Project 1: The Game of Hog

Introduction

In this project, you will develop a simulator and multiple strategies for the dice game Hog. You will need to implement some higher-order functions, experiment with random number generators, and generate some ASCII art.
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. She scores the sum of the dice outcomes, unless any of the dice come up a 1 (Pig out), in which case she scores only 1 point for the turn.
To spice up the game, we will play with three special rules:

1. **Hog Tied**. If the sum of both players' scores ends in a seven (e.g., 17, 27, 57), then the current player can roll at most once.
2. **Hog Wild**. If the sum of both players' 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. **Free Bacon**. If a player chooses to roll zero dice, she scores one more than the tens digit of her opponent's score. E.g., if the first player has 32 points, the second player can score four by rolling zero dice. If the opponent has fewer than 10 points (tens digit is zero), then the player scores one.

This project includes three files, but all of your changes will be made to the first one. You can 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 CS61A.

Logistics

This is a two-week project. You are 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.

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 the file called `hog.py`. You do not need to modify any other files to complete the project. To submit the project, change to the directory where the hog.py file 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).

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. To obtain a single outcome of a dice roll, call `dice()`. You should call this function *exactly* `num_rolls` times. You do not need to consider the special rules for this problem.

As you work, you can add `print` statements to see what is happening in your program.

Call `interact()` anywhere in your code to start an interactive session in the current environment. That way, you can test how different names and expressions will evaluate.

Problem 2 (1 pt). Implement the `take_turn` function, which returns the number of points scored by choosing to roll zero or more dice. To score zero dice correctly, see the*Free Bacon* special rule. For more than zero dice, call `roll_dice`.

To check your work so far, run the `take_turn_test` function by entering the following line into your terminal:

```
python3 hog.py -t
```

This function first tests `roll_dice` using deterministic test dice that always give predictable outcomes. Then, it tests `take_turn`. These tests are not exhaustive; you may still have errors in your functions even if they pass. You may add additional tests to `take_turn_test` if you wish.

Problem 3 (1 pt). You can now implement commentary for turns, which has two parts. First, update your `roll_dice` implementation to call `announce` every time dice are rolled, if `commentary` is `True`. The name `commentary` is bound in the global frame. You will need to read the docstring of `announce` to call it correctly.
Second, implement `draw_number`, which draws the outcome of a roll using text symbols. Such pictures are called *ASCII art*. *Note*: The sides with 2 and 3 dots have 2 possible depictions due to rotation. Either representation is acceptable.
A function to draw dice is actually written for you in `draw_dice`. However, it uses Python syntax that we haven't yet covered! You'll have to use this function as a black box, just by reading its docstring. Programming often involves using other people's code by reading the documentation.
You can check your work by running doctests for your whole program, by entering the following line into your terminal:

```
python3 -m doctest -v hog.py
```

Problem 4 (1 pt). Implement `num_allowed_dice` and `select_dice`, two functions that will simplify the implementation of `play` below. The function `num_allowed_dice` helps enforce special rule #1 (*Hog tied*). The function `select_dice` helps enforce special rule #2 (*Hog wild*). Both of these functions take two arguments: the scores for the current and opposing players. Make sure that doctests for these functions pass before moving on.

Problem 5 (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` should return 0 if player 0 wins, and 1 otherwise. Some hints:

- Remember to enforce the three special rules!
- The name of the current player, which should be passed as the `who` argument for `take_turn`, can be computed by calling `name`.
- 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`) take 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.

- *Important*: If a strategy returns a number of rolls greater than the maximum allowed dice for a turn, then the maximum allowed number should be passed to `take_turn`instead.
- Call the strategy function for the current player exactly once each turn. Bind the result to a local name if you want to refer to it multiple times. The interactive version of the game (see below) will prompt the user every time `strategy0` is called.

To simulate a single game in which player 0 always wants to roll 5 dice, while player 1 always wants to roll 6 dice, enter the following line into your terminal:

```
python3 hog.py -b
```

To play an interactive game of Hog against an opponent that always wants to roll 5 dice, enter the following line into your terminal:

```
python3 hog.py -p
```

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_average` function. This higher-order function takes a function `fn` as an argument, and returns another function that takes the same number of arguments as the original. It is different from the original function in that it returns the average value of repeatedly calling `fn` on its arguments. This function should call `fn` a total of `num_samples` times and return the average of their results.
Note: If the input function `fn` is a non-pure function (for instance, the `random` function), then `make_average` will also be a non-pure function.
To implement this function, you need a new element 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_average` carefully to understand how it is meant to work. Make sure that the doctests pass.
Using this function, you should now be able to
call `run_experiments` successfully. Spend some time to read this function and all of the functions it calls, so that you understand the experimental setup for the following questions.
To run a series of strategy experiments, which play many games of Hog and print the average results, enter the following line into your terminal:

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

Each game is played against the "baseline", which is the basic strategy of always rolling 5 dice. These experiments test strategies that always roll some other number of dice (the value). Most should be worse than always rolling 5.
Some of the experiments may take up to a minute to run. You can always reduce the number of random samples in `make_average` to speed up experiments.

Problem 7 (2 pt). It can be advantageous to take risks if you are behind in Hog. The "comeback" strategy rolls extra dice when losing.
Implement `make_comeback_strategy`, which returns the following strategy function: If the player is losing to the opponent by at least `margin` points, then roll `num_rolls+1`; otherwise, if the player is not losing by`margin`, roll `num_rolls`.
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). It can also be advantageous to be mean, by taking advantage of all three special rules in combination to the detriment of your opponent. Implement`make_mean_strategy`, which returns a strategy that rolls 0 whenever two conditions are true:

1. The points received from the *Free Bacon* rule are at least `min_points`, and
2. After receiving these points, the sum of player and opponent's current score will be a multiple of 7 (triggering *Hog Wild*), or end in 7 (triggering *Hog Tied*), or both.

Otherwise, roll `num_rolls`.

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

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 0.60 against the baseline`always_roll(5)` strategy. Some ideas:

- Think about how many points you receive when you choose to roll 0 dice. What should you do when you are within that many points of winning?
- If you are close to winning, perhaps you don't need to roll many dice to reach the goal.
- If you are in the lead, you might take fewer risks.

You are implementing a strategy function directly, as opposed to a function that returns a strategy. If your win rate is above 0.60, you have answered the question successfully. You can compute an approximate win rate by entering the following line in your terminal:

```
python3 hog.py -f
```

Note: You may want to increase the number of samples to improve the approximation of your win rate. The course autograder will compute your win rate for you exactly once you submit your project, and it will send it to you in an email.

Congratulations, you have reached the end of your first CS61A project!