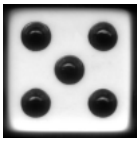
Project 1: The Game of Hog **hog.zip (hog.zip)**



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 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 1.
    - *Example 1*: The current player rolls 7 dice, 5 of which are 1's. They score 1 point for the turn.
    - *Example 2*: The current player rolls 4 dice, all of which are 3's. Since Pig Out did not occur, they score 12 points for the turn.
- **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.
- **Swine Swap**. After points for the turn are added to the current player's score, if both scores are larger than 1 and either one of the scores is a positive integer multiple of 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 opponent's score (92) is exactly twice the player's new total score (46). 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 37. The current player rolls five dice that total 20. The current player has 111, which is 3 times 37, so the scores are swapped. The opponent ends the turn with 111 and wins the game.

Download starter files

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

Logistics

This is a 2-week project. This is a solo project, so you will complete this project without a partner. You should not share your code with any other students, 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.

The project is worth 20 points. 18 points are assigned for correctness, and 2 points for the overall composition (../../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:

```
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).

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. 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, 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:

```
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.

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.

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 (hog.zip) and copy it over, but you will need to start unlocking from scratch.

If you do not want us to record a backup of your work or information about your progress, use the `--local` option when invoking `ok`. With this option, no information will be sent to our course servers.

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. Example: `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 (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 (*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`. Remember to call `dice()` exactly `num_rolls` times even if Pig Out happens in the middle of rolling. In this way, we correctly simulate rolling all the dice together.

Checking Your Work:

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
```

If the tests don't pass, it's time to debug. You can observe the behavior of your function using Python directly. First, start the Python interpreter and load the `hog.py` file.

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

Then, you can call your `roll_dice` function on any number of dice you want, such as 4.

```
>>> roll_dice(4)
```

In most systems, you can evaluate the same expression again by pressing the up arrow or Control-P, then pressing enter or return. You should find that evaluating this call expression gives a different answer each time, since dice rolls are random.

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. You can also use test dice that fix the outcomes of the dice in advance. For example, rolling twice when you know that the dice will come up 3 and 4 should give a total outcome of 7.

```
>>> fixed_dice = make_test_dice(3, 4)
>>> roll_dice(2, fixed_dice)
7
```

If you find a problem, you need to change your `hog.py` file, save it, quit Python, start it again, and then start evaluating expressions. Pressing the up arrow should give you access to your previous expressions, even after restarting Python.

Once you think that your `roll_dice` function is correct, run the `ok` tests again. Tests like these don't prove that your program is exactly correct, but they help you build confidence that this part of your program does what you expect, so that you can trust the abstraction it defines as you proceed.

Problem 2 (1 pt)

Implement the `free_bacon` helper function that returns the number of points scored by rolling 0 dice, based on the opponent's current `score`. You can assume that `score` is less than 100. For a score less than 10, assume that the first of the two digits is 0.

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 `free_bacon` interactively by entering `python3 -i hog.py` in the terminal and then calling `free_bacon` with various inputs.

Problem 3 (1 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. Call `free_bacon` in your implementation of `take_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)

Implement `is_swap`, which returns whether or not the scores should be swapped because one is an integer multiple of the other.

The `is_swap` function takes two arguments: the player scores. It returns a boolean value to indicate whether the *Swine Swap* condition is met.

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.
- Only call `take_turn` once per turn.
- Enforce all the special rules.
- You can get the number of the other player (either 0 or 1) by calling the provided function `other`.
- You can ignore the `say` argument to the `play` function for now. You will use it in Phase 2 of the project.
- 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 3.

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.

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: Commentary

In the second phase, you will implement commentary functions that print remarks about the game, such as, "22 points! That's the biggest gain yet for Player 1." A commentary function takes two arguments, the current score for Player 0 and the current score for Player 1. It returns another commentary function to be called on the next turn. It may also print some output as a side effect of being called.

The function `say_scores` in `hog.py` is an example of a commentary function. The function `announce_lead_changes` is an example of a higher-order function that returns a commentary function.

```python
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

def announce_lead_changes(previous_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 != previous_leader:
            print('Player', leader, 'takes the lead by', abs(score0 - score1))
        return announce_lead_changes(leader)
    return say
```

Problem 6 (2 pt)

Update your `play` function so that a commentary function is called at the end of each turn. `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 turn, a new commentary function should be called that is the return value of the previous call to a commentary function.

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

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

The way in which `announce_highest` announces is very specific, and your implementation should match the doctests provided. Notice in particular that if the gain is only 1 point, then the message includes "point" in singular form. If the gain is larger, then the message includes "points" in plural form.

Use Ok to test your code:

```
python3 ok -q 07
```

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.

When you are done, if play the game again, you will see the commentary.

```
python3 hog_gui.py
```

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!

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 `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 08 -u
```

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

```
python3 ok -q 08
```

Problem 9 (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 09 -u
```

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

```
python3 ok -q 09
```

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 wins slightly more often than it loses, giving a win rate around 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 10 (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.

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. You should find that it wins more than half of the time.

Problem 11 (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 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. You should find that it gives a significant edge over `always_roll(4)`.

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 `always_roll(4)` strategy. Some suggestions:

- `swap_strategy` is a good default strategy to start with.
- There's no point in scoring more than 100. Check whether you can win by rolling 0, 1 or 2 dice. 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.

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

```
python3 ok -q 12
```

You can also check your exact final winrate by running

```
python3 calc.py
```

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

```
python3 ok
```

Once you are satisfied, submit to Ok to complete the project.

```
python3 ok --submit
```

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! If you haven't already, relax and enjoy a few games of Hog with a friend.