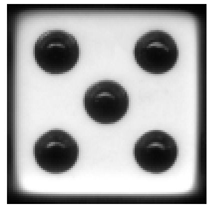
# Project 1: The Game of Hog **hog.zip (hog.zip)**



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

## Introduction

**Important submission note:** For full credit:

- Submit with Phase 1 complete by **Tuesday, September 5**, worth 1 pt.
- Submit the complete project by **Wednesday, September 13**.

Try to attempt the problems in order, as some later problems will depend on earlier problems in their implementation and therefore also when running `ok` tests.

You may complete the project with a partner.

You can get 1 bonus point by submitting the entire project by **Tuesday, September 12**. You can receive extensions on the project deadline and checkpoint deadline, but not on the early deadline, unless you're a DSP student with an accommodation for assignment extensions.

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 (https://www.composingprograms.com), the online textbook.

When students in the past have tried to implement the functions without thoroughly reading the problem description, they've often run into issues. 😱 **Read each description thoroughly before starting to code.**

## Rules

In Hog, two players alternate turns trying to be the first to end a turn with at least `GOAL` total points, where `GOAL` defaults to 100. 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. However, a player who rolls too many dice risks:

- **Sow Sad**. If any of the dice outcomes is a 1, the current player's score for the turn is `1`.

Examples

In a normal game of Hog, those are all the rules. To spice up the game, we'll include some special rules:

- **Boar Brawl**. A player who rolls zero dice scores three times the absolute difference between the tens digit of the opponent's score and the ones digit of the current player's score, or 1, whichever is higher. The ones digit refers to the rightmost digit and the tens digit refers to the second-rightmost digit. If a player's score is a single digit (less than 10), the tens digit of that player's score is 0.

Examples

- **Sus Fuss**. We call a number *sus* (https://en.wikipedia.org/wiki/Sus_%28genus%29) if it has exactly 3 or 4 factors, including 1 and the number itself. If, after rolling, the current player's score is a sus number, they gain enough points such that their score instantly increases to the next prime number.

Examples

# Download starter files

To get started, download all of the project code as a zip archive (hog.zip). Below is a list of all the files you will see in the archive once unzipped. For the project, you'll only be making changes to `hog.py`.

- `hog.py` : A starter implementation of Hog
- `dice.py` : Functions for making and rolling dice
- `hog_gui.py` : A graphical user interface (GUI) for Hog (updated)
- `ucb.py` : Utility functions for CS 61A
- `hog_ui.py` : A text-based user interface (UI) for Hog
- `ok` : CS 61A autograder
- `tests` : A directory of tests used by `ok`

- `gui_files` : A directory of various things used by the web GUI

You may notice some files other than the ones listed above too—those are needed for making the autograder and portions of the GUI work. Please do not modify any files other than `hog.py` .

# Logistics

The project is worth 25 points, of which 1 point is for submitting Phase 1 by the checkpoint date of Tuesday, September 5.

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, **submit the required files to the appropriate Gradescope assignment.**

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 or edit any files not listed above**. 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).

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.

If you want to test your code interactively, you can run

```
python3 ok -q [question number] -i
```

with the appropriate question number (e.g. `01` ) inserted. This will run the tests for that question until the first one you failed, then give you a chance to test the functions you wrote interactively.

You can also use the debugging print feature in OK by writing

```
print("DEBUG:", x)
```

which will produce an output in your terminal without causing OK tests to fail with extra output.

# 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!

Once you've done that, you can run the GUI from your terminal:

```
python3 hog_gui.py
```

# Getting Started Videos

These videos may provide some helpful direction for tackling the coding problems on this assignment.

> To see these videos, you should be logged into your berkeley.edu email.

YouTube link (https://youtu.be/playlist?list=PLx38hZJ5RLZebgMROlbtGHlmAbOjDegj5)

# Phase 1: Rules of the Game

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, and so a side-effect of calling the function may be changing what will happen when the function is called again. The documentation of `dice.py` describes the two different types of dice used in the project:

- **Fair** dice produce each possible outcome with equal probability. The `four_sided` and `six_sided` functions are examples—they have already been implemented for you in `dice.py`.
- **Test** dice are deterministic: they always cycle through a fixed sequence of values that are passed as arguments. Test dice are generated by the `make_test_dice` function.

Before writing any code, read over the `dice.py` file and check your understanding by unlocking the following tests.

```
python3 ok -q 00 -u
```

This should display a prompt that looks like this:

```
=====================================================================
Assignment: Project 1: Hog Ok, version v1.18.1
=====================================================================

~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~
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()`.

**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 times to roll a die and a `dice` function. It returns the number of points scored by rolling the die that number of times in a turn: either the sum of the outcomes or 1 *(Sow Sad)*.

- **Sow Sad**. If any of the dice outcomes is a 1, the current player's score for the turn is `1`.

Examples

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 Sow Sad happens in the middle of rolling**. By doing so, you will correctly simulate rolling all the dice together (and the user interface will work correctly).

> **Note:** The `roll_dice` function, and many other functions throughout the project, makes use of *default argument values*—you can see this in the function heading:
>
> ```
> def roll_dice(num_rolls, dice=six_sided): ...
> ```
>
> The argument `dice=six_sided` means that when `roll_dice` is called, the `dice` argument is **optional**. If no value for `dice` is provided, then `six_sided` is used by default.
>
> For example, calling `roll_dice(3, four_sided)`, or equivalently `roll_dice(3, dice=four_sided)`, simulates rolling 3 four-sided dice, while calling `roll_dice(3)` simulates rolling 3 six-sided dice.

**Understand the problem**:

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

```
python3 ok -q 01 -u
```

> **Note:** You will not be able to test your code using `ok` until you unlock the test cases for the corresponding question.

**Write code and check your work**:

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

```
python3 ok -q 01
```

Debugging Tips

## Problem 2 (2 pt)

Implement `boar_brawl`, which takes the player's current score `player_score` and the opponent's current score `opponent_score`, and returns the number of points scored by Boar Brawl when the player rolls 0 dice.

- **Boar Brawl**. A player who rolls zero dice scores three times the absolute difference between the tens digit of the opponent's score and the ones digit of the current player's score, or 1, whichever is higher. The ones digit refers to the rightmost digit and the tens digit refers to the second-rightmost digit. If a player's score is a single digit (less than 10), the tens digit of that player's score is 0.

Examples

> Don't assume that scores are below 100. Write your `boar_brawl` function so that it works correctly for any non-negative score.

> **Important:** Your implementation should **not** use `str`, lists, or contain square brackets `[ ]`. The test cases will check if those have been used.

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 `boar_brawl` interactively by running `python3 -i hog.py` from the terminal and calling `boar_brawl` on various inputs.

## Problem 3 (2 pt)

Implement the `take_turn` function, which returns the number of points scored for a turn by rolling the given `dice` `num_rolls` times.

Your implementation of `take_turn` should call both `roll_dice` and `boar_brawl` rather than repeating their implementations.

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 (2 pt)

First, implement `num_factors`, which takes in a positive integer `n` and determines the number of factors that `n` has.

> 1 and `n` are both factors of `n`!

After, implement `sus_points` and `sus_update`.

- `sus_points` takes in a player's score and returns the player's new score after applying the Sus Fuss rule (for example, `sus_points(5)` should return `5` and `sus_points(21)` should return `23`). You should use `num_factors` and the provided `is_prime` function in your implementation.
- `sus_update` returns a player's total score after they roll `num_rolls` dice, taking both Boar Brawl and Sus Fuss into account. You should use `sus_points` in this function.

> **Hint:** You can look at the implementation of `simple_update` provided in `hog.py` and use that as a starting point for your `sus_update` function.

- **Sus Fuss**. We call a number *sus* (https://en.wikipedia.org/wiki/Sus_%28genus%29) if it has exactly 3 or 4 factors, including 1 and the number itself. If, after rolling, the current player's score is a sus number, they gain enough points such that their score instantly increases to the next prime number.

Examples

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 (4 pt)

Implement the `play` function, which simulates a full game of Hog. Players take turns rolling dice until one of the players reaches the `goal` score, and the final scores of both players are returned by the function.

To determine how many dice are rolled each turn, call the current player's strategy function (Player 0 uses `strategy0` and Player 1 uses `strategy1` ). A *strategy* is a function that, given a player's score and their opponent's score, returns the number of dice that the current player will roll in the turn. An example strategy is `always_roll_5` which appears above `play` .

To determine the updated score for a player after they take a turn, call the `update` function. An `update` function takes the number of dice to roll, the current player's score, the opponent's score, and the dice function used to simulate rolling dice. It returns the updated score of the current player after they take their turn. Two examples of `update` functions are `simple_update` and `sus_update` .

If a player achieves the goal score by the end of their turn, i.e. after all applicable rules have been applied, the game ends. `play` will then return the final total scores of both players, with Player 0's score first and Player 1's score second.

Some example calls to `play` are:

- `play(always_roll_5, always_roll_5, simple_update)` simulates two players that both always roll 5 dice each turn, playing with just the Sow Sad and Boar Brawl rules.
- `play(always_roll_5, always_roll_5, sus_update)` simulates two players that both always roll 5 dice each turn, playing with the Sus Fuss rule in addition to the Sow Sad and Boar Brawl rules (i.e. all the rules).

**Important:** For the user interface to work, a strategy function should be called only once per turn. Only call `strategy0` when it is Player 0's turn and only call `strategy1` when it is Player 1's turn.

**Hints**:

- If `who` is the current player, the next player is `1 - who`.
- To call `play(always_roll_5, always_roll_5, sus_update)` and print out what happens each turn, run `python3 hog_ui.py` from the terminal.

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
```

Check to make sure that you completed all the problems in Phase 1:

```
python3 ok --score
```

Then, submit your work **to Gradescope** before the checkpoint deadline:

When you run `ok` commands, you'll still see that some tests are locked because you haven't completed the whole project yet. You'll get full credit for the checkpoint if you complete all the problems up to this point.

**Congratulations! You have finished Phase 1 of this project!**

# Interlude: User Interfaces

There are no required problems in this section of the project, just some examples for you to read and understand. See Phase 2 for the remaining project problems.

## Printing Game Events

We have built a simulator for the game, but haven't added any code to describe how the game events should be displayed to a person. Therefore, we've built a computer game that no one can play. (Lame!)

However, the simulator is expressed in terms of small functions, and we can replace each function by a version that prints out what happens when it is called. Using higher-order functions, we can do so without changing much of our original code. An example appears in `hog_ui.py`, which you are encouraged to read.

The `play_and_print` function calls the same `play` function just implemented, but using:

- new strategy functions (e.g., `printing_strategy(0, always_roll_5)`) that print out the scores and number of dice rolled.
- a new update function (`sus_update_and_print`) that prints the outcome of each turn.
- a new dice function (`printing_dice(six_sided)`) that prints the outcome of rolling the dice.

Notice how much of the original simulator code can be reused.

Running `python3 hog_ui.py` from the terminal calls `play_and_print(always_roll_5, always_roll_5)`.

## Accepting User Input

The built-in `input` function waits for the user to type a line of text and then returns that text as a string. The built-in `int` function can take a string containing the digits of an integer and return that integer.

The `interactive_strategy` function returns a strategy that let's a person choose how many dice to roll each turn by calling `input`.

With this strategy, we can finally play a game using our `play` function:

Running `python3 hog_ui.py -n 1` from the terminal calls `play_and_print(interactive_strategy(0), always_roll_5)`, which plays a game betweem a human (Player 0) and a computer strategy that always rolls 5.

Running `python3 hog_ui.py -n 2` from the terminal calls `play_and_print(interactive_strategy(0), interactive_strategy(1))`, which plays a game between two human players.

You are welcome to change `hog_ui.py` in any way you want, for example to use different strategies than `always_roll_5`.

## Graphical User Interface (GUI)

We have also provided a web-based graphical user interface for the game using a similar approach as `hog_ui.py` called `hog_gui.py`. You can run it from the terminal:

```
python3 hog_gui.py
```

Like `hog_ui.py`, the GUI relies on your simulator 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.

The source code for the Hog GUI is publicly available on Github (https://github.com/Cal-CS-61A-Staff/cs61a-apps/tree/master/hog) but involves several other programming languages: Javascript, HTML, and CSS.

# Phase 2: Strategies

In this phase, you will experiment with ways to improve upon the basic strategy of always rolling five dice. A *strategy* is a function that takes two arguments: the current player's score and their opponent's score. It returns the number of dice the player will roll, which can be from 0 to 10 (inclusive).

## Problem 6 (2 pt)

Implement `always_roll`, a higher-order function that takes a number of dice `n` and returns a strategy that always rolls `n` dice. Thus, `always_roll(5)` would be equivalent to `always_roll_5`.

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)

A strategy only has a fixed number of possible argument values. For example, in a game to 100, there are 100 possible `score` values (0-99) and 100 possible `opponent_score` values (0-99), giving 10,000 possible argument combinations.

Implement `is_always_roll`, which takes a strategy and returns whether that strategy always rolls the same number of dice for every possible argument combination up to `goal` points.

> **Reminder:** The game continues until one player reaches `goal` points (in the above example, `goal` is set to `100`). Make sure your solution accounts for every possible combination for the specified `goal` argument.

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 (2 pt)

Implement `make_averaged`, which is a higher-order function that takes a function `original_function` as an argument.

The return value of `make_averaged` is a function that takes in the same number of arguments as `original_function`. When we call this returned function on the arguments, it will return the average value of repeatedly calling `original_function` on the arguments passed in.

Specifically, this function should call `original_function` a total of `samples_count` times and return the average of the results of these calls.

**Important:** To implement this function, you will need to use a new piece of Python syntax. We would like to write a function that accepts an arbitrary number of arguments, and then calls another function using exactly those arguments. Here's how it works.

Instead of listing formal parameters for a function, you can write `*args`, which represents all of the **arg**ument**s** that get passed into the function. We can then call another function with these same arguments by passing these `*args` into this other function. For example:

```
>>> def printed(f):
...     def print_and_return(*args):
...         result = f(*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
```

Here, we can pass any number of arguments into `print_and_return` via the `*args` syntax. We can also use `*args` inside our `print_and_return` function to make another function call with the same arguments.

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 (2 pt)

Implement `max_scoring_num_rolls`, 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.

You might find it useful to read the doctest and the example shown in the doctest for this problem before doing the unlocking test.

> **Important:** In order to pass all of our tests, please make sure that you are testing dice rolls starting from 1 going up to 10, rather than from 10 to 1.

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
```

# Running Experiments

The provided `run_experiments` function calls `max_scoring_num_rolls(six_sided)` and prints the result. You will likely find that rolling 6 dice maximizes the result of `roll_dice` using six-sided dice.

To call this function and see the result, run `hog.py` with the `-r` flag:

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

In addition, `run_experiments` compares various strategies to `always_roll(6)`. You are welcome to change the implementation of `run_experiments` as you wish. Note that running experiments with `boar_strategy` and `sus_strategy` will not have accurate results until you implement them in the next two problems.

Some of the experiments may take up to a minute to run. You can always reduce the number of trials in your call to `make_averaged` to speed up experiments.

Running experiments won't affect your score on the project.

## Problem 10 (2 pt)

A strategy can try to take advantage of the *Boar Brawl* rule by rolling 0 when it is most beneficial to do so. Implement `boar_strategy`, which returns 0 whenever rolling 0 would give **at least** `threshold` points and returns `num_rolls` otherwise. This strategy should **not** also take into account the Sus Fuss rule.

> **Hint**: You can use the `boar_brawl` function you defined in Problem 2.

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
```

You should find that running `python3 hog.py -r` now shows a win rate for `boar_strategy` close to 66-67%.

## Problem 11 (2 pt)

A better strategy will take advantage of both *Boar Brawl* and *Sus Fuss* in combination. For example, if a player has 53 points and their opponent has 60, rolling 0 would bring them to 62, which is a sus number, and so they would end the turn with 67 points: a gain of 67 - 53 = 14!

The `sus_strategy` returns 0 whenever rolling 0 would result in a score that is **at least** `threshold` points more than the player's score at the start of turn.

> **Hint**: You can use the `sus_update` function.

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
```

You should find that running `python3 hog.py -r` now shows a win rate for `sus_strategy` close to 67-69%.

## 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 baseline strategy. Some suggestions:

- If you know the goal score (by default it is 100), there's no benefit to scoring more than the goal. Check whether you can win by rolling 0, 1 or 2 dice. If you are in the lead, you might decide to take fewer risks.
- Instead of using a threshold, roll 0 whenever it would give you more points on average than rolling 6.

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

```
python3 ok -q 12
```

# Project submission

Run `ok` on all problems to make sure all tests are unlocked and pass:

```
python3 ok
```

You can also check your score on each part of the project:

```
python3 ok --score
```

Once you are satisfied, submit this assignment by uploading `hog.py` **to Gradescope.** For a refresher on how to do this, refer to Lab 00 (/lab/lab00/#task-c-submitting-the-assignment).

You can add a partner to your Gradescope submission by clicking on **+ Add Group Member** under your name on the right hand side of your submission. Only one partner needs to submit to Gradescope.

**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.

/proj/hog_contest

# Hog Contest

If you're interested, you can take your implementation of Hog one step further by participating in the Hog Contest, where you play your `final_strategy` against those of other students. The winning strategies will receive extra credit and will be recognized in future semesters!

To see more, read the contest description (/proj/hog_contest). Or check out the leaderboard (https://hog-contest.cs61a.org).