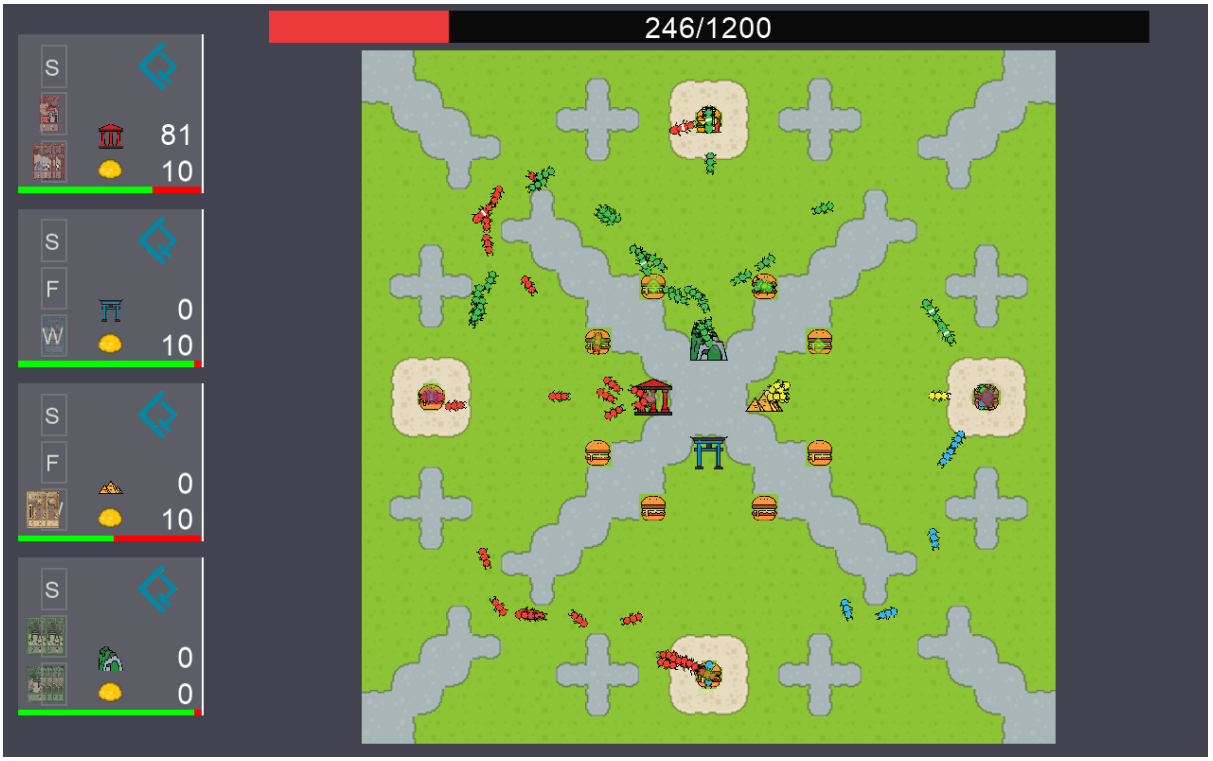
Code Quest 22 Documentation

Welcome to the Code Quest!
This document should contain all relevant information for playing and developing for the Code Quest 22 game. The document is rather large, simply because we want to include all information that could be necessary for you to know. It is split up into 4 parts:

- What is the Game: A quick intro to the basic premise of the game, the map tiles, and various ants.
- Game Setup: Getting the package installed, and then going over the basics of the sample bot.
- Game Interface: A directory of all modules you might want to use in the package, and description of features you must implement.
- Stats: A full list of all stats that the package exposes for you to use.

Obviously reading the entire thing is optimal, but you could probably get started after just the first 2 sections, and skim through the final 2 when needed.

What is the Game



“King of the Ant Hill” is a game played on a grid world. The game features the titular characters, ants, battling it out for resources, territory, and glory.

There are 2 commodities in this game; Energy and Hill points. Energy is collected from worker ants at food sites, and Hill points are collected from settler ants when sitting on “The Hill”.

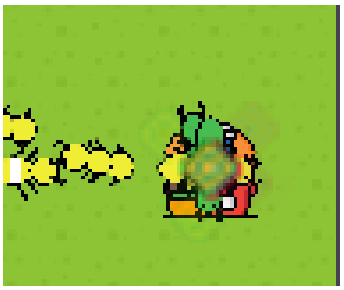


Figure 1: Workers collecting energy

Figure 2: Settlers going to the active hill

Hill points

Hill points are the deciding factor in a game - The player with a higher hill score wins.
There are only two ways to get hill score:

- Have settler ants idle on an active hill, OR
- Defeat an enemy queen ant, to collect all of that player's hill score.

Energy

Energy is collected by worker ants, and is used to spawn more ants, allowing you to collect hill points, capture territory, and so on.

The Map

The map is primarily made up of grass and walls, traversable and non-traversable terrain, with a select few special tiles.

The Hill

A map can have multiple "Hill" areas, but only at most one of them will be active at a time.
A hill is active when it has a purple outline, as visible in Figure 2.

In order to ensure fairness, each hill area will be active for approximately the same amount of time, and no hill area will be active for the first 100 ticks.

Food Tiles

Food tiles are scattered throughout the map, and without them the game would be over very quickly. They are the only source of energy in the game, allowing teams to build large armies of ants.
Whenever worker ants are on a food tile, they receive energy, which they then need to send back to the queen ant to deposit. Only then can the energy be used.

In particular, every 2 ticks, the energy tile will look for any unencumbered worker ants in the vicinity. It will pick 1 at random, and bestow upon that ant a certain amount of energy, making that ant encumbered, until they return to the queen.

The certain amount of energy the food tile gives is dependent on:

- The base stat of the particular tile
- If the food tile is overcharged

Much like hill zones, food tiles overcharge in a fair manner through the game, doubling the yield of food tiles.

Spawns

Each map will feature 4 spawn locations, which are also the positions of the queen ants. Any other ants spawned will start off here, and you can defeat other teams by sending fighter ants to the enemy queen.
As mentioned above, workers have to return here to deposit energy.

The Ants

Ants are the players of the game, each has HP and a lifespan. An ant dies when it runs out of HP, or when its lifespan is up.

Workers

A worker's one purpose is to collect energy for the queen. They don't primarily win you the game but are definitely vital in a sustainable ant colony.
A worker's lifespan is not defined in game ticks, but instead in the number of times they deposit energy. Workers are fast ants, but slow down when encumbered with energy.

Settlers

Settlers are the only way to sustainably and reliably collect Hill points for your team. They cannot fight and only serve the purpose of idling at the Hill zones. Settlers are mid range speed, allowing them to outrun fighters.

Fighters

Fighters spice up the game by allowing you to force enemy ants off of positions. Fighters have the most HP, and can attack up to 2 ants per tick. With a relatively short lifespan, they need to have a specific purpose in mind to be effective.

Game Setup

Installing

Before even installing Code Quest, you should:

1. Install Python if you haven't already (making sure to tick "Add Python to PATH" in the custom installation)
2. (Optional) Setup a virtualenv for this specific folder.

The first step is already covered in many tutorials, so we've omitted any instructions here. You can test if you've done this correctly by opening a terminal and typing `python` then pressing enter. If the windows store opens, or `python` is not recognised as a command, then you haven't done it correctly.

For the second step, creating a virtualenv is an easy way to separate the package installs for each project you work on. Setting up a virtualenv is quite simple:

```
pip install virtualenv
virtualenv -p 3.8 venv
./venv/Scripts/activate.bat    <- Windows
source venv/Scripts/activate   <- Linux
```

Replacing 3.8 with whatever Python version you want to work with. CodeQuest22 should be compatible with anything from 3.7+, and will be run on the server with Python 3.10.

Those last two lines will need to be entered each time you wish to use this virtual environment.

Now that we have our virtual environment, we can install the packages required for this game. To do this type:

```
python -m pip install -U codequest22
```

This should take a minute, after which you can test the install went fine by downloading the sample bot, unzipping it and running the game:

```
codequest22 sample_bot sample_bot
```

Provided that is all good, let's try writing a bot!

Writing a bot

This section essentially goes in detail on the code written in the Day 1 workshop. If you'd rather read the code yourself and infer this, you can do so by downloading the sample here. The structure of a bot for this game is rather simple. It need only be a python file that implements a few basic functions:

- `get_team_name`: Returns a string, your team name.
- `read_map`: Reads the map data for each game, and allows for precomputation.
- `read_index`: Reads your player index (Whether you are player 0, 1, 2 or 3).

And to have your bot actually react to the game, you need to also implement:

- `handle_events`: The mainloop of your program.
- `handle_failed_requests`: Handle any illegal requests you've made.
- If you want a profile picture, then add your `main.py` file to a folder, and add a file called `profile.png`

But enough of that, let's build up a program that will send a worker to the closest food tile. We'll start with the basic functions:

```
def get_team_name():
    return "My Cool Team"

num_players = None
my_index = None
def read_index(player_index, n_players):
    global my_index, num_players
    my_index = player_index
    num_players = n_players

def read_map(md, energy_info):
    pass

def handle_failed_requests(requests):
    pass

def handle_events(events):
    pass
```

Now, any precomputation on the map should be done in read_map. This function is given 1 second to run before the server will cut you off.

The `md` argument in read_map is short for map data. This is a 2D list representing the game map. Each entry in the list is either:

- ".": An empty square
- "W": A wall
- "F": A food tile

- “Z”: A hill tile
- “R/B/Y/G”: A player spawn location

So let’s start by writing some quick Dijkstra’s to find the closest production site to our spawn. I’m going to comment it out here, so the document isn’t too long, but you can read the source code in the download above.

```
map_data = {}
# A list of spawn locations per player.
spawns = [None]*4
# A list of all food tiles
food = []
# Storing the distance from your spawn
distance = {}
closest_site = None
def read_map(md, energy_info):
    # Dijkstra stuff happens here.
    food_sites = list(sorted(food, key=lambda prod:
distance[prod]))
    closest_site = food_sites[0]
```

Now we can work on the guts of our bot - how we get the bot to react to certain events in the game!

The brains of your bot lives in the `handle_events` function. This function is run every game tick, and should:

1. Parse the event list argument.
2. Return a list of all requests to be made.

`events` is a list of `codequest22.server.events.Event` objects. A full list of these events are included in that file, but for this bot we will only concern ourselves with Deposit, Production, and Die events.

The function should return a list of `codequest22.server.requests.Request` objects, although there are only two: `SpawnRequests` and `GoalRequests`.

So, let’s start by just spawning some worker ants whenever we have enough energy, and sending them to the closest food tile. We can achieve this like so:

```
import codequest22.stats as stats
from codequest22.server.ant import AntTypes
from codequest22.server.requests import
SpawnRequest

my_energy = stats.general.STARTING_ENERGY
def handle_events(events):
```

```
    requests = []
    if my_energy >= stats.ants.Worker.BASE_COST:
        requests.append(SpawnRequest(
            AntTypes.WORKER,
            id=None,
            color=None,
            goal=closest_site,
            cost=None,
        ))
    return requests
```

Now, if we run the game, you should see that we spawn a few worker ants that make their way to the food tile, but never return! To fix this, we need to change the worker's goal once we receive a ProductionEvent, which is emitted when an ant becomes encumbered:

```
import codequest22.stats as stats

my_energy = stats.general.STARTING_ENERGY
def handle_events(events):
    requests = []

    for event in events:
        if isinstance(event, ProductionEvent):
            if event.player_index == my_index:
                # One of my worker ants just made
it to the food site!
                # Let's send them back to the
Queen.
                requests.append(GoalRequest(
                    event.ant_id,
                    spawns[my_index]
                ))
            elif isinstance(event, DepositEvent):
            if event.player_index == my_index:
                # One of my worker ants just made
it back to the Queen!
                # Let's send them back to the food
site.
                requests.append(GoalRequest(
                    ev.ant_id,
                    closest_site
```

```
                ))
                # Additionally, let's update how
much energy I've got.
                my_energy = event.cur_energy

    if my_energy >= stats.ants.Worker.BASE_COST:
        requests.append(SpawnRequest(
            AntTypes.WORKER,
            id=None,
            color=None,
            goal=closest_site,
            cost=None,
        ))
    return requests
```

Nice, now we've got some ants that move back and forth between the food tiles, collecting energy for the queen, which then spends that energy on more workers!

A correct implementation should also ensure we don't spawn too many ants, as you can only have 100 ants alive at any given time, and so a full, safe implementation of this code is given in the sample bot, linked above.

Game Interface

There a few modules you should be aware of when implementing your bots:

Events

`codequest22.server.events` includes all events that can occur throughout the game. The full list is:

- `SpawnEvent`
- `MoveEvent`
- `DieEvent`
- `AttackEvent`
- `DepositEvent`
- `ProductionEvent`
- `ZoneActiveEvent`
- `ZoneDeactivateEvent`
- `FoodTileActiveEvent`
- `FoodTileDeactivateEvent`
- `SettlerScoreEvent`
- `QueenAttackEvent`
- `TeamDefeatedEvent`

For the full description of each event, alongside what data is exposed, see the docstrings for each of these events.

Requests

`codequest22.server.requests` includes all requests you can make throughout the game.
There are only two requests, namely:

`SpawnRequest(ant_type, id=None, color=None, goal=None)`

`ant_type` must be one of AntTypes.WORKER, AntTypes.FIGHTER or AntTypes.SETTLER.
`id` allows you to set the specific ant id. If no id is provided, one will be automatically assigned.
`color` allows you to set a little colour spot on your ant for debugging purposes.
`goal` specifies the current goal of the ant. If none is specified, the ant will stay at the queen.

`GoalRequest(id, position)`

`id` specifies the ant whose goal you want to change.
`position` specifies the location of this new goal.

Functions

This has already been slightly covered in the “Writing a bot” section, but to reiterate your bot should implement the following functions:

- `get_team_name`: Returns a string, your team name.
- `read_map`: Reads the map data for each game, and allows for precomputation.
- `read_index`: Reads your player index (Whether you are player 0, 1, 2 or 3), as well as how many players there are.

And to have your bot actually react to the game, you need to also implement:

- `handle_events`: The mainloop of your program.
- `handle_failed_requests`: Handle any illegal requests you’ve made.

`read_map(md, energy_info)`

`md` is the map data, a 2D list. `md[0][0]` is the top left square, and `md[0][-1]` is the top right square.
`energy_info` tells you how much energy each food tile gives an ant. It is a dictionary which maps string positions to values. So for the food tile at x=5, y=1 with 30 energy, would have entry: `energy_info["(5, 1)"] == 30`

Using the visual interface

You can interact with the simulation in various ways, by pressing certain buttons:

- P pauses and unpauses the simulation

- O speeds up the simulation, Shift+O greatly speeds up the simulation
- I slows down the simulation, Shift+I greatly slows down the simulation
- Left clicking an ant will highlight the ant's goal and current progress towards that goal
- Right clicking a tile will give you any information about that tile

Using the command line codequest22

Passing `--help` to the command line gives documentation for all extra arguments you can pass to `codequest22`.

Stats

There are a lot of rules in this competition, whether that be governing ant health, food tile replenish rules, or settler timeout rules.

All of these are completely editable, and you should use these dynamic values in your code, rather than static ones. This allows for:

- You to play around with certain situations
- Us to balance the game
- You to write more readable code

The stats are all available in `codequest22.stats`, in submodules `ants`, `general`, `hill` and `energy`.
Below we will list all stats, and their descriptions:

General

Any generic statistics about gameplay live here.

`MAX_ANTS_PER_PLAYER = 100`

The maximum number of ants a single player can have alive at any given time.

`SIMULATION_TICKS = 1200`

The total number of ticks the game will simulate for.

`MAX_SPAWNS_PER_TICK = 5`

The maximum number of ants a single player can spawn in a single tick.

`MAX_ENERGY_STORED = 750`

The maximum amount of energy that can be stored by a single player.

`QUEEN_HEALTH = 3000`

The health of each player's queen ant.

`STARTING_ENERGY = 100`

The starting energy of each player.

Hill

Any statistics about hill activation goes here.

`GRACE_PERIOD = 100`

No hill tile will activate for GRACE_PERIOD ticks.

`NUM_ACTIVATIONS = 2`

Each hill tile will activate NUM_ACTIVATIONS times.

`MIN_ZONE_TIME = 60`

Each time a hill activates, it will do so for at least MIN_ZONE_TIME ticks

`MAX_ZONE_TIME = 80`

Each time a hill activates, it will do so for at most MAX_ZONE_TIME ticks

`MIN_WAIT_TIME = 20`

The time between hill activations will be at least MIN_WAIT_TIME ticks.

Energy

Any statistics about energy/food tiles go here.

`DELAY = 2`

`PER_TICK = 1`

The food tile will encumber PER_TICK ants every DELAY ticks

`GRACE_PERIOD = 100`

Food tiles will not overcharge for the first GRACE_PERIOD ticks.

`MIN_OVERCHARGE_TIME = 50`

When a food tile overcharges, it will do so for at least MIN_OVERCHARGE_TIME ticks.

`MAX_OVERCHARGED_TIME = 80`

When a food tile overcharges, it will do so for at most MAX_OVERCHARGE_TIME ticks.

`MIN_WAIT_TIME = 50`

A food tile must wait MIN_WAIT_TIME ticks before it can overcharge again.

`NUM_ACTIVATIONS = 3`

Every food tile will overcharge exactly NUM_ACTIVATIONS times.

Ants

General statistics about ants live here.

Worker

`COST = 20`

`HP = 10`

`SPEED = 3`

How much distance the worker can travel in a single tick.

`ENCUMBERED_RATE = 0.3`

This multiplies the worker's speed whenever encumbered.

Fighter

`COST = 40`

`HP = 15`

`SPEED = 2`

How much distance the fighter can travel in a single tick

`ATTACK = 3.5`

How much HP damage the fighter does in a single attack

`RANGE = 1.5`

The distance over which a fighter can attack other ants

`NUM_ATTACKS = 2`

The maximum number of ants the fighter can attack

`LIFESPAN = 30`

The total number of turns the fighter stays alive for.

Settler

`COST = 30`

`HP = 10`

`SPEED = 1.5`

How much distance the settler can travel in a single tick

`LIFESPAN = 40`

The total number of turns the settler stays alive for.