ItoP: Assignment 2

Introduction

In this assignment you will build part of an event-driven simulator based on the video game Overcooked. Your main goal will be to fill in and write new callback functions to handle various events that occur in the kitchen simulation. A simulation of this style can be used to determine how to efficiently manage resources.

Context

In Overcooked, each player controls a chef working in a kitchen to complete orders. At random intervals, new orders come in to the kitchen. The players must coordinate with each other to prepare the ingredients in the order. Each ingredient takes some steps to completely prepare. For example, to prepare lettuce the player must grab a piece of lettuce from the storage box, place it on an available cutting board, chop up the lettuce, and then put it on the burger.

For this assignment, we will model a simplified version of Overcooked. When an order comes in, you must make sure the kitchen has enough available ingredients to complete the order. If so, you will prepare each ingredient. There are only a limited number of chefs available, and each chef can only prepare one ingredient at a time. If no chefs are available to prepare a certain ingredient, you must try again after a short period of time.

Directions

Fill in and write functions in `as2.py`. See Documentation for functions I have provided for your use. **In the following section, every time I mention 'printing' something, you should be calling the provided `print_time` function.**

Instructions continued on the next page

`handle_new_order` event

This function is the entry point for all orders and will create new events in the simulator for each ingredient in the order. The following is the high-level control flow for this function:

1. Check if there are enough ingredients to complete the order.
    - If there are, print out `Accepting order` and increment the accepted order count in the simulator.
    - If there are not, print out `Rejecting order` and increment the rejected order count in the simulator. Then return from the function.
2. For each ingredient in the order, check if there is a chef available to begin preparing the ingredient.
    - If there is, create a new event at `time` to begin preparation of the ingredient.
    - If there isn't, create a new event at `time + 1` to retry preparation of the ingredient, in case a chef is available at that point.

You must process ingredients from first (index 0) to last (index N) in the recipe.

Ingredient preparation

When you start preparing an ingredient, you must first allocate a chef for the task. Each ingredient takes a certain amount of time to prepare. The chef you have allocated for the task will be occupied for the entire preparation time and will not be able to switch to another task. Once the preparation is complete, you should free up the chef so that he/she is available to prepare other ingredients.

When you begin preparing an ingredient, print out `Starting to prep <ingredient>`. `<ingredient>` should be replaced with the name of the ingredient (e.g. `Starting to prep burger`). When you are done preparing an ingredient, print out `Done prepping <ingredient>`.

Retrying

Once an order has been accepted (the only criteria for acceptance is the number of available ingredients), it must be completed at some point. However, there may not be enough chefs available at the time to immediately begin working on the order. As such, there is a notion of retrying preparation. If an ingredient needs to be prepared and there are no chefs available, you must retry preparation in the next time-step (i.e. `time + 1`).

Note that a retry event is just an *attempt* at retrying. It is possible that in the next time-step there still will not be a chef available. If that's the case you must retry again.

Hint: The actual code for retrying is very concise and straightforward, if you approach it from the perspective of events. The retry functionality in my solution is only 1 line.

Instructions continued on the next page

Other functions and events

You will need to create functions to handle events in the simulator. One such function may be to begin preparing an ingredient. You will most likely need to write other functions as well.

Hint: How do you free up a chef after a certain amount of time?

Think carefully about what events and functions you need, as well as where you perform your checks for available chefs and ingredients. You may accidentally create broken situations. The previous sections describe the expected functionality, but the actual code structure may look drastically different (i.e. each section doesn't necessarily correspond to a function or event).

`setup_simulation` function

This function is not an event, but will be called one time at the beginning of the program - before simulation starts. This is where you will define the number of chefs and ingredients available. You should experiment with these values when testing your program.

Starter files and running your program

The starter files can be downloaded from Canvas as a file called `as2.zip`. There are five files in the zip: `as2.py`, `simlib.py`, `main.py`, `orders.csv`, and `README.txt`. You will write all of your code in `as2.py` where the comments direct you to. You should also fill out the information in `README.txt`.

`orders.csv` contains the series of orders that your simulation will have to process. Each order will be on a new line. The first comma-separated entry on a line is the time at which the order will be placed. The remaining comma-separated values on the line indicate the ingredients in the order. The provided `orders.csv` has two orders, one at time 0 and one at time 10.

To run the program, type in the command '`python3 main.py orders.csv`'. The second argument, `orders.csv` can be replaced with any file that follows the same format as the provided `orders.csv`. You should create multiple files when testing your program.

Instructions continued on the next page

Sample run

Once you have completed the assignment, you will get a result as shown below if you use the configuration in the starter files. The first section is a list of events that the simulator is handling. The second section is a report that summarizes the efficiency of the kitchen.

```
0: Accepting order
0: Starting to prep burger
0: Starting to prep bun
5: Done prepping bun
5: Starting to prep tomato
7: Done prepping tomato
8: Done prepping burger
10: Accepting order
10: Starting to prep burger
10: Starting to prep tomato
12: Done prepping tomato
12: Starting to prep lettuce
14: Done prepping lettuce
14: Starting to prep bun
18: Done prepping burger
19: Done prepping bun
-----------------
- Kitchen had 2 chef(s) and the following ingredient counts:
  - Burger: 8
  - Lettuce: 7
  - Tomato: 13
  - Bun: 18
- Accepted orders: 2 (100.00%)
- Rejected orders: 0 (0.00%)
- Total time: 19 minute(s)
```

Due to the way the event scheduler works, the exact sequence of events may differ between runs (although unlikely), exact Python versions, or machines. However, the report at the bottom, which is provided in the shell code, should be consistent. As such, I will be grading the report portion only.

If you believe your event ordering is correct but the report differs, please email me at chirag.sakhuja@utexas.edu or post on Piazza.

Instructions continued on the next page

Important Notes

- You may assume all inputs will be valid as described by this document.
- Your output must match the sample exactly, down to the spaces and capitalization, to receive credit.
- In my solution I added 2 functions and around a total of 25 lines of code. Your solution may be very different, but don't overthink it. Try to gain an understanding of how event-driven simulation works, and the code for this assignment will fall into place easily.

Submission

You should submit a zip file to Canvas called `as2.zip` with the same structure as the starter files. **The following conditions will result in a 0 on this assignment:**

- `README.txt` is incomplete or incorrectly filled out.
- The structure of the submitted `as2.zip` differs from the original structure.
- The output, including spaces and capitalization does not match the sample run **exactly**.
- The program does not terminate in a reasonable amount of time.

Documentation

`Simulation` interface

Although, we haven't talked about Object-Oriented programming in Python yet, you will be interacting with a single object for this assignment: `Simulation`. All you need to know about OOP for now is that member methods are called on objects the same way in Python as they are in C++ - using the dot operator. Each of these methods can be accessed in `as2.py` by using the `sim` object (e.g. `sim.use_item('burger')`).

`add_ingredient(ingredient, count, prep_time)`
Sets up the simulation so that there are `<count>` `ingredients` in the kitchen, each of which takes `prep_time` to prepare.

- `ingredient` (string): the name of the ingredient
- `count` (int): the number of this ingredient
- `prep_time` (int): the time it takes to prepare this ingredient

`set_available_chef_count(count)`
Sets up the simulation so that there are `<count>` chefs available in the kitchen.

- `count` (int): the number of chefs available

Instructions continued on the next page

`get_count(ingredient)`
Returns the remaining count of `ingredient`.

- `ingredient` (string): the ingredient you are checking the count for

`use_ingredient(ingredient)`
Consume one of `ingredient`.

- `ingredient` (string): the ingredient you are consuming

`assign_chef()`
Allocates a chef for a task (i.e. decrements the count of available chefs by 1).

`dismiss_chef()`
Frees up a chef from a task (i.e. increments the count of available chefs by 1).

`get_available_chef_count()`
Returns the number of chefs that are available to be assigned.

`get_prep_time(ingredient)`
Returns the prep time for `ingredient`.

- `ingredient` (string): the ingredient you are checking the prep time for

`increment_accepted_order_count()`
Increments the number of accepted orders by 1.

`increment_rejected_order_count()`
Increments the number of rejected orders by 1.

`schedule_event(time, func, *args, **kwargs)`
Schedules an event (which is the function `func`) to happen at time `time`. Accepts variadic arguments that will be forwarded to the `func` call.

- `time` (int): the time at which to schedule the event for
- `func` (function): the function that will be called when the event is ready to be processed
- `*args`: extra positional arguments that will be forwarded to the function call (optional)
- `**kwargs`: extra keyword arguments that will be forwarded to the function call (optional)

Instructions continued on the next page

`as2.py` functions

`print_time(time, msg)`
Prints a formatted string with a timestamp and message.

- `time` (int): the timestamp for the message
- `msg` (string): the message

`setup_simulation(sim)`
Sets up simulation parameters for the `Simulation` object.

- `sim` (Simulation): the simulation object to setup simulation parameters for

Event functions
Event functions, such as the provided `handle_new_order`, must always start off with two positional arguments: `time` and `sim`. After that, there can be any number of positional and keyword arguments that you may find useful.

- `time` (int): the time at which the event was triggered
- `sim` (Simulation): the simulation object that the event handler will modify

Extensions

The following are optional, ungraded extensions. The purpose is to help expand your understanding of Python beyond the base assignment. The difficulty of each extension is indicated by the number of ☠ icons (max of 5), relative to the material we have covered in lecture so far. **All extensions you choose to submit should be in a separate directory so that they do not interfere with the grading of the base assignment.** When I run the assignment that is turned in, the behavior should match that of the base assignment, not of any extensions described.

Keeping track of orders (☠)

The behavior of this extension is simple to describe, but requires some understanding of Python memory management to implement. All you must do is print out `Order complete` when the last item in an order has been prepped. The sample run output is modified on the next page to demonstrate this extension (changes in bold). While you may implement this extension however you like, keep in mind how Python lists are stored and what a reference to a list means. Global state, unless absolutely required, is the antithesis of code readability.

Instructions continued on the next page

```
0: Accepting order
0: Starting to prep burger
0: Starting to prep bun
5: Done prepping bun
5: Starting to prep tomato
7: Done prepping tomato
8: Done prepping burger
8: Order complete
10: Accepting order
10: Starting to prep burger
10: Starting to prep tomato
12: Done prepping tomato
12: Starting to prep lettuce
14: Done prepping lettuce
14: Starting to prep bun
18: Done prepping burger
19: Done prepping bun
19: Order complete
-----------------
- Kitchen had 2 chef(s) and the following ingredient counts:
  - Burger: 8
  - Lettuce: 7
  - Tomato: 13
  - Bun: 18
- Accepted orders: 2 (100.00%)
- Rejected orders: 0 (0.00%)
- Total time: 19 minute(s)
```

Add more resource constraints (☠)

In the `setup_simulation` function, you can add ingredients to the kitchen. Add another structure to the `Simulation` class in `simlib.py` that keeps track of other resources, such as pans and cutting boards. Come up with logical constraints, such as a burger requiring a frying pan, and implement them in the simulator.

Add more types of events (☠)

Overcooked has a series of steps of preparation for each ingredient. Instead of grouping all those steps into a single `prep_time` variable, separate them out into individual events. For example, you may split up the preparation time for lettuce into two steps: wash the lettuce (1 minute) and then cut the lettuce (1 minute).

Instructions continued on the next page

Find optimal number of resources (☠☠)

This extension can be done alone but also meshes well with the Add more resource constraints (☠) and/or Add more types of events (☠) extensions.

Event driven simulations are useful in determining how useful resources are. For example, in the sample run for the base assignment it takes 19 minutes to complete the two orders with two chefs. With one chef, it takes 33 minutes. Adding another chef results in a 74% percent reduction in overall time, making the chef very cost-effective. Direct calculations are sometimes difficult to make, and so simulating with many test cases can be useful in determining efficiency.

Assign costs to each of the resources in the kitchen. This means hourly pay for the chefs, cost for ingredients, cost for orders, etc. You can make the model as complex as you would like. Then, run simulations of many orders and try to determine the most cost-effective combination. This can be formalized by using gradient descent or some other optimization technique (☠☠☠).

Replace handlers with coroutines (☠☠☠☠)

Coroutines are an extension to the concept of subroutines. Normally, a subroutine executes its code and then returns to the caller. A coroutine can additionally suspend in the middle of the coroutine and then resume with the same scope at a later time. Using coroutines to model producer-consumer relationships is arguably more intuitive than using subroutines. In this simulation, events are produced and then consumed by the event handlers you've written. Convert your event handler functions and the simulator backend to use coroutines. The output should be identical to the base assignment. If done well, you may end up with less lines of code after you've converted to coroutines!

Though not inherently difficult, understanding how to effectively use coroutines takes some time. A good starting point is going through this slide deck, which walks you through building an OS-style scheduler using coroutines in Python. Note that the slides use Python2, so you may need to convert some syntax.