Welcome to Carnegie Cookoff!

Hello! We are so excited to see you at Algorithms with a Purpose 2026, hosted by ACM@CMU. This document contains the game rules, a reference manual for your code, and a list of the in-game constants.

If you have any questions feel free to ask one of our AWAP staff at Office Hours or on Discord.

Getting Started

>> Important Links

- Viewer: https://www.acmatcmu.com/awap-viewer-2026/
- Dashboard: dashboard.awap.acmatcmu.com
- Discord: https://discord.gg/2dcBVWMw
    - If you weren't automatically assigned the @AWAP 2026 role, it can be claimed from #role-select. Scroll down or look in the channel list if you don't see it! We'll post an @everyone when the event starts as well.
    - We also encourage you to drop a cool message in #intro and follow your fellow community members :)
    - For this event, please join #awap-2026 for announcements, #awap-2026-general for public questions and memes, and #support for anything individual!

Development & Submission Flow

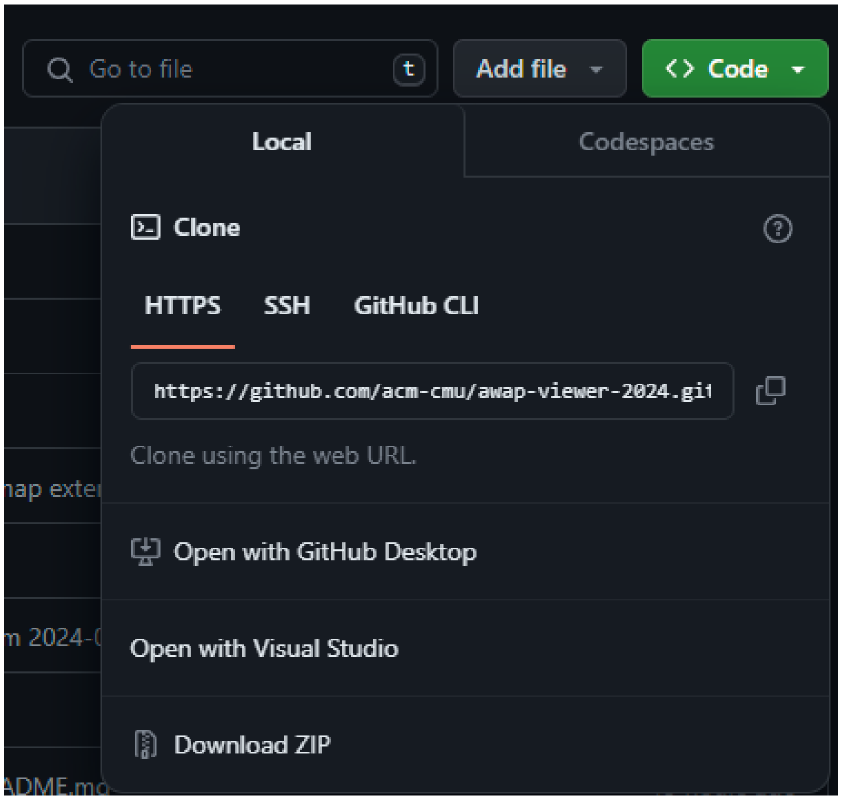
>> Local Development

1. Clone game engine at https://github.com/acm-cmu/awap-game-engine-2026-public
2. Write your bot and add it to the bots/ directory.
3. Put any maps you'd like to test with in the maps/ directory.
4. Specify players & maps in config.json then run 'python3 run_game.py -c config.json'. You may use the '--render' flag if you would like to see the game play out live. To save your replay in a JSON file, use the '--replay' flag, followed by the name of the file you would like to save it in (ex: '--replay sample.json')
5. After the replay file archive is created, you can view it again in your terminal by running 'python replay_game.py <file>.awap26r'

   a. A command-line view is available as well by running 'python replay_game_cli.py <file>.awap26r', which may be more compatible with browser-based editors.
   b. For both options, extracting the .gzip archive is optional.

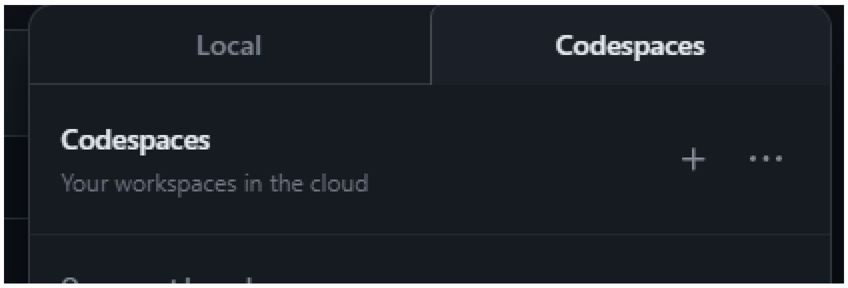
>> Using A Codespace

1. Upgrade your GitHub account for free with the Student Developer Pack! This will give your account Pro status and offers many other benefits. Sign up with your andrew.cmu.edu email after attaching it to your profile education.github.com/pack
2. Visit the repo you're interested in, like the game engine or viewer.
3. Click Code, and then switch from 'Local' to 'Codespace'



```
https://github.com/acm-cmu/awap-viewer-2024.git
```

Clone using the web URL.



4. Start a new Codespace! This is a virtual machine, with a VS Code interface. You can customize these with any extensions you're used to, and push commits to

your own fork if you'd like. After it starts up, follow the same instructions as local development (usually `npm install` and then `npm start` in the terminal interface), adding extensions like Python if necessary.
5. Just like developing locally, please `git pull` periodically so that we can deploy hotfixes if necessary. We'll let you know if so!
6. Similar to ACM@CMU HackCMU this past Fall, we'd recommend trying these out as this way all of our attendees can have the same development environment which makes debugging much easier.

>> Submission to Competition

1. Log into the dashboard at dashboard.awap.acmatcmu.com with the account username and password emailed to you
2. Upload your bot
    a. Request unranked scrimmages with other competitors or raffle bot
3. Review your match results
    a. Check match history
    b. Check leaderboard
    c. Download replay against other competitors

Game Overview

>> Background and Objective

You open your doors with a dream and a burner stove. Across the street, your rival does the same. Orders flood in. Tempers flare. Claws come out, teeth bared. The broth gets hotter. Make money. Complete orders. PURR-fect your noodles. Build your empire. WHEN NIGHT FALLS, sabotage your enemy and create a CAT-astrophe — but beware. But every hiss echoes back. When the dust settles, only one shop will still be standing. Welcome to the NOODLE War.

Each map represents a kitchen layout. You can cook, clean, and sabotage your opponent with various dishes.

Create a team of 2 bots to cook meals throughout the game. Stock up your kitchen with different foods, plates, and pans to cook with. You must keep your station clean as you fulfill the arriving orders. You can sabotage your opponents by throwing away their food into the trash.

>> Food Types

There are 5 different ingredients: eggs, onions, meat, noodles, and sauce. Each one of these items can be purchased from the shop, and some can be processed through chopping and cooking to make different dishes.

>> Actions

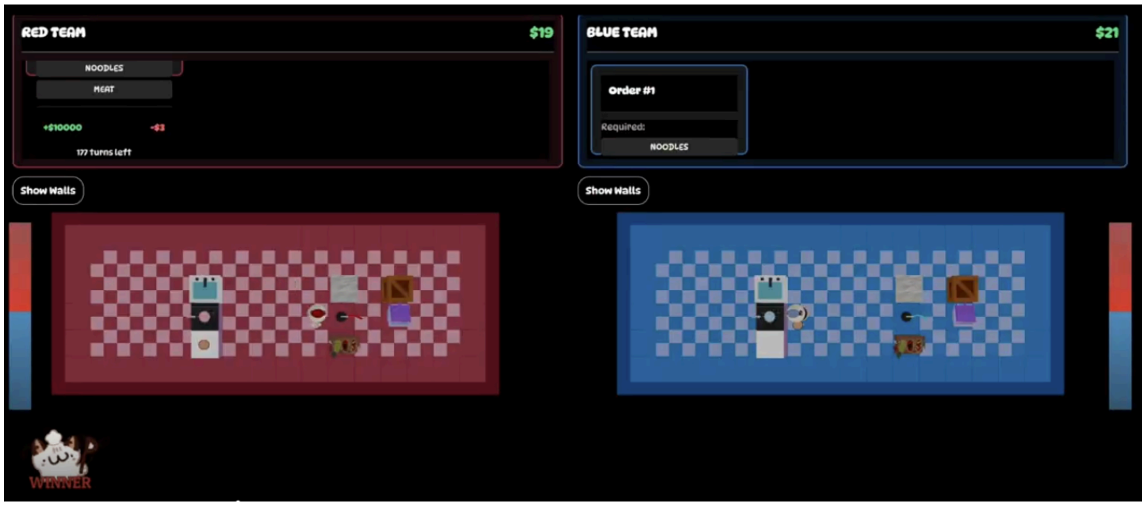
During the game, the player is able to access the current turn, their current team, their enemy team, the current map instance, and the current orders.

The bots themselves are able to perform the following actions:

- Move to specific coordinates
- Pick up items
- Place items
- Trash ingredients
- Purchase from the shop
- Chop certain ingredients
- Cook ingredients in a pan
- Wash and collect dishes
- Switch to the enemy map to sabotage them

>> Map Overview

The world is a grid with width and height dimensions that may range from 6 to 48. Each map has a kitchen with various orders arriving.



An example map with a kitchen containing a single sink, trash can, a box, counter, and submit location.

You can see all types of tile spaces on the Github.

>> Gameplay Details

Every team of two bots begins with 150 coins and needs to accumulate coins by fulfilling orders. There's passive money of 1 coin per turn. The map describes the kitchen layout. Your overall goal is to buy materials at the shop and cook food to submit at a submission station to earn as many coins as possible.

At a certain point in the game, you can choose to go to the other side and sabotage the opposing team, such as by throwing away their food!

Player code must run within a certain time limit. Each turn's code must run within 0.5 seconds.

Each turn, the following happens (in order):

- Each player gets passive income of 1 coin
- The cooking progresses by 1 tick
- An order may expire if it is has been untouched for too long
- You're moved back to your original side if you timed out while sabotaging the opponent
- Each player's code is called, allowing them to move and take actions

Per turn, each bot may move at most once AND perform at most one action. Information accesses (checking if a bot can move, game state accesses) are unlimited.

Plates are dirtied after they are used to submit an order and must be washed before they can be used again.

>> Tie Breaks

The team with the most coins at the end is deemed the winner. If both teams end with the same number of coins, they will be marked as draws (or redone if they occur during the ranked scrimmages).

>> Allowed Packages

- Python standard library
- Numpy

- Scipy
- Anything already imported by the game engine

All packages not listed are not allowed - please let us know if there are packages you want to use and we'll consider them; we're flexible on this.

API Documentation

You can find the GitHub repository for the game engine here:
https://github.com/acm-cmu/awap-game-engine-2026-public

This is an exhaustive list of all the functions or methods that players are allowed to access. If you have any remaining questions about the functionality, please reach out to us at in person office hours!

Note: Please do not attempt to exploit the game engine by accessing the internal state. Attempting to do so may result in disqualification.

RobotController

These are all the member functions of RobotController. These are used to control your player and obtain information about the game state.

RobotController - Main Functionalities

State Access Functions

`get_turn() -> int`

Gets the current turn of the game.

`get_team() -> Team`

Returns the team you are on, returning either Team.RED or Team.BLUE.

`get_enemy_team() -> Team`

Returns your opponent's team, returning either Team.RED or Team.BLUE.

`get_map() -> Map`

Returns the current map instance.

`get_orders() -> List[Dict]`

Returns the list of orders, with each order represented by a dictionary.
Dictionary Items:

- **order_id:** id number for the order,
- **required:** the names of the foods required,
- **created_turn:** the turn the order appears,
- **expires_turn:** the turn the order will expire,
- **reward:** the reward for successfully submitting the order,
- **penalty:** penalty for letting the order expire,
- **claimed_by:** the id of the bot the order was claimed by,
- **completed_turn:** the turn that the order was submitted, or None,
- **is_active:** whether or not the order is currently active,

`get_team_bot_ids() -> List[int]`

Returns the bot IDs of your team as a list.

`get_team_money() -> int`

Returns the amount of money your team currently has.

`get_bot_state(bot_id:int) -> Optional[Dict]`

Returns the state of the given bot as a dictionary.
The dictionary contains:

- **bot_id:** the bot id
- **team:** the team name
- **x:** the bot's x-coordinate
- **y:** the bot's y-coordinate
- **team_money:** the amount of money the team has
- **holding:** the item the bot is currently holding
- **map_team:** the team associated to the map the bot is currently on

`get_tile(team:Team, x:int, y:int) -> Optional[Tile]`

Returns the tile at the given (x, y) coordinate.

Movement Functions

`can_move(bot_id:int, dx:int, dy:int) -> bool`

Returns whether the given bot can move to coordinates (dx, dy).

`move(bot_id: int, dx: int, dy: int) -> bool`

Moves the bot to coordinates (dx, dy); returns True if successful and False if not.

Inventory Functions

`pickup(bot_id: int, target_x: Optional[int] = None, target_y: Optional[int] = None) -> bool`

Bot picks up an item from target location (target_x, target_y); returns True if successful and False if not. Can be used to pick up items from boxes.

`place(bot_id: int, target_x: Optional[int] = None, target_y: Optional[int] = None) -> bool`

Bot places an item at target location (target_x, target_y); returns True if successful and False if not. Can be used to place items on pans and boxes.
Placement rules:

- can place anything in an empty box
- can only place food on a cooker with a pan on it

`trash(bot_id: int, target_x: Optional[int] = None, target_y: Optional[int] = None) -> bool`

If coordinates (target_x, target_y) are the coordinates of the trash tile, the bot disposes of:

- the food item it is currently holding, or
- the food items in the pan or plate it is currently holding

And returns True. Otherwise, trash disposal fails and False is returned.

Shop Functions

`can_buy(bot_id: int, item: Buyable, target_x: Optional[int] = None, target_y: Optional[int] = None) -> bool`

Checks if the bot can buy an item from the shop.
Requirements:

- target tile (target_x, target_y) is a shop tile
- bot is not carrying anything.

Returns True if possible, False otherwise.

`buy(bot_id: int, item: Buyable, target_x: Optional[int] = None, target_y: Optional[int] = None) -> bool`

Buys an item from the shop.
Requirements:

- target tile (target_x, target_y) is a shop tile
- bot is not carrying anything.

Returns True if successful False otherwise.

Food Processing Functions

`chop(bot_id: int, target_x: Optional[int] = None, target_y: Optional[int] = None) -> bool`

Chops food on a counter; returns True if successful, False if not.

`can_start_cook(bot_id: int, target_x: Optional[int] = None, target_y: Optional[int] = None) -> bool`

Check if the bot can start cooking.
Requirements:

- target tile (target_x, target_y) is a cooker with an empty pan placed on it, and
- bot is carrying a cookable food item.

Returns True if possible, False otherwise.

`start_cook(bot_id: int, target_x: Optional[int] = None, target_y: Optional[int] = None) -> bool`

Begins cooking at the cooker at tile (target_x, target_y).
Requirements:

- target tile (target_x, target_y) is a cooker with an empty pan placed on it, and
- bot is carrying a cookable food item.

If the food item has been previously cooked, the food will continue cooking at the beginning of the stage it was previously removed from the cooker at.
Returns True if cooking begins successfully, False otherwise.

`take_from_pan(self, bot_id: int, target_x: Optional[int] = None, target_y: Optional[int] = None) -> bool:`

Removes food from the pan at tile (target_x, target_y).
Requirements:

- target tile (target_x, target_y) is a cooker with a non-empty pan placed on it, and
- bot is not carrying anything

Returns True if successful, False otherwise.

Plate Functions

`take_clean_plate(self, bot_id: int, target_x: Optional[int] = None, target_y: Optional[int] = None) -> bool`

Take a clean plate from the sink table at coordinates (target_x, target_y).
Requirements:

- sink table has available clean plates,
- target tile (target_x, target_y) is a sink table tile, and
- bot is not holding anything

Returns True if successful, False otherwise.

`put_dirty_plate_in_sink(self, bot_id: int, target_x: Optional[int] = None, target_y: Optional[int] = None) -> bool`

Place a dirty plate in the sink at coordinates (target_x, target_y).
Requirements:

- target tile (target_x, target_y) is a sink tile
- bot is holding a dirty plate

Returns True if successful, False otherwise.

`wash_sink(self, bot_id: int, target_x: Optional[int] = None, target_y: Optional[int] = None) -> bool`

Wash a dirty plate in the sink at coordinates (target_x, target_y).
Requirements:

- target tile (target_x, target_y) is a sink tile
- sink contains a dirty plate

Returns True if successful, False otherwise.

`add_food_to_plate(self, bot_id: int, target_x: Optional[int] = None, target_y: Optional[int] = None) -> bool`

Either:

- add the food item at coordinates (target_x, target_y) to the plate the bot is holding
- add the food item the bot is holding to the plate at coordinates (target_x, target_y)

Requirements:

- target tile (target_x, target_y) contains food or a clean plate
- bot is holding food or a clean plate

Returns True if successful, False otherwise.

Submit Functions

`can_submit(bot_id: int, target_x: Optional[int] = None, target_y: Optional[int] = None) -> bool`

Checks if the currently held plate can be submitted.

Requirements:

- target tile (target_x, target_y) is the submit station
- bot is holding a clean plate

Returns True if possible, False otherwise.

`submit(bot_id: int, target_x: Optional[int] = None, target_y: Optional[int] = None) -> bool`

Submits the currently held plate.

Requirements:

- target tile (target_x, target_y) is the submit station
- bot is holding a clean plate

Returns True if possible, False otherwise. Warns if no order matches the submission.

Switch Functions

`get_switch_info() -> Dict[str, Any]`

Provides information about the switch, including turn, switch duration, and the switch status of both teams.

`can_switch_maps() -> bool`

Returns True if the user can switch into the enemy map, False otherwise. Can switch any time between turn 250 and 350, but can only switch once.

`switch_maps() -> bool`

Immediately teleports all bots on the user's team into the enemy map with non-interferring spawns. Does not consume a bot's move for that turn. Can only be called once per game per team.
Returns True if successful, false otherwise.

Item Information

`item_to_public_dict(it: Optional[Item]) -> Any`

Provides information about an item such as type, name, and ID.
Dictionary Items:

- **type:** "Food", "Plate", or "Pan"
    - Food Additional Items
        - **food_name:** name of the food
        - **food_id:** id number for the food type
        - **chopped:** whether or not the food is chopped
        - **cooked_stage:** 0 (uncooked), 1(cooked), or 2(burnt)
    - Plate Additional Items
        - **dirty:** whether or not the plate is dirty
        - **food:** list of dicts with above items
    - Pan Additional Items
        - **food:** a dictionary with information about the food item in the pan

Game Constants

`Map`

A class that details the environment

Convention: bottom-left is [0][0], top-right is [width-1][height-1]

The map is structured as follows:

```
                 y == height -----
x == width   [[# # # # # # # # #],
      |       [# # # # # # # # #],
      |       [# # # # # # # # #],
      |      [# # # # # # # # #],
      v      [# # # # # # # # #]]
```

`Tile`

A class containing the following fields:

- **tile_name:** name of the tile
- **tile_id:** id of the tile
- **is_walkable:** whether the tile is walkable
- **is_dangerous:** whether the tile is dangerous
- **is_placeable:** whether items can be placed on the tile
- **is_interactable:** whether the tile can be interacted with

`Tile Types`

There are 10 types of tiles:

- **Floor:** walkable floor tiles
- **Wall:** unwalkable wall tiles
- **Counter:** unwalkable tiles that items and ingredients can be placed on
- **Box:** unwalkable tiles that multiple ingredients of the same type can be placed in and removed from
- **Sink:** where dirty dishes are placed and washed
- **SinkTable:** where clean dishes appear after being washed
- **Cooker:** where ingredients are cooked
- **Trash:** where food is thrown away
- **Submit:** where orders are submitted
- **Shop:** where ingredients and items can be purchased

`Ingredients`

There are 5 different ingredients that can be processed and purchased.

Ingredient	**ID**	**Choppable**	**Cookable**	**Cost**
Egg	0	False	True	20
Onion	1	True	False	30

Meat	2	True	True	80
Noodles	3	False	False	40
Sauce	4	False	False	10

Items
There are 2 different items that can be purchased from the shop.

Item	**Cost**
Plate	2
Pan	4