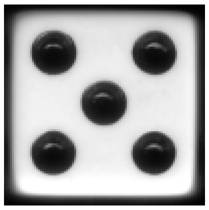
Project 1: The Game of Hog **hog.zip (hog.zip)**



I know! I'll use my
Higher-order functions to
Order higher rolls.

Before redownloading anything: Please run `python3 ok --submit` in order to back up your work, just in case that in the process of redownloading files, your work may get overwritten. If it does, you can visit okpy (https://okpy.org/) to see your backups for the project and reference your code there to restore your work so far.

Important points update: The points for the project have been updated to more accurately reflect the question weights including points for project composition. See Logistics.

If you downloaded the project zip file before **Sunday (9/5)**, running the `python3 ok --score` command will be inaccurate. There's no need to redownload the project files as this will not affect your score on our end. However, if you could like to see the changes on your end while running `--score`, you can redownload the project files and copy over the `tests/` folder from the zip. You may have to re-unlock the test cases.

Important GUI update: If you downloaded the project zip file before **Friday (9/3)**, the GUI files should be redownloaded from the current website. You can do so by redownloading the zip file from the website and copying over the gui files: `hog_gui.py` and the `gui_files` folder. Once you've done so, the gui should be working with your implementation of the project. Thanks for your patience!

Introduction

Important submission note: For full credit:

- Submit with Phase 1 complete by **Tuesday, September 7** (worth 1 pt).
- Submit with all phases complete by **Friday, September 10**.

Although Phase 1 is due only a few days before the rest of the project, you should not put off completing Phase 1. We recommend starting and finishing Phase 1 as soon as possible.

Try to attempt the problems in order, as some later problems will depend on earlier problems in their implementation and therefore also when running `ok` tests.

The entire project can be completed with a partner.

You can get 1 bonus point by submitting the entire project by **Thursday, September 9**.

In this project, you will develop a simulator and multiple strategies for the dice game Hog. You will need to use *control statements* and *higher-order functions* together, as described in Sections 1.2 through 1.6 of Composing Programs (http://composingprograms.com), the online textbook.

When students in the past have tried to implement the functions without thoroughly reading the problem description, they've often run into issues. 😱 **Read each description thoroughly before starting to code.**

Rules

In Hog, two players alternate turns trying to be the first to end a turn with at least 100 total points. On each turn, the current player chooses some number of dice to roll, up to 10. That player's score for the turn is the sum of the dice outcomes. However, a player who rolls too many dice risks:

- **Sow Sad**. If any of the dice outcomes is a 1, the current player's score for the turn is `1`.

Examples

In a normal game of Hog, those are all the rules. To spice up the game, we'll include some special rules:

- **Picky Piggy**. A player who chooses to roll zero dice scores the `n` th digit of the decimal expansion of 1/7 (0.14285714...), where `n` is the opponent's score. As a special case, if `n` is 0, the player scores 7 points.

Examples

- **Hog Pile**. After points for the turn are added to the current player's score, if the players' scores are the same, the current player's score doubles.

Examples

Final product

You can try out the online Hog GUI with the staff solution to the project at hog.cs61a.org (https://hog.cs61a.org). When you finish the project, you'll have implemented a significant part of this game yourself.

Download starter files

To get started, download all of the project code as a zip archive (hog.zip). Below is a list of all the files you will see in the archive once unzipped. For the project, you'll only be making changes to `hog.py` .

- `hog.py` : A starter implementation of Hog
- `dice.py` : Functions for rolling dice
- `hog_gui.py` : A graphical user interface (GUI) for Hog (updated)
- `ucb.py` : Utility functions for CS 61A
- `ok` : CS 61A autograder
- `tests` : A directory of tests used by `ok`
- `gui_files` : A directory of various things used by the web GUI
- `calc.py` : A file you can use to approximately test your final strategy (in progress)

You may notice some files other than the ones listed above too—those are needed for making the autograder and portions of the GUI work. Please do not modify any files other than `hog.py` .

Logistics

The project is worth 25 points. 22 points are assigned for correctness, 2 points for composition, and 1 point for submitting Phase 1 by the checkpoint date.

> **Important points update:** The points for the project have been updated to more accurately reflect the question weights including points for project composition.
>
> If you downloaded the project zip file before **Sunday (9/5)**, running the `python3 ok --score` command will be inaccurate. There's no need to redownload the project files as this will not affect your score on our end. However, if you could like to see the changes on your end while running `--score`, you can redownload the project files and copy over the `tests/` folder from the zip. You may have to re-unlock the test cases.

You will turn in the following files:

- `hog.py`

You do not need to modify or turn in any other files to complete the project. To submit the project, run the following command:

```
python3 ok --submit
```

You will be able to view your submissions on the Ok dashboard (http://ok.cs61a.org).

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 autograder tests. Also, please do not change any function signatures (names, argument order, or number of arguments).

Throughout this project, you should be testing the correctness of your code. It is good practice to test often, so that it is easy to isolate any problems. However, you should not be testing *too* often, to allow yourself time to think through problems.

We have provided an **autograder** called `ok` to help you with testing your code and tracking your progress. The first time you run the autograder, you will be asked to **log in with your Ok account using your web browser**. Please do so. Each time you run `ok`, it will back up your work and progress on our servers.

The primary purpose of `ok` is to test your implementations.

We recommend that you submit **after you finish each problem**. Only your last submission will be graded. It is also useful for us to have more backups of your code in case you run into a submission issue. **If you forget to submit, your last backup will be automatically converted to a submission.**

If you do not want us to record a backup of your work or information about your progress, you can run

```
python3 ok --local
```

With this option, no information will be sent to our course servers. If you want to test your code interactively, you can run

```
python3 ok -q [question number] -i
```

with the appropriate question number (e.g. `01`) inserted. This will run the tests for that question until the first one you failed, then give you a chance to test the functions you wrote interactively.

You can also use the debugging print feature in OK by writing

```
print("DEBUG:", x)
```

which will produce an output in your terminal without causing OK tests to fail with extra output.

Graphical User Interface

A **graphical user interface** (GUI, for short) is provided for you. At the moment, it doesn't work because you haven't implemented the game logic. Once you complete the `play` function, you will be able to play a fully interactive version of Hog!

> Note: The GUI has been updated. See the announcement at the top of the page for instructions.

Once you've done that, you can run the GUI from your terminal:

```
python3 hog_gui.py
```

Phase 1: Simulator

In the first phase, you will develop a simulator for the game of Hog.

Problem 0 (0 pt)

The `dice.py` file represents dice using non-pure zero-argument functions. These functions are non-pure because they may have different return values each time they are called. The documentation of `dice.py` describes the two different types of dice used in the project:

- A **fair** dice produces each possible outcome with equal probability. Two fair dice are already defined, `four_sided` and `six_sided`, and are generated by the `make_fair_dice` function.
- A **test** dice is deterministic: it always cycles through a fixed sequence of values that are passed as arguments. Test dice are generated by the `make_test_dice` function.

Before writing any code, read over the `dice.py` file and check your understanding by unlocking the following tests.

```
python3 ok -q 00 -u
```

This should display a prompt that looks like this:

```
=====================================================================
Assignment: Project 1: Hog
Ok, version v1.18.2
=====================================================================

~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~
Unlocking tests

At each "? ", type what you would expect the output to be.
Type exit() to quit

---------------------------------------------------------------------
Question 0 > Suite 1 > Case 1
(cases remaining: 1)

>>> test_dice = make_test_dice(4, 1, 2)
>>> test_dice()
?
```

You should type in what you expect the output to be. To do so, you need to first figure out what `test_dice` will do, based on the description above.

You can exit the unlocker by typing `exit()`.

Typing Ctrl-C on Windows to exit out of the unlocker has been known to cause problems, so avoid doing so.

In general, for each of the unlocking tests, you might find it helpful to read through the provided skeleton for that problem before attempting the unlocking test.

Problem 1 (2 pt)

Implement the `roll_dice` function in `hog.py`. It takes two arguments: a positive integer called `num_rolls` giving the number of dice to roll and a `dice` function. It returns the number of points scored by rolling the dice that number of times in a turn: either the sum of the outcomes or 1 *(Sow Sad)*.

- **Sow Sad**. If any of the dice outcomes is a 1, the current player's score for the turn is `1`.

Examples

To obtain a single outcome of a dice roll, call `dice()`. You should call `dice()` exactly `num_rolls` times in the body of `roll_dice`. **Remember to call `dice()` exactly `num_rolls` times even if Sow Sad happens in the middle of rolling.** In this way, you correctly simulate rolling all the dice together.

Understand the problem:

Before writing any code, unlock the tests to verify your understanding of the question. **Note: you will not be able to test your code using `ok` until you unlock the test cases for the corresponding question**.

```
python3 ok -q 01 -u
```

Write code and check your work:

Once you are done unlocking, begin implementing your solution. You can check your correctness with:

```
python3 ok -q 01
```

Debugging Tips

Problem 2 (3 pt)

Implement `picky_piggy`, which takes the opponent's current `score` and returns the number of points scored by rolling 0 dice.

- **Picky Piggy**. A player who chooses to roll zero dice scores the `n` th digit of the decimal expansion of 1/7 (0.14285714...), where `n` is the opponent's score. As a special case, if `n` is 0, the player scores 7 points.

Examples

The goal of this question is for you to practice retrieving the digits of a number, so it may be helpful to keep in mind the techniques used in previous assignments for digit iteration.

However, your code should not use `str`, lists, or contain square brackets `[ ]` in your implementation. Aside from this constraint, you can otherwise implement this function how you would like to.

> **Note:** Remember to remove the `"*** YOUR CODE HERE ***"` string from the function once you've implemented it so that you're not getting an unintentional `str` check error.
>
> If the syntax check isn't passing on the docstring, try upgrading your Python version to 3.8 or 3.9. It seems that the docstring being included in the check is specific to Python version 3.7, so updating your Python version should resolve the issue.
>
> **Hint:** The decimal expansion of 1/7 is a 6-digit repeating decimal with the digits 142857. Therefore, the 2nd digit is the same as the 8th digit, the 14th, 20th, 26th, 32nd, etc.

Before writing any code, unlock the tests to verify your understanding of the question.

```
python3 ok -q 02 -u
```

Once you are done unlocking, begin implementing your solution. You can check your correctness with:

```
python3 ok -q 02
```

You can also test `picky_piggy` interactively by entering `python3 -i hog.py` in the terminal and then calling `picky_piggy` with various inputs.

Problem 3 (2 pt)

Implement the `take_turn` function, which returns the number of points scored for a turn by rolling the given `dice` `num_rolls` times.

Your implementation of `take_turn` should call both `roll_dice` and `picky_piggy` when possible.

Before writing any code, unlock the tests to verify your understanding of the question.

```
python3 ok -q 03 -u
```

Once you are done unlocking, begin implementing your solution. You can check your correctness with:

```
python3 ok -q 03
```

Pair programming? (/articles/pair-programming) Remember to alternate between driver and navigator roles. The driver controls the keyboard; the navigator watches, asks questions, and suggests ideas.

Problem 4 (1 pt)

Implement `hog_pile` , which takes the current player and opponent scores and returns the points that the current player will receive due to Hog Pile. If Hog Pile is not applicable, the current player could also recieve 0 additional points.

- **Hog Pile**. After points for the turn are added to the current player's score, if the players' scores are the same, the current player's score doubles.

Examples

Before writing any code, unlock the tests to verify your understanding of the question.

```
python3 ok -q 04 -u
```

Once you are done unlocking, begin implementing your solution. You can check your correctness with:

```
python3 ok -q 04
```

Problem 5 (4 pt)

Implement the `play` function, which simulates a full game of Hog. Players take turns rolling dice until one of the players reaches the `goal` score. A turn is defined as one roll of the dice.

To determine how many dice are rolled each turn, each player uses their respective strategy (Player 0 uses `strategy0` and Player 1 uses `strategy1`). A *strategy* is a function that, given a player's score and their opponent's score, returns the number of dice that the current player will roll in the turn. Don't worry about implementing strategies yet; you'll do that in Phase 3.

Important: Your implementation should only need to use a single loop; you don't need multiple loops. This might not affect passing the test cases if your logic is correct overall, but this could affect your composition (https://cs61a.org/articles/composition/) grade for the project. Here's the section of the syllabus on composition for projects (https://cs61a.org/articles/about#projects).

Additionally, each strategy function should be called only once per turn. This means you only want to call `strategy0` when it is Player 0's turn and only call `strategy1` when it is Player 1's turn. Otherwise, the GUI and some `ok` tests may get confused.

If a player achieves the goal score by the end of their turn, i.e. after all applicable rules have been applied, the game ends. `play` will then return the final total scores of both players, with Player 0's score first and Player 1's score second.

Hints:

- You should call the functions you have implemented already.
- Call `take_turn` with four arguments (don't forget to pass in the `goal`). Only call `take_turn` once per turn.
- Call `hog_pile` to determine if the current player will gain additional points due to Hog Pile, and if so, how many points.
- You can get the number of the next player (either 0 or 1) by calling the provided function `next_player`.
- You can ignore the `say` argument to the `play` function for now. You will use it in Phase 2 of the project.
- For the unlocking tests, `hog.always_roll` refers to the `always_roll` function defined in `hog.py`.

Before writing any code, unlock the tests to verify your understanding of the question.

```
python3 ok -q 05 -u
```

Once you are done unlocking, begin implementing your solution. You can check your correctness with:

```
python3 ok -q 05
```

Once you are finished, you will be able to play a graphical version of the game. We have provided a file called `hog_gui.py` that you can run from the terminal:

```
python3 hog_gui.py
```

> Note: The GUI has been updated. See the announcement at the top of the page for instructions.

The GUI relies on your implementation, so if you have any bugs in your code, they will be reflected in the GUI. This means you can also use the GUI as a debugging tool; however, it's better to run the tests first.

Make sure to submit your work so far before the checkpoint deadline:

```
python3 ok --submit
```

Check to make sure that you did all the problems in Phase 1:

```
python3 ok --score
```

Congratulations! You have finished Phase 1 of this project!

👩‍💻👨‍💻 Pair programming? (/articles/pair-programming) This would be a good time to switch roles. Switching roles makes sure that you both benefit from the learning experience of being in each role.

Phase 2: Commentary

In the second phase, you will implement commentary functions that print remarks about the game after each turn, such as: `"22 point(s)! That's a record gain for Player 1!"`

A commentary function takes two arguments, Player 0's current score and Player 1's current score. It can print out commentary based on either or both current scores and any other information in its parent environment. Since commentary can differ from turn to turn depending on the current point situation in the game, a commentary function always returns another commentary function to be called on the next turn. The only side effect of a commentary function should be to print.

Commentary examples

The function `say_scores` in `hog.py` is an example of a commentary function that simply announces both players' scores. Note that `say_scores` returns itself, meaning that the same commentary function will be called each turn.

```
def say_scores(score0, score1):
    """A commentary function that announces the score for each player."""
    print("Player 0 now has", score0, "and Player 1 now has", score1)
    return say_scores
```

The function `announce_lead_changes` is an example of a higher-order function that returns a commentary function that tracks lead changes. A different commentary function will be called each turn.

```
def announce_lead_changes(last_leader=None):
    """Return a commentary function that announces lead changes.

    >>> f0 = announce_lead_changes()
    >>> f1 = f0(5, 0)
    Player 0 takes the lead by 5
    >>> f2 = f1(5, 12)
    Player 1 takes the lead by 7
    >>> f3 = f2(8, 12)
    >>> f4 = f3(8, 13)
    >>> f5 = f4(15, 13)
    Player 0 takes the lead by 2
    """
    def say(score0, score1):
        if score0 > score1:
            leader = 0
        elif score1 > score0:
            leader = 1
        else:
            leader = None
        if leader != None and leader != last_leader:
            print('Player', leader, 'takes the lead by', abs(score0 - score1))
        return announce_lead_changes(leader)
    return say
```

You should also understand the function `both`, which takes two commentary functions (`f` and `g`) and returns a *new* commentary function. This returned commentary function returns *another* commentary function which calls the functions returned by calling `f` and `g`, in that order.

```
def both(f, g):
    """Return a commentary function that says what f says, then what g says.

    >>> h0 = both(say_scores, announce_lead_changes())
    >>> h1 = h0(10, 0)
    Player 0 now has 10 and Player 1 now has 0
    Player 0 takes the lead by 10
    >>> h2 = h1(10, 8)
    Player 0 now has 10 and Player 1 now has 8
    >>> h3 = h2(10, 17)
    Player 0 now has 10 and Player 1 now has 17
    Player 1 takes the lead by 7
    """
    def say(score0, score1):
        return both(f(score0, score1), g(score0, score1))
    return say
```

Problem 6 (1 pt)

Update your `play` function so that a commentary function is called at the end of each turn. The return value of calling a commentary function gives you the commentary function to call on the next turn.

For example, `say(score0, score1)` should be called at the end of the first turn. Its return value (another commentary function) should be called at the end of the second turn. Each consecutive turn, call the function that was returned by the call to the previous turn's commentary function.

Hint: For the unlocking tests for this problem, remember that when calling `print` with multiple arguments, Python will put a space between each of the arguments. For example:

```
>>> print(9, 12)
9 12
```

Before writing any code, unlock the tests to verify your understanding of the question.

```
python3 ok -q 06 -u
```

Once you are done unlocking, begin implementing your solution. You can check your correctness with:

```
python3 ok -q 06
```

Problem 7 (3 pt)

Implement the `announce_highest` function, which is a higher-order function that returns a commentary function. This commentary function announces whenever a particular player gains more points in a turn than ever before. For example, `announce_highest(1)` ignores Player 0 entirely and just prints information about Player 1. (So does its return value; another commentary function about only Player 1.)

To compute the gain, it must compare the score from last turn (`last_score`) to the score from this turn for the player of interest (designated by the `who` argument). This function must also keep track of the highest gain for the player so far, which is stored as `running_high` .

The way in which `announce_highest` announces is very specific, and your implementation should match the doctests provided. Don't worry about singular versus plural when announcing point gains; you should simply use "point(s)" for both cases.

> **Hint:** The `announce_lead_changes` function provided to you is an example of how to keep track of information using commentary functions. If you are stuck, first make sure you understand how `announce_lead_changes` works.

> **Hint:** If you're getting a `local variable [var] reference before assignment` error:
>
> This happens because in Python, you aren't normally allowed to modify variables defined in parent frames. Instead of reassigning `[var]`, the interpreter thinks you're trying to define a new variable within the current frame. We'll learn about how to work around this in a future lecture, but it is not required for this problem.
>
> To fix this, you have two options:
>
> 1) Rather than reassigning `[var]` to its new value, create a new variable to hold that new value. Use that new variable in future calculations.
>
> 2) For this problem specifically, avoid this issue entirely by not using assignment statements at all. Instead, pass new values in as arguments to a call to `announce_highest` .

Before writing any code, unlock the tests to verify your understanding of the question.

```
python3 ok -q 07 -u
```

Once you are done unlocking, begin implementing your solution. You can check your correctness with:

```
python3 ok -q 07
```

When you are done, you will see commentary in the GUI:

```
python3 hog_gui.py
```

> Note: The GUI has been updated. See the announcement at the top of the page for instructions.

The commentary in the GUI is generated by passing the following function as the `say` argument to `play` .

```
both(announce_highest(0), both(announce_highest(1), announce_lead_changes()))
```

Great work! You just finished Phase 2 of the project!

👩‍💻👨‍💻 Pair programming? (/articles/pair-programming) Celebrate, take a break, and switch roles!

Phase 3: Strategies

In the third phase, you will experiment with ways to improve upon the basic strategy of always rolling a fixed number of dice. First, you need to develop some tools to evaluate strategies.

Problem 8 (2 pt)

Implement the `make_averaged` function, which is a higher-order function that takes a function `original_function` as an argument.

The return value of `make_averaged` is a function that takes in the same number of arguments as `original function` . When we call this returned function on arguments, it will return the average value of repeatedly calling `original_function` on the arguments passed in.

Specifically, this function should call `original_function` a total of `trials_count` times and return the average of the results of these calls.

Important: To implement this function, you will need to use a new piece of Python syntax. We would like to write a function that accepts an arbitrary number of arguments, and then calls another function using exactly those arguments. Here's how it works.

Instead of listing formal parameters for a function, you can write `*args`, which represents all of the **arg**ument**s** that get passed into the function. We can then call another function with these same arguments by passing these `*args` into this other function. For example:

```
>>> def printed(f):
...     def print_and_return(*args):
...         result = f(*args)
...         print('Result:', result)
...         return result
...     return print_and_return
>>> printed_pow = printed(pow)
>>> printed_pow(2, 8)
Result: 256
256
>>> printed_abs = printed(abs)
>>> printed_abs(-10)
Result: 10
10
```

Here, we can pass any number of arguments into `print_and_return` via the `*args` syntax. We can also use `*args` inside our `print_and_return` function to make another function call with the same arguments.

Read the docstring for `make_averaged` carefully to understand how it is meant to work.

Before writing any code, unlock the tests to verify your understanding of the question.

```
python3 ok -q 08 -u
```

Once you are done unlocking, begin implementing your solution. You can check your correctness with:

```
python3 ok -q 08
```

Problem 9 (2 pt)

Implement the `max_scoring_num_rolls` function, which runs an experiment to determine the number of rolls (from 1 to 10) that gives the maximum average score for a turn. Your implementation should use `make_averaged` and `roll_dice`.

If two numbers of rolls are tied for the maximum average score, return the lower number. For example, if both 3 and 6 achieve a maximum average score, return 3.

You might find it useful to read the doctest and the example shown in the doctest for this problem before doing the unlocking test.

> **Important:** In order to pass all of our tests, please make sure that you are testing dice rolls starting from 1 going up to 10, rather than starting from 10 to 1.

Before writing any code, unlock the tests to verify your understanding of the question.

```
python3 ok -q 09 -u
```

Once you are done unlocking, begin implementing your solution. You can check your correctness with:

```
python3 ok -q 09
```

Running experiments:

To run this experiment on randomized dice, call `run_experiments` using the `-r` option:

```
python3 hog.py -r
```

For the remainder of this project, you can change the implementation of `run_experiments` as you wish. The function includes calls to `average_win_rate` for evaluating various Hog strategies, but most of the calls are currently commented out. You can un-comment the calls to try out strategies, like to compare the win rate for `always_roll(8)` to the win rate for `always_roll(6)`.

Some of the experiments may take up to a minute to run. You can always reduce the number of trials in your call to `make_averaged` to speed up experiments.

Running experiments won't affect your score on the project.

Pair programming? (/articles/pair-programming) We suggest switching roles now, if you haven't recently. Almost done!

Problem 10 (1 pt)

A strategy can try to take advantage of the *Picky Piggy* rule by rolling 0 when it is most beneficial to do so. Implement `picky_piggy_strategy`, which returns 0 whenever rolling 0 would give **at least** `cutoff` points and returns `num_rolls` otherwise.

> **Hint**: You can use the function `picky_piggy` you defined in Problem 2.

Before writing any code, unlock the tests to verify your understanding of the question.

```
python3 ok -q 10 -u
```

Once you are done unlocking, begin implementing your solution. You can check your correctness with:

```
python3 ok -q 10
```

Once you have implemented this strategy, change `run_experiments` to evaluate your new strategy against the baseline. Is this strategy an improvement over the baseline?

Problem 11 (1 pt)

A strategy can also take advantage of the *Hog Pile* rules. The Hog Pile strategy always rolls 0 if doing so triggers the rule. In other cases, it rolls 0 if rolling 0 would give **at least** `cutoff` points. Otherwise, the strategy rolls `num_rolls`.

> **Hint**: You can use the function `picky_piggy_strategy` you defined in Problem 10.
>
> **Hint**: Remember that the `hog_pile` check should be done after the points from `picky_piggy` have been added to the score.

Before writing any code, unlock the tests to verify your understanding of the question.

```
python3 ok -q 11 -u
```

Once you are done unlocking, begin implementing your solution. You can check your correctness with:

```
python3 ok -q 11
```

Once you have implemented this strategy, update `run_experiments` to evaluate your new strategy against the baseline.

Optional: Problem 12 (0 pt)

Implement `final_strategy`, which combines these ideas and any other ideas you have to achieve a high win rate against the baseline strategy. Some suggestions:

- `picky_piggy_strategy` or `hog_pile_strategy` are default strategies you can start with.
- If you know the goal score (by default it is 100), there's no point in scoring more than the goal. Check whether you can win by rolling 0, 1 or 2 dice. If you are in the lead, you might decide to take fewer risks.
- Choose the `num_rolls` and `cutoff` arguments carefully.
- Take the action that is most likely to win the game.

You can check that your final strategy is valid by running `ok`.

```
python3 ok -q 12
```

You can also play against your final strategy with the graphical user interface:

```
python3 hog_gui.py
```

> Note: The GUI has been updated. See the announcement at the top of the page for instructions.

The GUI will alternate which player is controlled by you.

Project submission

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

```
python3 ok
```

You can also check your score on each part of the project:

```
python3 ok --score
```

Once you are satisfied, submit to complete the project.

```
python3 ok --submit
```

If you have a partner, make sure to add them to the project submission on okpy (https://okpy.org).

Congratulations, you have reached the end of your first CS 61A project! If you haven't already, relax and enjoy a few games of Hog with a friend.

Hog Contest

If you're interested, you can take your implementation of Hog one step further by participating in the Hog Contest, where you play your `final_strategy` against those of other students. The winning strategies will receive extra credit and will be recognized in future semesters!

To see more, read the contest description (/proj/hog_contest). Or check out the leaderboard (https://hog-contest.cs61a.org).