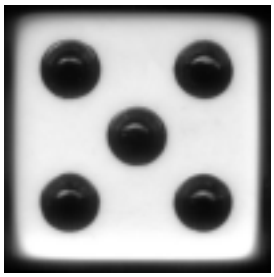
Project 1: The Game of Hog



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.

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.

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

- **Pig Out**. If any of the dice outcomes is a 1, the current player's score for the turn is the number of 1's rolled.
- **Free Bacon**. A player who chooses to roll zero dice scores one more than the largest digit in the opponent's total score.
  - *Example 1*: If the opponent has 42 points, the current player gains 1 + max(4, 2) = 5 points by rolling zero dice.
  - *Example 2*: If the opponent has 48 points, the current player gains 1 + max(4, 8) = 9 points by rolling zero dice.
  - *Example 3*: If the opponent has 7 points, the current player gains 1 + max(0, 7) = 8 points by rolling zero dice.
- **Hogtimus Prime**. If a player's score for the turn is a prime number, then the turn score is increased to the next larger prime number. For example, if the dice outcomes sum to 11, the current player scores 13 points for the turn. This boost only applies to the current player. *Note:* 1 is not a prime number!
- **When Pigs Fly**. The score for a turn is limited to 25 points minus the number of dice rolled.
  - *Example 1*: The current player rolls 10 dice which have a sum total of 35

points. The player's score for the turn is 25 - 10 = 15 points.
    - *Example 2*: The current player rolls 5 dice which have a sum total of 19 points. Hogtimus Prime bumps the turn score up to 23. However, the player's score for the turn is 25 - 5 = 20 points.
- **Hog Wild**. If the sum of both players' total scores is a multiple of seven (e.g., 0, 7, 14, 21, 35), then the current player rolls special re-rolling dice. When re-rolling dice are rolled and the outcome is odd, they are rolled again exactly once.
    - *Example 1*: The scores are 25 and 38 (which sum to 63), and the current player chooses to roll 4 dice. The outcomes are 4, 2, 6, and 2. Because all of the initial outcomes are even, no dice are re-rolled and the current player's score for the turn is 14.
    - *Example 2*: The scores are 58 and 33 (which sum to 91), and the current player chooses to roll 2 dice. The first roll outcome is 1, which is odd, so it is rolled again and comes up 4. The second roll outcome is 5, which is odd, so it is also rolled again and comes up 3. Even though 3 is an odd number, that outcome is kept, since re-rolling dice are only re-rolled once. Thus, the player has a turn score of 7.
    - *Example 3*: The scores are 0 and 0 and the current player chooses to roll 3 dice. The dice outcomes are 2, 3, and 4. The 2 and 4 remain the same while the 3 is re-rolled and comes up 1. The player's score for the turn is 1 because of Pig Out.
- **Swine Swap**. After the turn score is added, if one of the scores is double the other, then the two scores are swapped.
    - *Example 1*: The current player has a total score of 37 and the opponent has 92. The current player rolls two dice that total 9. The current player's new total score (46) is half of the opponent's score. These scores are swapped! The current player now has 92 points and the opponent has 46. The turn ends.
    - *Example 2*: The current player has 91 and the opponent has 55. The current player rolls five dice that total 17, a prime that is boosted to 19 points for the turn. The current player has 110, so the scores are swapped. The opponent ends the turn with 110 and wins the game.
- **Pork Chop**. A player may choose to roll -1 dice, which scores 1 point for the turn, but swaps the normal six-sided dice with four-sided dice for all subsequent turns. The special re-rolling dice are affected by this rule and will become four-sided as well. The odd re-rolling rule will still apply to the four-sided dice. The next time either player rolls -1 dice, the six-sided dice will be swapped back. Subsequent rolls of -1 dice will continue swapping the dice back and forth.

Download starter files

To get started, download all of the project code as a zip archive. 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
- tests: A directory of tests used by ok
- images: A directory of images used by hog_gui.py

Logistics

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.

Remember that you can earn an additional bonus point by submitting the project at least 24 hours before the deadline.

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 of your program.

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

You will be informed of the submission details but roughly the project is due for submission by **3rd week of March**.

Testing

NOTE: You have been mostly working with IDLE IDE provided with python installer. Although this project can work fine with IDLE but to use the testing facilities to max, it is advised that you try using terminal(Linux) or DOS prompt. If you are having trouble working with python on terminal you can either ask your instructor or continue working with IDLE but beware some of the instructions below might not work with IDLE as they are mentioned.

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 and your partner the time to think through problems.

We have provided an **autograder** called ok to help you with testing your code and tracking your progress. The primary purpose of ok is to test your implementations, but there are two things you should be aware of.

First, some of the test cases are *locked*. To unlock tests, run the following command from your terminal: *Note: that you have to be connected with internet for unlocking the test cases for the first time, because it might ask for your email address.*

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 ?, 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 **do not 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 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.py describes 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 v1.5.2
=====================================================================


~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~
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() (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 (1 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 the number of ones rolled (*Pig Out*).

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.

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 for dice that is a random six-sided dice function. The tests use test_dice.

Problem 2 (2 pt)

Implement the take_turn function, which returns the number of points scored for a turn by the current player. Your implementation should call roll_dice when possible.

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 the two digits is 0. To make your life easier later in the project, first implement the free_bacon helper function that returns the number of points scored by rolling 0 dice. Call free_bacon in your implementation of take_turn.

You will also need to implement the *Hogtimus Prime* rule, which applies to both regular turns and Free Bacon turns! To implement *Hogtimus Prime*, write your own helper functions above the `take_turn` function. One approach is to write two helper functions: `is_prime` and `next_prime`. There are no tests for `is_prime` and `next_prime`, but you can test them on your own using doctests that you create. Remember, 1 isn't prime!

When Pigs Fly also needs to be implemented in this function! It should be applied after the *Hogtimus Prime* rule takes effect. Remember that *When Pigs Fly* caps the score for a turn, so `take_turn` should return the minimum of `25 - num_rolls` and the uncapped score for the turn.

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 `reroll` function, which is a higher-order function that takes a `dice` function and returns a `rerolled` function. The `rerolled` function should roll the `dice` and check whether the outcome is even or odd. Any even outcome is returned, but any odd outcome is discarded, and the next roll of `dice` is returned instead.

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)

Implement the select_dice and function, which picks dice for a turn in a way that enforces the *Hog Wild* and *Pork Chop* special rules.

select_dice takes three arguments: the scores for the current and opposing players and whether four-sided dice have been swapped for the usual six-sided dice. It returns the dice function to be used in take_turn.

To account for *Hog Wild*, return reroll(dice) after selecting dice with the appropriate number of sides, as provided in the starter code.

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:

- You should use the functions you have already written! You will need to call take_turn with all three arguments.
- Enforce all the remaining special rules: *Hog Wild*, *Pork Chop*, and *Swine Swap*.
- 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, which both must be non-negative integers. A strategy function returns the number of dice that the current player wants to roll in the turn. Each strategy function should be called only once per turn. Don't worry about the 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

The last test for Question 5 is a *fuzz test*, which checks that your play function works for a large number of different inputs. Failing this test means something is wrong, but you should look at other tests to see where the problem might be.

Hint: If you fail the fuzz test, check that you're only calling take_turn once per turn!
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 6 (1 pt)

Implement the check_strategy function , which takes a strategy function as an argument and returns None. It calls the strategy with all valid inputs and verifies that

the strategy always returns a valid output. Use the provided check_strategy_roll function to raise an error with a helpful message if num_rolls is an invalid output.

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 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_samples times 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_abs = printed(abs)
>>> printed_abs(-10)
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 07 –u

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

python3 ok -q 07

Problem 8 (1 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.

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

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(4). 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 9 (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** margin points and returns num_rolls otherwise. Don't forget about the Hogtimus Prime rule!

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, change run_experiments to evaluate your new strategy against the baseline. You should find that it wins more than half of the time.

Problem 10 (2 pt)

A strategy can also take advantage of the *Swine Swap* rule. The swap_strategy rolls 0 if it would cause a beneficial swap. It also returns 0 if rolling 0 would give **at least** margin points and would not cause a swap. Otherwise, the strategy rolls num_rolls.

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, update run_experiments to evaluate your

new strategy against the baseline. You should find that it gives a significant edge over always_roll(4).

Problem 11 (3 pt)

Implement final_strategy, which combines these ideas and any other ideas you have to achieve a win rate of at least 0.80 (for full credit) against the baseline always_roll(4) strategy. Partial credit is also given if you are close. Some suggestions:

- swap_strategy is a good default strategy to start with.
- There's no point in scoring more than 100. Check for chances to win. If you are in the lead, you might take fewer risks.
- Try to force a beneficial swap.
- Choose the num_rolls and margin arguments carefully. If Hog Wild is in effect, you may want to have a different num_rolls and margin.
- It can be advantageous to avoid giving the re-rolled dice to your opponent.
- Take full advantage of the Pork Chop rule.

You can check your final strategy win rate by running OK.

python3 ok –q 11

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

python3 ok

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