Introduction

> The bees are coming!
>
> Create a better soldier with inherit-ants.

In this project, you will create a tower defense game called Ants Vs. SomeBees. As the ant queen, you populate your colony with the bravest ants you can muster. Your ants must protect their queen from the evil bees that invade your territory. Irritate the bees enough by throwing leaves at them, and they will be vanquished. Fail to pester the airborne intruders adequately, and your queen will succumb to the bees' wrath. This game is inspired by PopCap Games' Plants Vs. Zombies.

This project combines functional and object-oriented programming paradigms, focusing on the material from Chapter 2.5 of Composing Programs. The project also involves understanding, extending, and testing a large program.

> **Important submission notes**: This project has four basic phases and two bonus phases. You have 3 weeks in total. We recommend you finish each phase with no more than 3 days. Check the exact deadline on our OJ website.

After completing any problems required in each phase, you need to submit your answer to our OJ website to get your answer scored. We recommend that you submit **after you finish each problem** so that you can find bugs as soon as possible.

Project Structure

To get started, download project materials `proj03-Code.zip` from our QQ group if you don't have one. Below is a list of all the files you will see in the `proj03-Code.zip`. You only have to make changes to `ants/ants.py` to finish the project.

```
ants
  |-ants.py     # The game logic of Ants Vs. SomeBees
  |-gui.py      # Web-based GUI for Ants Vs. SomeBees
  |-ants_gui.py # The original GUI for Ants Vs. SomeBees
  |-graphics.py # Utilities for displaying simple two-dimensional animations
  |-utils.py    # Some functions to facilitate the game interface
  |-ucb.py      # Utility functions from CS 61A
  |-assets      # A directory of images and files used by gui.py
  |-img         # A directory of images used by ants_gui.py
  |-ok          # The autograder
  |-proj03.ok   # The ok configuration file
  -tests        # A directory of tests used by ok
```

As you may see, we have two version of GUIs in the project. One is in `gui.py`, the other is in `ants_gui.py`. Feel free to choose any one you like.

> *Note*: In this time we enable the `unlock` feature in `ok` to strength your understanding of the problems.

Suggestion:

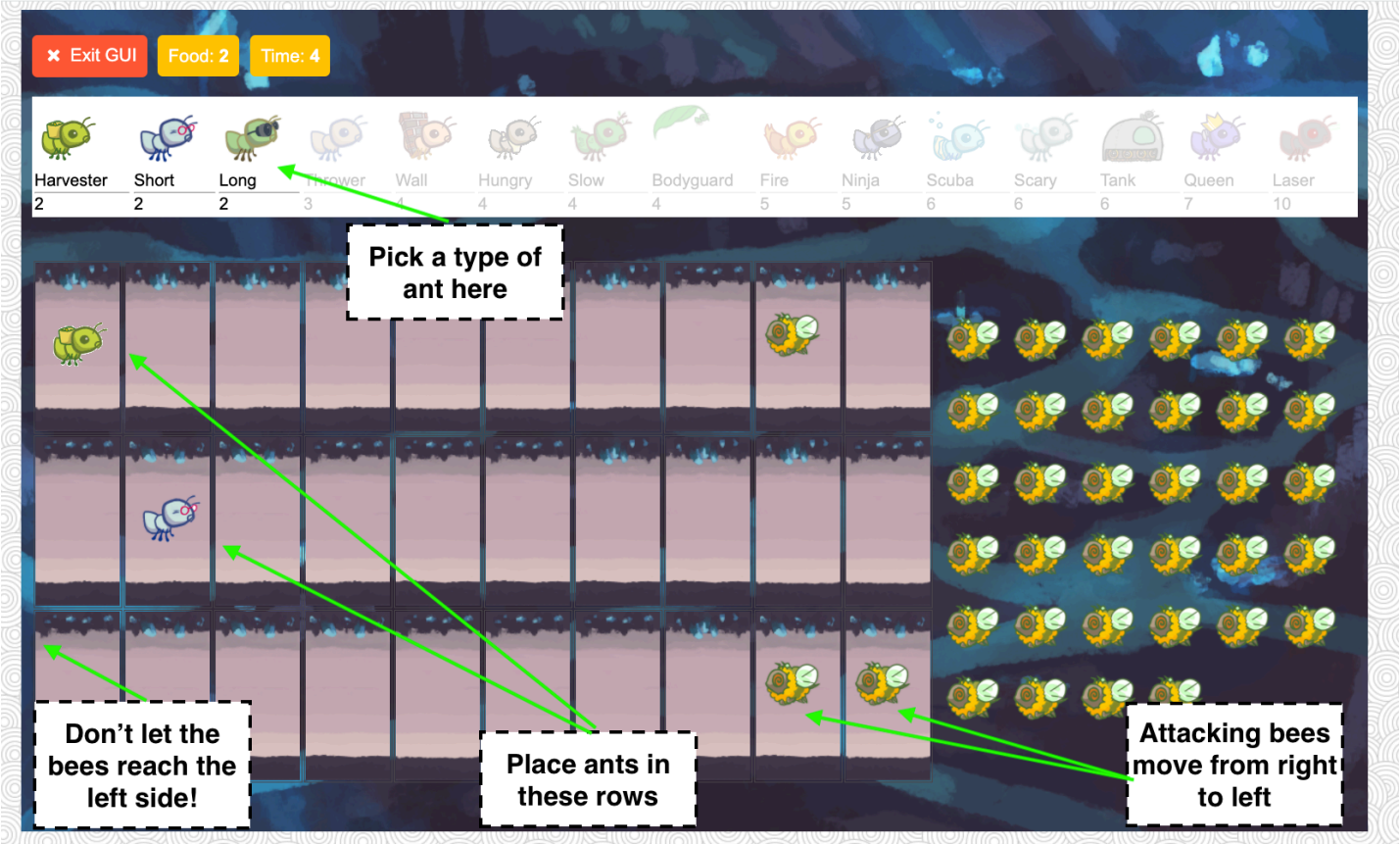
For the functions that we ask you to complete, there may be some initial code that we provide. If you would rather not use that code, feel free to delete it and start from scratch. You may also add new function definitions as you see fit.

However, please do **not** modify any other functions. Doing so may result in your code failing our framework. Also, please do **not** change any function signatures (names, argument order, or number of arguments).

And, check the given tests when you are confused about how some functions should behave.

The Game

A game of Ants Vs. SomeBees consists of a series of turns. In each turn, new bees may enter the ant colony. Then, new ants are placed to defend their colony. Finally, all insects (ants, then bees) take individual actions. Bees either try to move toward the end of the tunnel or sting ants in their way. Ants perform a different action depending on their type, such as collecting more food or throwing leaves at the bees. The game ends either when a bee reaches the end of the tunnel (you lose), the bees destroy the `QueenAnt` if it exists (you lose), or the entire bee fleet has been vanquished (you win).



Core Concepts

The Colony. This is where the game takes place. The colony consists of several Places that are chained together to form a tunnel where bees can travel through. The colony also has some quantity of food which can be expended in order to place an ant in a tunnel.

Places. A place links to another place to form a tunnel. The player can put a single ant into each place. However, there can be many bees in a single place.

The Hive. This is the place where bees originate. Bees exit the beehive to enter the ant colony.

Ants. Players place an ant into the colony by selecting from the available ant types at the top of the screen. Each type of ant takes a different action and requires a different amount of colony food to place. The two most basic ant types are the HarvesterAnt, which adds one food to the colony during each turn, and the ThrowerAnt, which throws a leaf at a bee each turn. You will be implementing many more!

Bees. In this game, bees are the antagonistic forces that the player must defend the ant colony from. Each turn, a bee either advances to the next place in the tunnel if no ant is in its way, or it stings the ant in its way. Bees win when at least one bee reaches the end of a tunnel.

Core Classes

The concepts described above each have a corresponding class that encapsulates the logic for that concept. Here is a summary of the main classes involved in this game:

GameState: Represents the colony and some state information about the game, including how much food is available, how much time has elapsed, where the AntHomeBase is, and all the Places in the game.

Place: Represents a single place that holds insects. At most one Ant can be in a single place, but there can be many Bees in a single place. Place objects have an exit to the left and an entrance to the right, which are also places. Bees travel through a tunnel by moving to a Place's exit.

Hive: Represents the place where Bees start out (on the right of the tunnel).

AntHomeBase: Represents the place Ants are defending (on the left of the tunnel). If Bees get here, they win :(

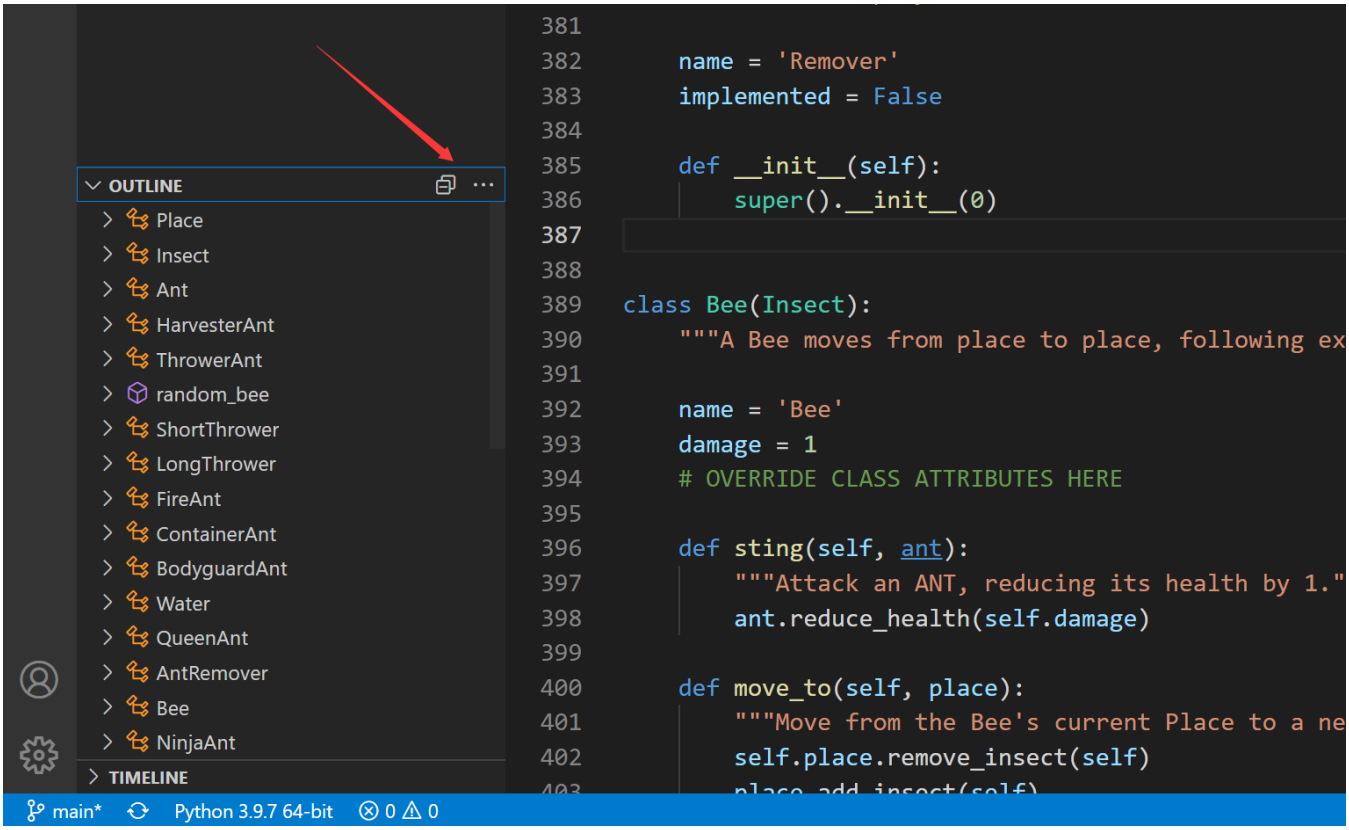
Insect: A superclass for Ant and Bee. All insects have health attribute, representing their remaining health, and a place attribute, representing the Place where they are currently located. Each turn, every active Insect in the game performs its action.

Ant: Represents ants. Each Ant subclass has special attributes or a special action that distinguish it from other Ant types. For example, a `HarvesterAnt` gets food for the colony and a `ThrowerAnt` attacks Bees. Each ant type also has a `food_cost` attribute that indicates how much it costs to deploy one unit of that type of ant.

Bee: Represents bees. Each turn, a bee either moves to the exit of its current Place if the Place is not blocked by an ant, or stings the ant occupying its same Place.

To help visualize how all the classes fit together, `cs61a` also created an object map for you to reference as you work, which you can find here. Pay attention that we do not have same scores with `cs61a` :D

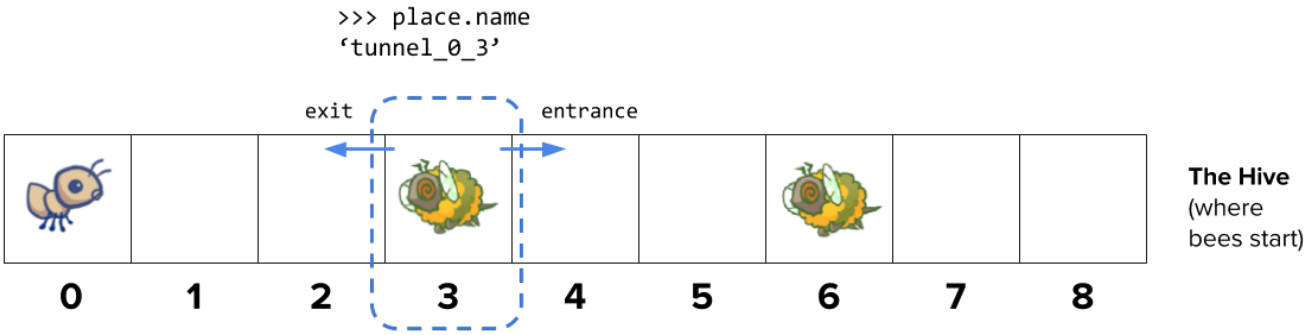
> *Tips about VSCode*: For VSCode user, we can use **OUTLINE** in the explorer to view the source code structure conveniently.



Try many useful actions pointed to by the red arrow if we would like to maximize our coding power.

Game Layout

Below is a visualization of a GameState. ***As you work through the unlocking tests and problems, we recommend drawing out similar diagrams to help your understanding.***



Example: AntColony with dimensions (1, 9)

Playing the Game

The game can be run in two modes: as a text-based game or using a graphical user interface (GUI). The game logic is the same in either case, but the GUI enforces a turn time limit that makes playing the game more exciting. The text-based interface is provided for debugging and development.

The files are separated according to these two modes. `ants.py` knows nothing of graphics or turn time limits.

To start a text-based game, run

```
python ants_text.py
```

To start a graphical game, run

```
python gui.py
```

or

```
python ants_gui.py
```

When you start the graphical version, a new window should appear. In the starter implementation, you have unlimited food and your ants can only throw leaves at bees in their current Place.

Before you complete Problem 2, the GUI may crash since it doesn't have a full conception of what a Place is yet! Try playing the game anyway! You'll need to place a lot of `ThrowerAnts` (the second type) in order to keep the bees from reaching your queen.

The game has several options that you will use throughout the project, which you can view with `python3 ants_text.py --help`.

```
usage: ants_text.py [-h] [-d DIFFICULTY] [-w] [--food FOOD]

Play Ants vs. SomeBees

optional arguments:
  -h, --help     show this help message and exit
  -d DIFFICULTY  sets difficulty of game (test/easy/normal/hard/extra-hard)
  -w, --water    loads a full layout with water
  --food FOOD    number of food to start with when testing
```

Phase 1: Basic Gameplay

In the first phase you will complete the implementation that will allow for basic gameplay with the two basic Ants: the `HarvesterAnt` and the `ThrowerAnt`.

Problem 0 (0pts): About the Game

Answer the following questions after you have read the entire `ants.py` file, to guarantee you have the right understanding about the game.

```
python ok -q 00 -u
```

If you get stuck while answering these questions, you can try reading through `ants.py` again, consult the core concepts or core classes sections above.

1. What is the significance of an Insect's `health` attribute? Does this value change? If so, how?
2. Which of the following is a class attribute of the `Insect` class?
3. Is the `health` attribute of the `Ant` class an instance attribute or a class attribute? Why?
4. Is the `damage` attribute of an `Ant` subclass (such as `ThrowerAnt`) an instance attribute or class attribute? Why?
5. Which class do both `Ant` and `Bee` inherit from?
6. What do instances of Ant and instances of Bee have in common?
7. How many insects can be in a single `Place` at any given time (before Problem 8)?
8. What does a `Bee` do during one of its turns?
9. When is the game lost?

Remember to run:

```
python ok -q 00 -u
```

Problem 1 (150pts): Ants Need Food

Part A: Currently, there is no cost for placing any type of `Ant`, and so there is no challenge to the game. The base class `Ant` has a `food_cost` of zero. Override this class attribute for `HarvesterAnt` and `ThrowerAnt` according to the "Food Cost" column in the table below.

	Class	Food Cost	Health
	`HarvesterAnt`	2	1
	`ThrowerAnt`	3	1

Part B: Now that deploying cost food of `Ant`, we need to be able to gather more food! To fix this issue, implement the `HarvesterAnt` class. A `HarvesterAnt` is a subclass of `Ant` that adds one food to the `gamestate.food` total in its `action`.

Before

Before writing any code, read the instructions and test your understanding of the problem:

```
python ok -q 01 -u
```

After

After writing code, test your implementation:

```
python ok -q 01
```

Try playing the game by running `python3 gui.py`. Once you have placed a `HarvesterAnt`, you should accumulate food each turn. You can also place `ThrowerAnt`s, but you'll see that they can only attack bees that are in their `Place`, making it a little

difficult to win.

Problem 2 (200pts): Build Tunnel

Complete the `Place` constructor by adding code that tracks entrances. Right now, a `Place` keeps track only of its `exit`. We would like a `Place` to keep track of its entrance as well. A `Place` needs to track only one `entrance`. Tracking entrances will be useful when an `Ant` needs to see what `Bee`s are in front of it in the tunnel.

However, simply passing an entrance to a `Place` constructor will be problematic; we would need to have both the exit and the entrance before creating a `Place`! (It's a chicken or the egg problem.) To get around this problem, we will keep track of entrances in the following way instead. The `Place` constructor should specify that:

- A newly created `Place` always starts with its `entrance` as `None`.
- If the `Place` has an `exit`, then the `exit`'s `entrance` is set to that `Place`.

Hint1: Remember that when the `__init__` method is called, the first parameter, `self`, is bound to the newly created object.

Hint2: Try drawing out two `Place`s next to each other if things get confusing. In the GUI, a place's `entrance` is to its right while the `exit` is to its left.

Hint3: Remember that Places are not stored in a list, so you can't index into anything to access them. This means that you **can't** do something like `colony[index + 1]` to access an adjacent Place. How can you move from one place to another?

Before

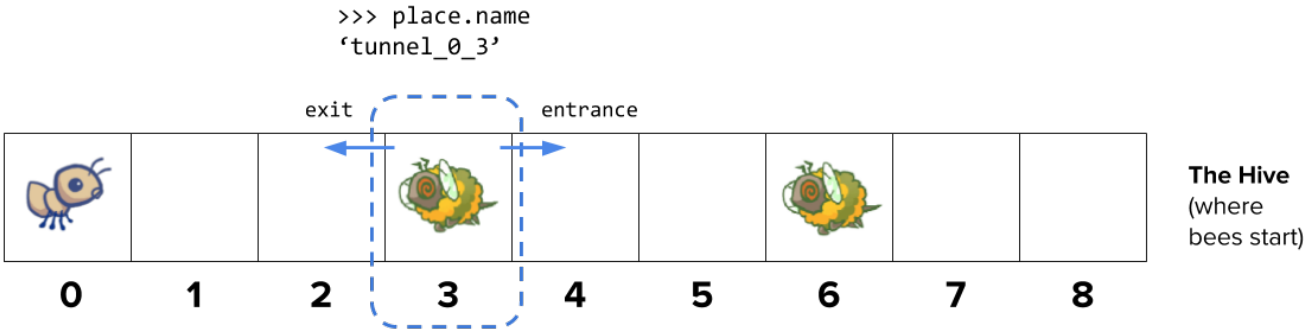
Before writing any code, read the instructions and test your understanding of the problem:

```
python ok -q 02 -u
```

After

After writing code, test your implementation:

```
python ok -q 02
```



Example: AntColony with dimensions (1, 9)

Problem 3 (200pts): Thrower Throws

In order for a `ThrowerAnt` to attack, it must know which bee it should hit. The provided implementation of the `nearest_bee` method in the `ThrowerAnt` class only allows them to hit bees in the same `Place`. Your job is to fix it so that a `ThrowerAnt` will `throw_at` the nearest bee in front of it that is not still in the `Hive`. Go back to Core Concepts to review the definition of `Hive`.

> *Hint*: All Places have an `is_hive` attribute which is `True` when that place is the `Hive`.

The `nearest_bee` method returns a random `Bee` from the nearest place that contains bees. Places are inspected in order by following their `entrance` attributes.

- Start from the current `Place` of the `ThrowerAnt`.
- For each place, return a random bee if there is any, or consider the next place that is stored as the current place's `entrance`.
- If there is no bee to attack, return `None`.

> *Hint1*: The `random_bee` function provided in `ants.py` returns a random bee from a list of bees or None if the list is empty.
>
> *Hint2*: As a reminder, if there are no bees present at a Place, then the bees attribute of that Place instance will be an empty list.
>
> *Hint3*: Having trouble visualizing the test cases? Try drawing them out on paper! The example diagram provided in Game Layout shows the first test case for this problem.

Before

Before writing any code, read the instructions and test your understanding of the problem:

```
python ok -q 03 -u
```

After

After writing code, test your implementation:

```
python ok -q 03
```

After implementing `nearest_bee`, a `ThrowerAnt` should be able to `throw_at` a `Bee` in front of it that is not still in the `Hive`. Make sure that your ants do the right thing! To start a game with ten food (for easy testing):

```
python gui.py --food 10
# or
python ants_gui.py --food 10
```

Now you have finished ***Phase One*** !

Phase 2: Ants!

Now that you've implemented basic gameplay with two types of `Ant`s, let's add some flavor to the ways ants can attack bees. In this phase, you'll be implementing several different `Ant`s with different offensive capabilities.

After you implement each `Ant` subclass in this section, you'll need to set its `implemented` attribute to `True` so that that type of ant will show up in the GUI. Feel free to try out the game with each new ant to test the functionality!

With your Phase 2 ants, try `python ants_gui.py -d easy` to play against a full swarm of bees in a multi-tunnel layout and try `-d normal`, `-d hard`, or `-d extra-hard` if you want a real challenge! If the bees are too numerous to vanquish, you might need to create some new ants.

Problem 4 (250pts): Short and Long Thrower

The `ThrowerAnt` is a great offensive unit, but it'd be nice to have a cheaper unit that can throw. Implement two subclasses of `ThrowerAnt` that are less costly but have constraints on the distance they can throw:

- The `LongThrower` can only `throw_at` a `Bee` that is found after following at least 5 `entrance` transitions. It cannot hit `Bee`s that are in the same `Place` as it or the first 4 `Place`s in front of it. If there are two `Bee`s, one too close to the `LongThrower` and the other within its range, the `LongThrower` should throw past the closer `Bee` and target the farther one, which is within its range.
- The `ShortThrower` can only `throw_at` a `Bee` that is found after following at most 3 `entrance` transitions. It cannot throw at any bees further than 3 `Place`s in front of it.

Neither of these specialized throwers can `throw_at` a `Bee` that is exactly 4 `Place`s away.

	Class	Food Cost	Health
	`ShortThrower`	2	1
	`LongThrower`	2	1

A good way to approach the implementation to `ShortThrower` and `LongThrower` is to have it inherit the `nearest_bee` method from the base `ThrowerAnt` class. The logic of choosing which bee a thrower ant will attack is essentially the same, except the `ShortThrower` and `LongThrower` ants have maximum and minimum ranges, respectively.

To implement these behaviors, you will need to modify the `nearest_bee` method to reference `min_range` and `max_range` attributes, and only return a bee that is in range.

Make sure to give these `min_range` and `max_range` sensible defaults in `ThrowerAnt` that do not change its behavior. Then, implement the subclasses `LongThrower` and `ShortThrower` with appropriately constrained ranges and correct food costs.

Hint: `float('inf')` returns an infinite positive value represented as a float that can be compared with other numbers.

Don't forget to set the `implemented` class attribute of `LongThrower` and `ShortThrower` to `True`.

Note: Please make sure your attributes are called `max_range` and `min_range` rather than `maximum_range` and `minimum_range` or something. The tests directly reference the variable via this name.

Before

Before writing any code, read the instructions and test your understanding of the problem:

```
python ok -q 04 -u
```

After

After writing code, test your implementation (rerun the tests for 03 to make sure they still work):

```
python ok -q 03
python ok -q 04
```

Problem 5 (250pts): Fire Ant

Implement the `FireAnt`, which does damage when it receives damage. Specifically, if it is damaged by `health` health units, it does a damage of `health` to all bees in its place (this is called *reflected damage*).

If it dies, it does an additional amount of damage, which is specified by its `damage` attribute (by default 3).

To implement this, we have to override the `FireAnt`'s `reduce_health` method. Normally, `Insect.reduce_health` will decrement the insect's `health` by the given `amount` and remove the insect from its place if `health` reaches zero or lower. However, `FireAnt` also does damage to all the bees in its place when it receives damage, with a special bonus when its health drops to 0, before being removed from its `place`.

	Class	Food Cost	Health
	`FireAnt`	5	3

Hint1: To damage a `Bee`, call the `reduce_health` method inherited from `Insect`.

Hint2: Damaging a bee may cause it to be removed from its place. If you iterate over a list, but change the contents of that list at the same time, you may not visit all the elements. This can be prevented by making a copy of the list. You can either use a list slice, or use the built-in `list` function.

```
>>> lst = [1,2,3,4]
>>> lst[:]
[1, 2, 3, 4]
>>> list(lst)
[1, 2, 3, 4]
>>> lst[:] is not lst and list(lst) is not lst
True
```

Once you've finished implementing the `FireAnt`, give it a class attribute `implemented` with the value `True`.

Note: even though you are overriding the `Insect.reduce_health` function, you can still use it in your implementation by calling it directly (rather than via `self`). Note

that this is not recursion (why?)

Before

Before writing any code, read the instructions and test your understanding of the problem:

```
python ok -q 05 -u
```

After

After writing code, test your implementation:

```
python ok -q 05
```

You can also test your program by playing a game or two! A `FireAnt` should destroy all co-located `Bee`s when it is stung. To start a game with ten food (for easy testing):

```
python gui.py --food 10
```

Phase 3: More Ants

Problem 6 (200pts): Wall Ant

We are going to add some protection to our glorious home base by implementing the `WallAnt`, which is an ant that does nothing each turn. A `WallAnt` is useful because it has a large `health` value.

	Class	Food Cost	Health
	`WallAnt`	4	4

Unlike with previous ants, we have not provided you with a class header. Implement the `WallAnt` class from scratch. Give it a class attribute `name` with the value `'Wall'` (so that the graphics work) and a class attribute `implemented` with the value `True` (so that you can use it in a game).

Before

Before writing any code, read the instructions and test your understanding of the problem:

```
python ok -q 06 -u
```

After

After writing code, test your implementation:

```
python ok -q 06
```

Problem 7 (250pts): Hungry Ant

Implement the `HungryAnt`, which will select a random `Bee` from its `place` and eat it whole. After eating a `Bee`, it must spend 3 turns chewing before eating again. If there is no bee available to eat, it will do nothing.

	Class	Food Cost	Health
	`HungryAnt`	4	1

Give `HungryAnt` a `chew_duration` class attribute that holds the number of turns that it takes a `HungryAnt` to chew (default to 3). Also, give each `HungryAnt` an instance attribute `chew_countdown` that counts the number of turns it has left to chew (default is 0, since it hasn't eaten anything at the beginning).

Implement the `action` method of the `HungryAnt` to check if it is chewing; if so, decrement its `chew_countdown` counter. Otherwise, eat a random `Bee` in its `place` by reducing the `Bee`'s health to 0 and restart the `chew_countdown` timer.

Before

Before writing any code, read the instructions and test your understanding of the problem:

```
python ok -q 07 -u
```

After

After writing code, test your implementation:

```
python ok -q 07
```

Problem 8 (250pts): Bodyguard Ant

Right now, our ants are quite frail. We'd like to provide a way to help them last longer against the onslaught of the bees. Enter the `BodyguardAnt`.

	Class	Food Cost	Health
	`BodyguardAnt`	4	2

A `BodyguardAnt` differs from a normal ant because it is a `ContainerAnt`; it can contain another ant and protect it, all in one `Place`. When a `Bee` stings the ant in a `Place` where one ant contains another, only the container is damaged. The ant inside the container can still perform its original action. If the container perishes, the contained ant still remains in the place (and can then be damaged).

Each `ContainerAnt` has an instance attribute `ant_contained` that stores the ant it contains. It initially starts off as `None`, to indicate that no ant is being protected. Implement the `store_ant` method so that it sets the bodyguard's `ant_contained` instance attribute to the passed in `ant` argument. Also implement the `ContainerAnt`'s `action` method to perform its `ant_contained`'s action if it is currently containing an ant.

In addition, you will need to make the following modifications throughout your program so that a container and its contained ant can both occupy a place at the same time (a maximum of two ants per place), but only if exactly one is a container:

1. There is an `Ant.can_contain` method, but it always returns `False`. Override the method `ContainerAnt.can_contain` so that it takes an ant other as an argument and returns `True` if:
    - This `ContainerAnt` does not already contain another ant.
    - The other ant is not a container.
2. Modify `Ant.add_to` to allow a container and a non-container ant to occupy the same place according to the following rules:
    - If the ant originally occupying a place can contain the ant being added, then both ants occupy the place and original ant contains the ant being added.
    - If the ant being added can contain the ant originally in the space, then both ants occupy the place and the (container) ant being added contains the original ant.
    - If neither Ant can contain the other, raise the same AssertionError as before (the one already present in the starter code).
    - **Important**: If there are two ants in a specific `Place`, the `ant` attribute of the

`Place` instance should refer to the container ant, and the container ant should contain the non-container ant.

3. Add a `BodyguardAnt.__init__` that sets the initial amount of health for the ant.

Hint: You may find the `is_container` attribute that each `Ant` has useful for checking if a specific `Ant` is a container. You should also take advantage of the `can_contain` method you wrote and avoid repeating code.

Note: the constructor of `ContainerAnt.__init__` is implemented as such

```
def __init__(self, *args, **kwargs):
    Ant.__init__(self, *args, **kwargs)
    self.contained_ant = None
```

As we saw in Hog, we have that `args` is bound to all positional arguments (that is all arguments not passed not with keywords, and `kwargs` is bound to all the keyword arguments. This ensures that both sets of arguments are passed to the Ant constructor).

Effectively, this means the constructor is exactly the same as `Ant.__init__` but sets `self.ant_contained = None`

Before

Before writing any code, read the instructions and test your understanding of the problem:

```
python ok -q 08 -u
```

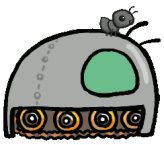
After

After writing code, test your implementation:

```
python ok -q 08
```

Problem 9 (250pts): Tank Ant

The `BodyguardAnt` provides great defense, but they say the best defense is a good offense. The `TankAnt` is a container that protects an ant in its place and also deals 1 damage to all bees in its place each turn.

	Class	Food Cost	Health
	`TankAnt`	6	2

You should not need to modify any code outside of the `TankAnt` class. If you find yourself needing to make changes elsewhere, look for a way to write your code for the previous question such that it applies not just to `BodyguardAnt` and `TankAnt` objects, but to container ants in general.

Before

Before writing any code, read the instructions and test your understanding of the problem:

```
python ok -q 9 -u
```

After

After writing code, test your implementation:

```
python ok -q 9
```

Phase 4: Water and Might

In the final phase, you're going to add one last kick to the game by introducing a new type of place and new ants that are able to occupy this place. One of these ants is the most important ant of them all: the queen of the colony!

Problem 10 (250pts): Moat...

Let's add water to the colony! Currently there are only two types of places, the `Hive` and a basic `Place`. To make things more interesting, we're going to create a new type of `Place` called `Water`.

Only an ant that is watersafe can be deployed to a `Water` place. In order to determine whether an `Insect` is watersafe, add a new attribute to the `Insect` class named `is_waterproof` that is `False` by default. Since bees can fly, make their `is_waterproof` attribute `True`, overriding the default.

Now, implement the `add_insect` method for `Water`. First, add the insect to the place regardless of whether it is watersafe. Then, if the insect is not watersafe, reduce the insect's health to 0. *Do not repeat code from elsewhere in the program.* Instead, use methods that have already been defined.

Before

Before writing any code, read the instructions and test your understanding of the problem:

```
python ok -q 10 -u
```

After

After writing code, test your implementation:

```
python ok -q 10
```

Once you've finished this problem, play a game that includes water. To access the `wet_layout` which includes water, add the `--water` option (or `-w` for short) when you start the game.

```
python gui.py --water
# or
python ants_gui.py --water
```

Problem 11 (250pts): ScubaThrower

Currently there are no ants that can be placed on `Water`. Implement the `ScubaThrower`, which is a subclass of `ThrowerAnt` that is more costly and watersafe, but otherwise identical to its base class. A `ScubaThrower` should not lose its health when placed in `Water`.

	Class	Food Cost	Health
	`ScubaThrower`	6	1

We have not provided you with a class header. Implement the `ScubaThrower` class from scratch. Give it a class attribute `name` with the value `'Scuba'` (so that the graphics work) and remember to set the class attribute `implemented` with the value `True` (so that you can use it in a game).

Before

Before writing any code, read the instructions and test your understanding of the problem:

```
python ok -q 11 -u
```

After

After writing code, test your implementation:

```
python ok -q 11
```

Problem 12 (300pts): QueenAnt

Finally, implement the `QueenAnt`. The queen is a waterproof `ScubaThrower` that inspires her fellow ants through her bravery. In addition to the standard `ScubaThrower` action, the `QueenAnt` doubles the damage of all the ants behind her each time she performs an action. Once an ant's damage has been doubled, it is *not* doubled again for subsequent turns.

Note: The reflected damage of a `FireAnt` should not be doubled, only the extra damage it deals when its health is reduced to 0.

	Class	Food Cost	Health
	`QueenAnt`	7	1

However, with great power comes great responsibility. The `QueenAnt` is governed by three special rules:

1. If the queen ever has its health reduced to 0, the ants lose. You will need to override `Ant.reduce_health` in `QueenAnt` and call `ants_lose()` in that case in order to signal to the simulator that the game is over. (The ants also still lose if any bee reaches the end of a tunnel.)
2. There can be only one queen. A second queen cannot be constructed. To check if an Ant can be constructed, we use the `Ant.construct()` class method to either construct an Ant if possible, or return `None` if not. You will need to override `Ant.construct` as a class method of `QueenAnt` in order to add this check. To keep track of whether a queen has already been created, you can use an instance variable added to the current `GameState`.
3. The queen cannot be removed. Attempts to remove the queen should have no effect (but should not cause an error). You will need to override `Ant.remove_from` in `QueenAnt` to enforce this condition.

Hint1: Think about how you can call the `construct` method of the superclass of `QueenAnt`. Remember that you ultimately want to construct a `QueenAnt`, not a regular `Ant` or a `ScubaThrower`.

Hint2: You can find each `Place` in a tunnel behind the `QueenAnt` by starting at the

ant's `place.exit` and then repeatedly following its `exit`. The `exit` of a `Place` at the end of a tunnel is `None`.

Hint3: To avoid doubling an ant's damage twice, mark the ants that have been buffed in some way, in a way that persists across calls to `QueenAnt.action`.

Hint4: When buffing the ants' damage, keep in mind that there can be more than one ant in a `Place`, such as if one ant is guarding another.

Before

Before writing any code, read the instructions and test your understanding of the problem:

```
python ok -q 12 -u
```

After

After writing code, test your implementation:

```
python ok -q 12
```

Extra Credit: Debuff

Problem 13 (300pts): Slow and Scary Thrower

Implement two final thrower ants that do zero damage, but instead apply a temporary "status" on the `action` method of a `Bee` instance that they `throw_at`. This "status" lasts for a certain number of turns, after which it ceases to take effect.

We will be implementing two new ants that inherit from ThrowerAnt.

- `SlowThrower` throws sticky syrup at a bee, slowing it for 3 turns. When a bee is slowed, it can only move on turns when `gamestate.time` is even, and can do nothing otherwise. If a bee is hit by syrup while it is already slowed, it is slowed for an additional 3 turns.
- `ScaryThrower` intimidates a nearby bee, causing it to back away instead of advancing. (If the bee is already right next to the Hive and cannot go back further, it should not move. To check if a bee is next to the Hive, you might find the `is_hive` instance attribute of `Place`s useful). Bees remain scared until they have tried to back away twice. Bees cannot try to back away if they are slowed and `gamestate.time` is odd. Once a bee has been scared once, it can't be scared ever again.

	Class	Food Cost	Health
	`SlowThrower`	4	1
	`ScaryThrower`	6	1

In order to complete the implementations of these two ants, you will need to set their class attributes appropriately and implement the `slow` and `scare` methods on `Bee`, which apply their respective statuses on a particular bee. You may also have to edit some other methods of `Bee`.

Before

Before writing any code, read the instructions and test your understanding of the

problem:

```
python ok -q EC -u
```

After

You can run some provided tests, but they are not exhaustive:

```
python ok -q EC
```

Make sure to test your code! Your code should be able to apply multiple statuses on a target; each new status applies to the current (possibly previously affected) action method of the bee.

Optional: Ninja & Laser

Optional 1: Ninja

Implement the `NinjaAnt`, which damages all `Bee`s that pass by, but can never be stung.

	Class	Food Cost	Health
	`Ninja`	5	1

A `NinjaAnt` does not block the path of a `Bee` that flies by. To implement this behavior, first modify the `Ant` class to include a new class attribute `blocks_path` that is `True` by default. Set the value of `blocks_path` to `False` in the `NinjaAnt` class.

Second, modify the `Bee`'s method `blocked` to return `False` if either there is no `Ant` in the `Bee`'s `place` or if there is an `Ant`, but its `blocks_path` attribute is `False`. Now `Bee`s will just fly past `NinjaAnts`.

Finally, we want to make the `NinjaAnt` damage all `Bee`s that fly past. Implement the `action` method in `NinjaAnt` to reduce the armor of all `Bee`s in the same `place` as the `NinjaAnt` by its `damage` attribute. Similar to the `FireAnt`, you must iterate over a list of bees that may change.

> *Hint*: Having trouble visualizing the test cases? Try drawing them out on paper! See the example in Game Layout for help.

Before

Before writing any code, read the instructions and test your understanding of the problem:

```
python ok -q optional1 -u
```

After

After writing code, test your implementation:

```
python ok -q optional1
```

For a challenge, try to win a game using only `HarvesterAnt` and `NinjaAnt`.

Optional 2: Dangerous Laser

We've been developing this ant for a long time in secret. It's so dangerous that we had to lock it in the super hidden CS61A underground vault, but we finally think it is ready to go out on the field. In this problem, you'll be implementing the final ant -- `LaserAnt`, a `ThrowerAnt` with a twist.

Note: There are no unlocking tests for this question.

	Class	Food Cost	Health
	`Laser`	10	1

The `LaserAnt` shoots out a powerful laser, damaging all that dare to stand in its path. Both `Bee`s and `Ant`s, of all types, are at risk of being damaged by `LaserAnt`. When a `LaserAnt` takes its action, it will damage all `Insect`s in its place (excluding itself, but including its container if it has one) and the `Place`s in front of it, excluding the `Hive`.

If that were it, `LaserAnt` would be too powerful for us to contain. The `LaserAnt` has a base damage of `2`. But, `LaserAnt`'s laser comes with some quirks. The laser is weakened by `0.25` each place it travels away from `LaserAnt`'s place. Additionally, `LaserAnt` has limited battery. Each time `LaserAnt` actually damages an `Insect` its laser's total damage goes down by 0.0625 (1/16). If `LaserAnt`'s damage becomes negative due to these restrictions, it simply does 0 damage instead.

The exact order in which things are damaged within a turn is unspecified.

In order to complete the implementation of this ultimate ant, read through the LaserAnt class, set the class attributes appropriately, and implement the following two functions:

1. `insects_in_front` is an instance method, called by the `action` method, that returns a dictionary where each key is an `Insect` and each corresponding value is the distance (in places) that that `Insect` is away from `LaserAnt`. The dictionary should include all `Insects` on the same place or in front of the `LaserAnt`, excluding `LaserAnt` itself.
2. `calculate_damage` is an instance method that takes in `distance`, the distance that

an insect is away from the `LaserAnt` instance. It returns the damage that the `LaserAnt` instance should afflict based on:

3. The `distance` away from the `LaserAnt` instance that an `Insect` is.
4. The number of `Insect`s that this `LaserAnt` has damaged, stored in the `insects_shot` instance attribute.

In addition to implementing the methods above, you may need to modify, add, or use class or instance attributes in the `LaserAnt` class as needed.

Before

There are no locked test for this problem.

After

You can run the provided test, but it is not exhaustive:

```
python ok -q optional2
```

Make sure to test your code!

Project Submission

At this point, run the entire autograder to see if there are any tests that don't pass:

```
python ok
```

You can also check your score on each part of the project, including the extra credit problem:

```
python ok --score
```

Once you are satisfied, submit to complete the project.

```
python ok --submit
```

You are now done with the project! If you haven't yet, you should try playing the game!

```
python gui.py [-h] [-d DIFFICULTY] [-w] [--food FOOD]
```

Acknowledgments: All materials is from cs61a. Tom Magrino and Eric Tzeng developed this project with John DeNero. Jessica Wan contributed the original artwork. Joy Jeng and Mark Miyashita invented the queen ant. Many others have contributed to the project as well!

The new concept artwork was drawn by Alana Tran, Andrew Huang, Emilee Chen, Jessie Salas, Jingyi Li, Katherine Xu, Meena Vempaty, Michelle Chang, and Ryan Davis.