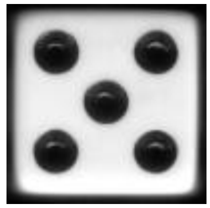
Project 1: The Game of Hog



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

Table of Contents

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 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.

  Examples: 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.

- 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.
- Hogtimus prime. If at the end of a turn the sum of the scores of both players is a prime number, then the points earned during the current turn are also added to the score of the current leader at the end of the turn. If after adding the boost, the total score happens to be another prime number, subsequent boosts are not applied. In addition, if the two scores are equal, no boost is applied to either score.

  Example 1 Player 0 has 5 points and Player 1 has 20; it is Player 0's turn. Player 0 scores 4 more points, bringing the total number of points to 29. The current leader is Player 1, who has more than Player 0's 9 points, so 4 points are added to Player 1's score. Player 0 now has 9 points, and Player 1 has 24.

  Example 2: Player 0 has 34 points and Player 1 has 29; it is Player 1's turn. Player 1 scores 8 more points, bringing the total number of points to 71. The current leader is Player 1 with 37 points, so 8 more points are added to Player 1's score. Player 0 now has 34 points, and Player 1 has 45 points.

This project includes five files and two directories, but all of your changes will be made to the first file, 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
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

This is a one-week project. You may work with one other partner. You should not share your code with students who are not your partner.

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, readers, lab assistants, and your fellow students are here to help. Try attending office hours or posting on Piazza.

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.

The only file that you will 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 your hog directory (with hog.py and ok) and run python3 ok --submit. You will be able to view your submissions on the ok dashboard.
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).

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
```

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 @berkeley.edu 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: Project 1: Hog
OK, version v1.3.7
=====================================================================

~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~
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()

?
```

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
```

To help with debugging, `ok` can also be run in interactive mode:

```
python3 ok -i
```

If an error occurs, the autograder will start an interactive Python session in the environment used for the test, so that you can explore the state of the environment.

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 and copy it over, but you will need to start unlocking from scratch.

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 q00 -u
```

This should display a prompt that looks like this:

```
=====================================================================

Assignment: Project 1: Hog

```

```
OK, version v1.3.7

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



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

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.

Once you successfully unlock all cases for this question, you can verify that the test dice work correctly by checking the tests:

```
python3 ok -q q00
```

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 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.

To test the correctness of your implementation, first unlock the tests for this problem:

```
python3 ok -q q01 -u
```

And then check that the tests pass:

```
python3 ok -q q01
```

Remember that you can start an interactive Python session if an error occurs by adding a `-i` option to the end:

```
python3 ok -q q01 -i
```

The `roll_dice` function has a [default argument value] 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.

Test your implementation before moving on:

```
python3 ok -q q02 -u


python3 ok -q q02
```

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.

Test your implementation before moving on:

```
python3 ok -q q03 -u


python3 ok -q q03
```

Problem 4 (1 pt)

To help you implement the Hogtimus prime special rule, we've written an is_prime function. The is_prime function should return True if the number is prime and False if it is not. However, there are mistakes in the implementation provided! Your job is to correct the errors. You can change the function however you wish, but the structure provided is a good place to start. You may find this debugging guide helpful.

Test and debug the given implementation before moving on:

```
python3 ok -q q04 -u


python3 ok -q q04
```

Problem 5 (3 pt)

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 calling select_dice), as well as the Hogtimus Prime 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.
- 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 ok -q q05 -u


python3 ok -q q05
```

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 (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
```

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

Test your implementation before moving on:

```
python3 ok -q q06 -u


python3 ok -q q06
```

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 use make_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.

Test your implementation before moving on:

```
python3 ok -q q07 -u


python3 ok -q q07
```

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 8 (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.

Test your implementation before moving on:

```
python3 ok -q q08 -u
```

```
python3 ok -q q08
```

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 Hogtimus prime rule. The prime_strategy

1. Rolls 0 if it would cause a beneficial boost that gains points.
2. Rolls num_rolls if rolling 0 would give a boost to the opponent.
3. If rolling 0 does not cause either score to be boosted, then do so if it would give at least margin points and roll num_rolls otherwise.

Test your implementation before moving on:

```
python3 ok -q q09 -u

python3 ok -q q09
```

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 0.56 (for full credit) against the baseline always_roll(5) strategy. (At the very least, try to achieve a win rate above 0.54 for partial credit.) 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 may want to increase the number of samples to improve the approximation of your win rate. After submitting your project to ok, you can also check your exact average win rate (without sampling error) on the submission page.

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!