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, the online textbook.

Rules of Hog

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.

However, a player who rolls too many dice risks:

- **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. The player scores 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.

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

- **Picky Piggy.** A player who chooses to roll zero dice scores the n-th digit of the decimal expansion of $1/21 (0.04761904\ldots)$ where n is the opponent's score. As a special case, if n is 0, the player scores only 1 point.
    - Example 1: The current player rolls zero dice and the opponent has a score of 3. The third digit of the decimal expansion of $1/21$ is 7, the current player will receive 7 points.
    - Example 2: The current player rolls zero dice and the opponent has a score of 13. The 13th digit of the decimal expansion of $1/21$ is 0, the current player will receive 0 points.
    - Example 3: The current player rolls zero dice and the opponent has a score of 0. The current player will receive 1 point.
- **Feral Hogs.** A player gets 1 extra point for each dice whose outcome has the same parity (i.e. both odd or both even) of the player's score.
    - Example 1: The current player has a score of 21 and rolls three dices. The outcomes are 1, 3, 5. The player will get $1 + 3 = 4$ points due to the Pig Out rule.
    - Example 2: The current player has a score of 40 and rolls four dices. The outcomes are 2, 3, 4, 5. The player will get $2 + 3 + 4 + 5 + 2 = 16$ points.
    - Example 3: The current player has a score of 60 and rolls one dice. The outcome is 3 and the player will get 3 points.
- **Swine Swap.** Define *the excitement of the game* to be three to the power of the sum of both players' scores. After points of the turn are added to the current player's score, if the excitement's first digit and last digit are the same, the scores of both players should be swapped.

- Example 1: At the end of a turn, the players have scores of 2 and 4. Since $3^{2+4} = 729$, and $7 \neq 9$, the scores are not swapped.
- Example 2: At the end of a turn, the players have scores of 11 and 1. Since $3^{11+1} = 531441$, and $5 \neq 1$, the scores are not swapped.
- Example 3: At the end of a turn, the players have scores of 23 and 4. Since $3^{23+4} = 7625597484987$, and $7 = 7$, the scores are swapped.

Project Logistics

When you finish the project, you'll have implemented a significant part of this game yourself.

To get started, download all of the project code as a zip archive from course website or from QQ group. 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 rolling dice
- `hog_gui.py`: A graphical user interface (GUI) for Hog (updated)
- `ok`: The 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`.

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:

```
$ python ok --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, 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. 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 and backup your

code.

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

```
$ python 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:

```
$ python hog_gui.py
```

Phase 1: Simulator

In the first phase, you will develop a simulator for the game of Hog.

Problem 0 (0pts)

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:

- **A fair dice** produces each possible outcome with equal probability. Two fair dice are already defined, `four_sided` and `six_sided`, and are generated by the `make_fair_dice` function.
- **A test dice** is deterministic: it always cycles 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.

```
$ python ok -q 00 -u
```

This should display a prompt that looks like this:

```
=====================================================================
Assignment: proj01: Hog
Ok, version vx.y.z
=====================================================================

~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~
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.

In general, for each of the unlocking tests, you might find it helpful to read through the provided skeleton for that problem before attempting the unlocking test.

Problem 1 (300pts)

Implement the `roll_dice` function in `hog.py`. It takes three arguments: the current player's score `cur_score`, 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), plus the number of dices that have a same parity of player's score (Feral Hogs).

- **Pig Out.** If any of the dice outcomes is a 1, the current player's score for the turn is 1.
- **Feral Hogs.** A player gets 1 extra point for each dice whose outcome has the same parity (i.e. both odd or both even) of the player's score.

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, you correctly simulate rolling all the dice together.

Understand the problem:

Before writing any code, unlock the tests to verify your understanding of the question. **Note: you will not be able to test your code using `ok` until you unlock the test cases for the corresponding question.**

```
$ python ok -q 01 -u
```

Write code and check your work:

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

```
$ python ok -q 01
```

Debugging Tips

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.

```
$ python -i hog.py
```

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

`roll_dice` function has a default argument value for `dice` that is a random six-sided dice function. Therefore, the following call to `roll_dice` simulates rolling four fair six-sided dice.

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

You will find that the previous expression may have a different result each time you call it, since it is simulating random dice rolls. 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
```

> On most systems, you can evaluate the same expression again by pressing the up arrow, then pressing enter or return. To evaluate earlier commands, press the up arrow repeatedly.
>
> If you find a problem, you need to change your `hog.py` file, save it, quit Python, start Python again, and then start evaluating expressions. Pressing the up arrow should give you access to your previous expressions, even after restarting Python.

Continue debugging your code and running the `ok` tests until they all pass. You should follow this same procedure of understanding the problem, implementing a solution, testing, and debugging for all the problems in this project.

> One more debugging tip: to start the interactive interpreter automatically upon failing an `ok` test, use `-i`. For example, `python ok -q 01 -i` will run the tests for question 1, then start an interactive interpreter with `hog.py` loaded if a test fails.

Problem 2 (300pts)

Implement `picky_piggy`, which takes the opponent's current score and returns the number of points scored by rolling 0 dice.

- **Picky Piggy.** A player who chooses to roll zero dice scores the n-th digit of the decimal expansion of $1/21 (0.04761904\ldots)$ where n is the opponent's score. As a special case, if n is 0, the player scores only 1 point.

The goal of this question is for you to practice retrieving the digits of a number, so it may be helpful to keep in mind the techniques used in previous assignments for digit iteration.

However, your code should not use `str`, lists, or contain square brackets `[ ]` in your implementation. Aside from this constraint, you can otherwise implement this function how you would like to.

> Note: Remember to remove the `"*** YOUR CODE HERE ***"` string from the function once you've implemented it so that you're not getting an unintentional str check error.
>
> If the syntax check isn't passing on the docstring, try upgrading your Python version to 3.8 or 3.9. It seems that the docstring being included in the check is specific to Python version 3.7, so updating your Python version should resolve the issue.
>
> Hint: The decimal expansion of $1/21$ is a 6-digit repeating decimal with the digits 047619. Therefore, the 2nd digit is the same as the 8th digit, the 14th, 20th, 26th, 32nd, etc.

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

```
$ python ok -q 02 -u
```

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

```
$ python ok -q 02
```

You can also test `picky_piggy` interactively by entering `python -i hog.py` in the terminal and then calling `picky_piggy` with various inputs.

Problem 3 (200pts)

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 `picky_piggy` when possible.

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

```
$ python ok -q 03 -u
```

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

```
$ python ok -q 03
```

Problem 4 (100pts)

Implement `swine_swap`, which takes the current player and opponent scores and returns `True` if the scores of both players will be swapped due to Swine Swap.

- **Swine Swap.** Define *the excitement of the game* to be three to the power of the sum of both players' scores. After points of the turn are added to the current player's score, if the excitement's first digit and last digit are the same, the scores of both players should be swapped.

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

```
$ python ok -q 04 -u
```

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

```
$ python ok -q 04
```

Problem 5 (400pts)

注意！！！助教不知道脑子哪里抽风了把错的文件打包了进去！！！

如果你是在10月14日11点之前在QQ群里获得的代码（文件名版本号小于1.6）或者在课程网站获得的代码，请重新下载最新的代码包，并且用`tests`文件夹中的`05.py`、`06.py`、……、`11.py`这些文件覆盖你之前已经解压出来的对应文件。

在解锁Problem 05测试点的过程中，你会遇到这样一个问题：

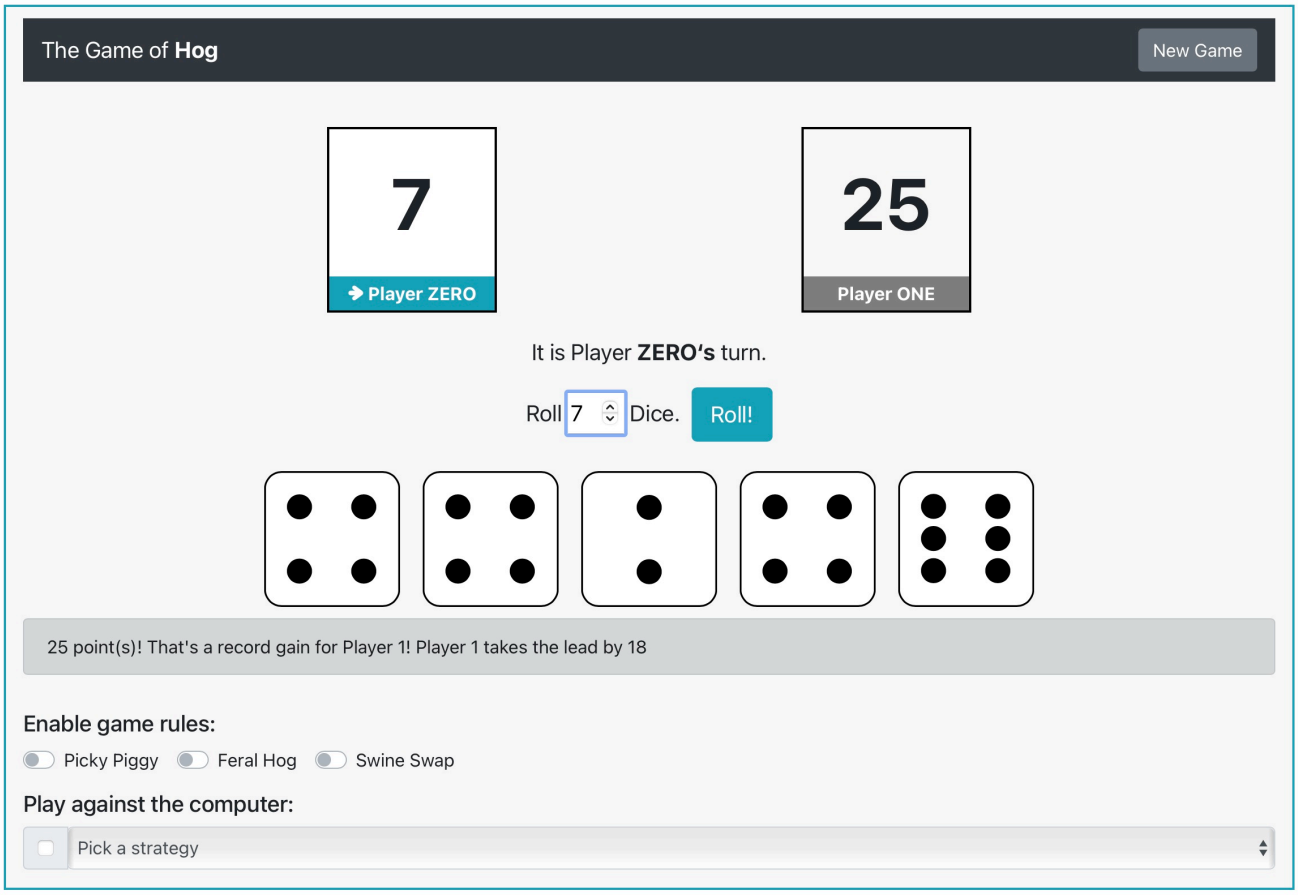
```
>>> # Hint: use Python!
>>> s0, s1 = hog.play(always(3), always(5), goal=20, dice=always_four)
>>> s0
?
```

这个问题的答案不是15。如果你输入15却显示Ok，说明你的文件是错误的，用错误的文件测试代码会导致你的代码不正确。

再注意一下！！！！！！助教不知道脑子哪里抽风了把错的文件打包了进去！！！

如果你是在10月13日23点之前在QQ群里获得的代码（文件名版本号小于1.5）或者在课程网站获得的代码，请重新下载最新的代码包，并且覆盖`hog_gui.py`这个文件和`gui_files`这个文件夹。

不操作的话，**对于你完成作业没有影响**，但是你玩不到好玩的图形界面了，嘿嘿嘿。



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. A turn is defined as one roll of the dice.

To determine how many dice are rolled each turn, each player uses their respective strategy (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. Don't worry about implementing strategies yet; you'll do that in Phase 3.

> Important: Your implementation should only need to use a single loop; you don't need multiple loops. Additionally, each strategy function should be called only once per turn. This means you only want to call strategy0 when it is Player 0's turn and only call strategy1 when it is Player 1's turn. Otherwise, the GUI and some ok tests may get confused.

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.

Hints:

- You should call the functions you have implemented already.
- Call `take_turn` with four arguments (don't forget to pass in the `goal`). Only call `take_turn` once per turn.
- Call `swine_swap` to determine if two players will swap their scores.
- You can get the number of the next player (either 0 or 1) by calling the provided function `next_player.
- You can ignore the `say` argument to the `play` function for now. You will use it in Phase 2 of the project. For the unlocking tests, `hog.always_roll` refers to the `always_roll` function defined in `hog.py`.

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

```
$ python ok -q 05 -u
```

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

```
$ python ok -q 05
```

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:

```
$ python hog_gui.py
```

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 after each turn, such as: `"22 point(s)! That's a record gain for Player 1!"`

A commentary function takes two arguments, Player 0's current score and Player 1's current score. It can print out commentary based on either or both current scores and any other information in its parent environment. Since commentary can differ from turn to turn depending on the current point situation in the game, a commentary function always returns another commentary function to be called on the next turn. The only side effect of a commentary function should be to print.

Commentary examples

The function `say_scores` in `hog.py` is an example of a commentary function that simply announces both players' scores. Note that `say_scores` returns itself, meaning that the same commentary function will be called each turn.

```
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
```

The function `announce_lead_changes` is an example of a higher-order function that returns a commentary function that tracks lead changes. A different commentary function will be called each turn.

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

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

```
def both(f, g):
    """Return a commentary function that says what f says, then what g says.

    >>> h0 = both(say_scores, announce_lead_changes())
    >>> h1 = h0(10, 0)
    Player 0 now has 10 and Player 1 now has 0
    Player 0 takes the lead by 10
    >>> h2 = h1(10, 8)
    Player 0 now has 10 and Player 1 now has 8
    >>> h3 = h2(10, 17)
    Player 0 now has 10 and Player 1 now has 17
    Player 1 takes the lead by 7
    """
    def say(score0, score1):
        return both(f(score0, score1), g(score0, score1))
    return say
```

Problem 6 (100pts)

Update your `play` function so that a commentary function is called at the end of each turn. The return value of calling a commentary function gives you the commentary function to call on the next turn.

For example, `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 consecutive turn, call the function that was returned by the call to the previous turn's commentary function.

> Hint: For the unlocking tests for this problem, remember that when calling print with multiple arguments, Python will put a space between each of the arguments. For example:

```
>>> print(9, 12)
9 12
```

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

```
$ python ok -q 06 -u
```

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

```
$ python ok -q 06
```

Problem 7 (400pts)

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. For example, `announce_highest(1)` ignores Player 0 entirely and just prints information about Player 1. (So does its return value; another commentary function about only Player 1.)

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

The way in which `announce_highest` announces is very specific, and your implementation should match the doctests provided. Don't worry about singular versus plural when announcing point gains; you should simply use "point(s)" for both cases.

> 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.

> Hint: If you're getting a `local variable [var] reference before assignment` error:
>
> This happens because in Python, you aren't normally allowed to modify variables defined in parent frames. Instead of reassigning `[var]` , the interpreter thinks you're trying to define a new variable within the current frame. We'll learn about how to work around this in a future lecture, but it is not required for this problem.
>
> To fix this, you have two options:
>
> 1. Rather than reassigning `[var]` to its new value, create a new variable to hold that new value. Use that new variable in future calculations.
> 2. For this problem specifically, avoid this issue entirely by not using assignment statements at all. Instead, pass new values in as arguments to a call to `announce_highest` .

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

```
$ python ok -q 07 -u
```

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

```
$ python ok -q 07
```

When you are done, you will see commentary in the GUI:

```
$ python 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 (200pts)

Implement the `make_averaged` function, 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 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 `trials_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.

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.

```
$ python ok -q 08 -u
```

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

```
$ python ok -q 08
```

Problem 9 (200pts)

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.

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 starting from 10 to 1.

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

```
$ python ok -q 09 -u
```

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

```
$ python ok -q 09
```

Running experiments:

To run this experiment on randomized dice, call `run_experiments` using the `-r` option:

```
$ python hog.py -r
```

For the remainder of this project, you can change the implementation of `run_experiments` as you wish. The function includes calls to `average_win_rate` for evaluating various Hog strategies, but most of the calls are currently commented out. You can un-comment the calls to try out strategies, like to compare the win rate for `always_roll(8)` to the win rate for `always_roll(6)`.

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 (100pts)

> 注意！！！在较早版本的代码中助教忘记他删除了一个函数。
>
> 如果你看到VSCode提示你有一个`hog_pile_strategy`函数未定义，直接删掉那一行就可以了。不过不删除也不会影响你本地和OJ测试的得分。

A strategy can try to take advantage of the Picky Piggy rule by rolling 0 when it is most beneficial to do so. Implement `picky_piggy_strategy`, which returns 0 whenever rolling 0 would give **at least** `cutoff` points and returns `num_rolls` otherwise.

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

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

```
$ python ok -q 10 -u
```

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

```
$ python ok -q 10
```

Once you have implemented this strategy, change `run_experiments` to evaluate your new strategy against the baseline. Is this strategy an improvement over the baseline?

Problem 11 (100pts)

A strategy can also take advantage of the Swine Swap rules. The Swine Swap strategy always rolls 0 if doing so **triggers the rule and the player's gain in this turn is at least `cutoff` points**. In other cases, the strategy rolls `num_rolls`.

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

```
$ python ok -q 11 -u
```

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

```
$ python ok -q 11
```

Once you have implemented this strategy, update `run_experiments` to evaluate your new strategy against the baseline.

Optional: Problem 12 (0pts)

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:

- `picky_piggy_strategy` or `swine_swap_strategy` are default strategies you can start with.
- If you know the goal score (by default it is 100), there's no point in 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.
- Choose the `num_rolls` and `cutoff` arguments carefully.
- Take the action that is most likely to win the game.

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

```
$ python ok -q 12
```

You can also play against your final strategy with the graphical user interface:

```
$ python hog_gui.py
```

The GUI will alternate which player is controlled by you.

Project submission

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

```
$ python ok
```

Once you are satisfied, submit to complete the project. You may submit more than once, and your final score of the project will be the highest score of all your submissions.

```
$ python ok --submit
```

Congratulations, you have reached the end of your first SICP project! If you haven't already, relax and enjoy a few games of Hog with a friend.