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 statements* and *higher-order functions* together, as described in Sections 1.2 through 1.6 of [Composing Programs](http://composingprograms.com/).

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. That player's score for the turn is the sum of the dice outcomes, unless any of the dice comes up a 1, in which case the score for the turn is only 1 point (the **Pig out** rule).

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

* **Free bacon**. A player who chooses to roll zero dice scores one more than the largest digit in the opponent's score.
  + *Example 1*: If Player 1 has 42 points, Player 0 gains 1 + max(4, 2) = 5 points by rolling zero dice.
  + *Example 2*: If Player 1 has 48 points, Player 0 gains 1 + max(4, 8) = 9 points.
  + *Example 3*: If Player 1 has 7 points, Player 0 gains 1 + max(0, 7) = 8 points by rolling zero dice.
* **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.
* **Swine Swap**. At the end of each turn, if the last two digits of Player 0's score are the reverse of the last two digits of Player 1's score, the players' score will be swapped.   
  + *Example 1*: Player 0 has a score of 19 and Player 1 has a score of 91 after Player 0 has rolled. Reversing the last two digits of Player 0's score (19) results in 91, which are the last two digits of Player 1's score. This is considered a swap and the player's scores are switched. Player 0 now has a score of 91, Player 1 now has a score of 19 and Player 0's turn is over.
  + *Example 2*: Player 0 has a score of 80 and Player 1 has a score of 8 at the end of Player 1's turn. In this example, Player 1's score is viewed as 08, which is the reverse of 80. The player's scores are swapped, leaving, Player 0 with 8 and Player 1 with 80. Player 1's turn ends.
  + *Example 3*: Player 0 begins their turn with a score of 90 while Player 1 has 70 points. Player 0 rolls 7 dice, giving them 17 points. They now have a score of 107 and Player 1 has a score of 70. Swapping the last two digits of 107 will give back 70, so the two scores are swapped. Player 0 ends their turn with a score of 70 while Player 1 now has a score of 107. Because the swap occurs before Player 0's turn is over, Player 1 wins the game.

Download starter files

To get started, download all of the project code as a [zip archive](http://cs61a.org/proj/hog/hog.zip). You only have to make changes to hog.py.

* hog.py: A starter implementation of Hog
* dice.py: Functions for rolling dice
* hog\_gui.py: A graphical user interface for Hog
* ucb.py: Utility functions for CS 61A
* ok: CS 61A autograder
* hog\_eval.py: Utility for evaluating the Hog project
* tests: A directory of tests used by ok
* images: A directory of images used by hog\_gui.py

Logistics

This is a 1-week project. You may work with one other partner. You should not share your code with students who are not your partner or copy from anyone else's solutions.

In the end, you will submit one project for both partners. The project is worth 20 points. 18 points are assigned for correctness, and 2 points for the overall [composition](http://cs61a.org/articles/composition.html) of your program.

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

python3 ok --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, please do not change any function signatures (names, argument order, or number of arguments).

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.

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, but there is a catch. At first, the test cases are*locked*. To unlock tests, run the following command from your terminal:

python3 ok -u

This command will start an interactive prompt that looks like:

=====================================================================

Assignment: The Game of Hog

OK, version ...

=====================================================================

~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~

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)

>>> Code here

?

At the ?, you can type what you expect the output to be. If you are correct, then this test case will be available the next time you run the autograder.

The idea is to understand *conceptually* what your program should do first, before you start writing any code.

Once you have unlocked some tests and written some code, you can check the correctness of your program using the tests that you have unlocked:

python3 ok

Most of the time, you will want to focus on a particular question. Use the -q option as directed in the problems below.

The tests folder is used to store autograder tests, so make sure **not to modify it**. You may lose all your unlocking progress if you do. If you need to get a fresh copy, you can download the [zip archive](http://cs61a.org/proj/hog/hog.zip) and copy it over, but you will need to start unlocking from scratch.

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!

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 complete the project, you can play against the final strategy that you've created!

python3 hog\_gui.py -f

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.pydescribes the two different types of dice used in the project:

* Dice can be fair, meaning that they produce each possible outcome with equal probability. Examples:four\_sided, six\_sided.
* For testing functions that use dice, deterministic test dice always cycle through a fixed sequence of values that are passed as arguments to the make\_test\_dice function.

Before we start writing any code, let's understand the make\_test\_dice function by unlocking its tests.

python3 ok -q 00 -u

This should display a prompt that looks like this:

=====================================================================

Assignment: Project 1: Hog

OK, version v1.3.32

=====================================================================

~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~

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.

**Note:** you can exit the unlocker by typing exit() (without quotes). **Typing Ctrl-C on Windows to exit out of the unlocker has been known to cause problems, so avoid doing so.**

Problem 1 (2 pt)

Implement the roll\_dice function in hog.py. It takes two arguments: the number of dice to roll, num\_rolls, and a dice function. It returns the number of points scored by rolling that number of dice **simultaneously**: either the sum of the outcomes or 1 (pig out).

To obtain a single outcome of a dice roll, call dice(). You must call the dice function *exactly* the number of times specified by the first argument (even if a 1 is rolled) since we are rolling all dice simultaneously in the game.

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

python3 ok -q 01 -u

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

python3 ok -q 01

The roll\_dice function has a [default argument value](http://composingprograms.com/pages/14-designing-functions.html#default-argument-values) for dice that is a random six-sided dice function. The tests use fixed dice.

Problem 2 (1 pt)

Implement the take\_turn function, which returns the number of points scored for a turn. You will need to implement the *Free bacon* rule. You can assume that opponent\_score is less than 100. For a score less than 10, assume that the first of two digits is 0. Your implementation should call roll\_dice.

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

Problem 3 (1 pt)

Implement the select\_dice function, which helps enforce the *Hog wild* special rule. This function takes two arguments: the scores for the current and opposing players. It returns either four\_sided or six\_sided dice that will be used during the turn.

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

Problem 4 (1 pt)

To help you implement the *Swine Swap* special rule, write a function called is\_swap that checks to see if the last two digits of the players' scores are swapped.

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 (3 pt)

Implement the play function, which simulates a full game of Hog. Players alternate turns, each using their respective strategy function (Player 0 uses strategy0, etc.), 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 callingselect\_dice), as well as the *Swine Swap* special rule here.
* You should use the take\_turn and is\_swap functions that you've already written.
* You can get the number of the other player (either 0 or 1) by calling the provided function other.
* A *strategy* is a function that, given a player's score and their opponent's score, returns how many dice the player wants to roll. 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.

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

**Note**: the last test for Question 5 is a *fuzz test*, which checks your play function works for any arbitrary inputs. Failing this test means something is wrong, but you should look at other tests to see where the problem might be.

Once you are finished, you will be able to play a graphical version of the game. We have provided a file calledhog\_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 6 (2 pt)

Implement the make\_averaged function, which is a higher-order function that takes a function fn as an argument. It returns another function that takes the same number of arguments as fn (the function originally passed into make\_averaged). 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\_samplestimes and return the average of the results.

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

>>> printed\_add = printed(add)

>>> printed\_add(1, 2, 3, 4)

Result: 10

10

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 06 -u

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

python3 ok -q 06

Problem 7 (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 usemake\_averaged and roll\_dice.

**Note:** 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.

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

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 ofrun\_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 inmake\_averaged to speed up experiments.

Problem 8 (1 pt)

A strategy can take advantage of the *Free bacon* rule by rolling 0 when it is most beneficial to do so. Implementbacon\_strategy, which returns 0 whenever rolling 0 would give **at least** margin points and returnsnum\_rolls otherwise.

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

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 9 (2 pt)

A strategy can also take advantage of the *Swine Swap* rule. The swap\_strategy

1. Rolls 0 if it would cause a beneficial swap.
2. Rolls num\_rolls if rolling 0 would cause a harmful swap in favor of the opponent.
3. If rolling 0 does not cause a swapped, then roll 0 if it would give **at least** margin points and rollnum\_rolls otherwise.

(Note: if a swap would result in the scores not changing, such as both players having a score of 55, it is considered to be neutral, and should be handled in case 3 as if the scores had not been swapped at all)

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

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 ok

Problem 10 (3 pt)

Implement final\_strategy, which combines these ideas and any other ideas you have to achieve a win rate of at least 0.58 (for full credit) against the baseline always\_roll(5) strategy. Partial credit is also given if you are close. Some ideas:

* You only need 100 points to win. If you are near the goal, try not to pig out and give your opponent a chance to win.
* 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.
* Find a way to leave your opponent with four-sided dice more often.

You can check your final strategy win rate by running:

python3 hog.py --final

As usual, you can also run OK to check if your win rate is high enough:

python3 ok -q 10

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!