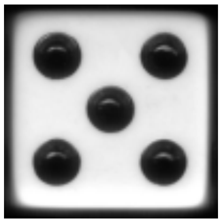
Project 1: The Game of Hog



I know! I'll use my

Higher-order functions to

Order higher rolls.

Introduction

In this project, you will develop a simulator and multiple strategies for the dice game Hog. You will need to use control and higher-order functions together, from Sections 1.1 through 1.6 of the Composing Programs online text.

In Hog, two players alternate turns trying to reach 100 points first. On each turn, the current player chooses some number of dice to roll, up to 10. Her turn score is the sum of the dice outcomes, unless any of the dice come up a 1, in which case the score for her turn is only 1 point (the **Pig out** rule).

To spice up the game, we will play with some special rules:

1. **Free bacon**. If a player chooses to roll zero dice, she scores one more than the largest digit in her opponent's score. For example, if Player 1 has 42 points, Player 0 gains 1 + max(4, 2) = 5 points by rolling zero dice. If Player 1 has 48 points, Player 0 gains 1 + max(4, 8) = 9 points.
2. **Hog wild**. If the sum of both players' total scores is a multiple of seven (e.g., 14, 21, 35), then the current player rolls four-sided dice instead of the usual six-sided dice.
3. **Swine swap**. If at the end of a turn one of the player's total score is exactly double the other's, then the players swap total scores. *Example 1:* Player 0 has 20 points and Player 1

has 5; it is Player 1's turn. She scores 5 more, bringing her total to 10. The players swap scores: Player 0 now has 10 points and Player 1 has 20. It is now Player 0's turn. *Example 2*: Player 0 has 90 points and Player 1 has 50; it is Player 0's turn. She scores 10 more, bringing her total to 100. The players swap scores, and Player 1 wins the game 100 to 50.

This project includes six files, but all of your changes will be made to the first one, and it is the only one you should need to read and understand. To get started, **download** all of the project code as a zip archive.

`hog.py`	A starter implementation of Hog.
`dice.py`	Functions for rolling dice.
`ucb.py`	Utility functions for CS 61A.
`hog_gui.py`	A graphical user interface for Hog.
`hog_grader.py`	Tests to check the correctness of your implementation.
`autograder.py`	Utility functions for grading.

Logistics

This is a two-week project. You are strongly encouraged to complete this project with a partner, although you may complete it alone.

Start early! The amount of time it takes to complete a project (or any program) is unpredictable.

You are not alone! Ask for help early and often -- the TAs, lab assistants, and your fellow students are here to help. Try attending office hours or posting on Piazza.

In the end, you and your partner will submit one project. The project is worth 20 points. 17

points are assigned for correctness, and 3 points for the overall composition of your program.

The only file that you are required to submit is `hog.py`. You do not need to modify or turn in any other files to complete the project. To submit the project, change to the directory where `hog.py` is located and run `submit proj1`. Expect a response via email whenever you submit.

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

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 finish Problem 4 (the `play` function), you will be able to play a fully interactive version of Hog!

In order to render the graphics, make sure you have Tkinter, Python's main graphics library, installed on your computer. Once you've done that, you can run the GUI from your terminal:

```
python3 hog_gui.py
```

Once you're done with Problem 9, you can play against the final strategy that you've created!

```
python3 hog_gui.py -f
```

Testing

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.

Many of the tests are contained within the docstrings of `hog.py`. Additional tests are implemented in `hog_grader.py`. To run all tests until a problem is found, run

```
python3 hog_grader.py
```

The command above runs all the tests until an error occurs, at which point it will stop and print some error messages. You can also run tests for a specific question with `-q`:

```
python3 hog_grader.py -q 1
```

Within `hog.py`, we've also provided a way to call certain functions interactively from the terminal:

```
python3 hog.py -i roll_dice
```

Phase 1: Simulator

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

Problem 1 (2 pt). Implement the `roll_dice` function in `hog.py`, which returns the number of points scored by rolling a fixed positive number of dice: either the sum of the dice or 1. To obtain a single outcome of a dice roll, call `dice()`. You should call this function *exactly* `num_rolls` times in your implementation. The only rule you need to consider for this problem is *Pig out.*

As you work, you can add `print` statements to see what is happening in your program. Remove them when you are finished.

Test your implementation before moving on:

```
python3 hog_grader.py -q 1
```

You can also run an interactive test, which allows you to type in the dice outcome, which is helpful for catching cases that are not handled in [hog_grader.py]:

```
python3 hog.py -i roll_dice
```

Problem 2 (1 pt). Implement the `take_turn` function, which returns the number of points scored for the turn. You will need to implement the *Free bacon* rule here. You can assume that `opponent_score` is less than 100. Your implementation should call `roll_dice`.

Test your implementation before moving on:

```
python3 hog_grader.py -q 2
```

You can also run `take_turn` interactively, which allows you to choose the number of rolls, the opponent's score, and the result of rolling the dice.

```
python3 hog.py -i take_turn
```

Problem 3 (1 pt). Implement `select_dice`, a helper function that will simplify the implementation of `play` (next problem). The function `select_dice` helps enforce the *Hog wild* special rule. This function takes two arguments: the scores for the current and opposing players.

Test your implementation before moving on:

```
python3 hog_grader.py -q 3
```

Problem 4 (3 pt). Finally, implement the `play` function, which simulates a full game of Hog. Players alternate turns, each using the strategy originally supplied, until one of the players reaches the `goal` score. When the game ends, `play` returns the final total scores of both players, with Player 0's score first, and Player 1's score second.

Here are some hints:

- Remember to enforce all the special rules! You should enforce the *Hog wild* special rule here (by using `select_dice`), as well as the *Swine swap* special rule here.

- You should use the `take_turn` function that you've already written.
- You can get the value of the other player (either 0 or 1) by calling the provided function `other`. For example, `other(0)` evaluates to 1.
- A *strategy* is a function that determines how many dice a player wants to roll, depending on the scores of both players. A strategy function (such as `strategy0` and `strategy1`) takes two arguments: scores for the current player and opposing player. A strategy function returns the number of dice that the current player wants to roll in the turn. Don't worry about details of implementing strategies yet. You will develop them in Phase 2.

Test your implementation before moving on:

```
python3 hog_grader.py -q 4
```

You can also run an interactive test, where you can choose how many dice to roll for both players. You will want to add `print` statements to show the result of playing the game, but be sure to remove them before moving on to Phase 2.

```
python3 hog.py -i play
```

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
```

If you don't already have Tkinter (Python's graphics library) installed, you'll need to install it first before you can run the GUI.

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.

Congratulations! You have finished Phase 1 of this project!

Phase 2: Strategies

In the second 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 5 (2 pt). Implement the `make_averaged` function. This higher-order function takes a function `fn` as an argument. It returns another function that takes the same number of arguments as the original. This returned function differs from the input function in that it returns the average value of repeatedly calling `fn` on the same arguments. This function should call `fn` a total of `num_samples` times and return the average of the results.

Note: If the input function `fn` is a non-pure function (for instance, the `random` function), then `make_averaged` will also be a non-pure function.

To implement this function, you need a new piece of Python syntax! You must write a function that accepts an arbitrary number of arguments, then calls another function using exactly those arguments. Here's how it works.

Instead of listing formal parameters for a function, we write `*args`. To call another function using exactly those arguments, we call it again with `*args`. For example,

```
>>> def printed(fn):
...     def print_and_return(*args):
...         result = fn(*args)
...         print('Result:', result)
...         return result
...     return print_and_return
>>> printed_pow = printed(pow)
>>> printed_pow(2, 8)
Result: 256
256
```

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

Test your implementation before moving on:

```
    python3 hog_grader.py -q 5
```

Problem 6 (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`. It should print out the average for each possible number of rolls, as in the doctest for `max_scoring_num_rolls`.

Test your implementation before moving on:

```
    python3 hog_grader.py -q 6
```

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

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

Running experiments

For the remainder of this project, you can change the implementation of `run_experiments` as you wish. By calling `average_win_rate`, you can evaluate various Hog strategies. For example, change the first `if False:` to `if True:` in order to evaluate `always_roll(8)` against the baseline strategy of `always_roll(5)`. You should find that it loses more often than it wins, giving a win rate below 0.5.

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

Problem 7 (1 pt). A strategy can take advantage of the *Free bacon* rule by rolling 0 when it is most beneficial to do so. Implement `bacon_strategy`, which returns 0 whenever rolling 0 would give **at least** `BACON_MARGIN` points and returns `BASELINE_NUM_ROLLS` otherwise (these two global variables are located right above the `always_roll` function).

Test your implementation before moving on:

```
python3 hog_grader.py -q 7
```

Once you have implemented this strategy, change `run_experiments` to evaluate your new strategy against the baseline. You should find that it wins more than half of the time.

Problem 8 (2 pt). A strategy can also take advantage of the *Swine swap* rule. Implement `swap_strategy`, which

1. Rolls 0 if it would cause a beneficial swap that gains points.
2. Rolls `BASELINE_NUM_ROLLS` if rolling 0 would cause a harmful swap that loses points.
3. If rolling 0 would not cause a swap, then do so if it would give **at least** `BACON_MARGIN` points and roll `BASELINE_NUM_ROLLS` otherwise.

Test your implementation before moving on:

```
python3 hog_grader.py -q 8
```

Once you have implemented this strategy, update `run_experiments` to evaluate your new strategy against the baseline. You should find that it performs even better than `bacon_strategy`, on average.

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

```
python3 hog_grader.py
```

Problem 9 (3 pt). Implement `final_strategy`, which combines these ideas and any other ideas you have to achieve a win rate of at least 0.59 against the baseline `always_roll(5)` strategy. Some ideas:

- Find a way to leave your opponent with four-sided dice more often.
- If you are in the lead, you might take fewer risks. If you are losing, you might take bigger risks to catch up.
- Vary your rolls based on whether you will be rolling four-sided or six- sided dice.

Note: You may want to increase the number of samples to improve the approximation of your win rate. The course autograder will compute your exact average win rate (without sampling error) for you once you submit your project, and it will send it to you in an email.

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

```
python3 hog_gui.py -f
```

The GUI will alternate which player is controlled by you.

Congratulations, you have reached the end of your first CS 61A project!