based on the given name file and available foods to order. The arrival time is also set.

The Cafe has the function that generates random Customer's. However it is called on by a separate thread from the driver.py file to keep track of intervals between customer arrivals.

Driver Class:

- Creates the Cafegame
- Displays start and end messages
- Produces the REPL
- Creates the thread that adds new customers periodically

Modes: "testing" or "timed"

- I didn't implement this before class presentation, but the modes are passed into the constructor
- "testing":
    - Non threaded
    - New customers arrive after each dish is completed
- "Timed" (default):
    - Threaded
    - New customers arrive every 45 seconds (default) even if user hasn't finished

Important: As the top level file, the main function code that creates the game is here too. Any defaults that want to be overridden are to be set in the constructor.

utils.py:

- Contains utility functions mainly for operations involving bags/multisets

- Used throughout each class

Defaults: (can be adjusted in driver.py file)

Chef:

- Max inventory: 5
- Wallet: 15$

Recipe:

- Recipes file: “defaults/recipes.txt”

Cafe:

- Max counter: 10
- Customer names file: “defaults/customer_names.txt”

Pantry:

- Pantry defaults file: “defaults/pantry.txt”

Customer:

- Customer interval (how often do customers arrive at the cafe): 45
- Customer count (how many customers will arrive): 5

Other:

- Launch screen file: “defaults/launch_screen.txt”

File Formatting:

Note: If any files are not formatted correctly, game will not start

Customer names file:

- Any string of lowercase characters on 1 line
- No leading or trailing whitespace
- Can have whitespace in between but that will just count as part of the the name
- Separated by new line
- Doesn’t matter if ends with new line or not

Pantry default file:

- Same as customer names file

Recipes file:

a. “name: {name of recipe/food}”
b. “ingredients: {list of ingredients}”
    i. Ingredients can be 1 or if more then separated by “, “
c. “steps:”
d. “{step number}. {cooking verb} {ingredient to use}
    i. Step number should start at 1 and increment for every step
    ii. Cooking verb is one word
    iii. Only one ingredient can be used
    iv. All the ingredients from the list in (b) must be included at least once in the steps
        1. Note: there can be an ingredient in the step that is not in the ingredient list from (b) — since it can be from the default pantry ingredients
e. New line indicates this recipe is done and then repeat steps a-e if there are more recipes

In easier terms and other things

- Everything in the {} is lowercase alphabetical characters.
- Must have at least one ingredient
- Must have at least one step
- It is possible where someone lists an ingredient and then in the steps, it uses an ingredient that is not listed and not in the pantry either
- No leading or trailing whitespace
- Can have whitespace in between but that will just count as part of the the name
- Must have exactly 1 new line after each recipe, including at the end of the file.

What I wish I added:

- Better economy system:
    - Formula for calculating ingredient prices (maybe based on length of ingredient)
    - Food items have a price already and customer adds by tipping
    - Formula for calculating tips that each customer gives (some inverse function based on time)
- Randomize intervals of customers
    - Maybe start slow then progressively get more and more customers
- An ‘all’ keyword to add after some actions to do more at once (set all, store all,...)
- Better win condition
- Add command line arguments to replace default files better
- Fix the problem where if a user doesn’t have money to buy ingredients then they are stuck:
    - Create a checker when the wallet gets below a certain amount and exits game automatically
    - Create a side quest to earn some quick money