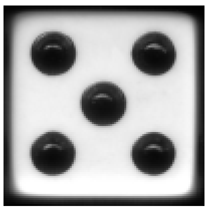
Project 1: The Game of Hog **hog.zip (hog.zip)**



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 Friday, July 3 (worth 1 pt).
- Submit with Phases 2 and 3 complete by Wednesday, July 8.

Although the checkpoint date is only a few days from the final due date, you should not put off completing Phase 1. We recommend starting and finishing Phase 1 as soon as possible to give yourself adequate time to complete Phases 2 and 3, which are can be more time consuming.

You do not have to wait until after the checkpoint date to start Phases 2 and 3.

Phase 1 is individual, Phases 2 and 3 can be completed with a partner.

You can get 1 extra credit point by submitting the entire project one day early on Tuesday, July 7.

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).

Rules

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.

Examples

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

- Free Bacon. A player who chooses to roll zero dice scores points equal to ten minus the value of the opponent score's ones digit, summed with the value of the opponent's score's tens digit.

Examples

- Feral Hogs. If the number of dice you roll is exactly 2 away (absolute difference) from the number of points you scored on the previous turn, you get 3 extra points for the turn. Treat the turn before the first turn as scoring 0 points. Do not take into account any previous feral hog bonuses or swine swap (next rule) when calculating the number of points scored the previous turn.

Examples

- Swine Swap. After points for the turn are added to the current player's score, if the absolute value of the difference between the current player score's ones digit and the opponent score's ones digit is equal to the value of the opponent score's tens digit, the scores should be swapped. A swap may occur at the end of a turn in which a player reaches the goal score, leading to the opponent winning.

Examples

Final Product

Our staff solution to the project can be interacted with at hog.cs61a.org (https://hog.cs61a.org) -- if you'd like, try it out now! When you finish the project, you'll have implemented a significant part of this game yourself!

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. However, 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`
- `gui_files` : A directory of various things used by the web gui.

Logistics

The project is worth 25 points. 22 points are assigned for correctness, 1 point for submitting Part I by the checkpoint date, and 2 points for the overall composition (https://cs61a.org//articles/composition.html).

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).

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.

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.

If you do not want us to record a backup of your work or information about your progress, you can run

```
python3 ok --local
```

With this option, no information will be sent to our course servers. 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 debug printing 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
```

Phase 1: Simulator

> Important submission note: For full credit:
>
> - submit with Phase 1 complete by Friday, July 3 (worth 1 pt).
>
> All Phase 1 tests must pass in order to receive this point.

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 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 v1.5.2
=====================================================================

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

> The Pig Out rule is reproduced below:

- Pig Out. 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 Pig Out happens in the middle of rolling. In this way, you correctly simulate rolling all the dice together.

> You can't implement the feral hogs rule in this problem, since `roll_dice` doesn't have the previous number of rolls as an argument. It will be implemented later.

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.

```
python3 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:

```
python3 ok -q 01
```

Debugging Tips

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 tens digit of a player's score is 0.

> The Free Bacon rule is reproduced below:

- Free Bacon. A player who chooses to roll zero dice scores points equal to ten minus the value of the opponent score's ones digit, summed with the value of the opponent's score's tens digit.

  Examples

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 (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 `free_bacon` when possible.

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)

Implement `is_swap`, which returns whether or not the scores should be swapped.

> The Swine Swap rule is reproduced below:

- Swine Swap. After points for the turn are added to the current player's score, if the absolute value of the difference between the current player score's ones digit and the opponent score's ones digit is equal to the value of the opponent score's tens digit, the scores should be swapped. A swap may occur at the end of a turn in which a player reaches the goal score, leading to the opponent winning.

  Examples

> Hint: The `%` operator may be useful here.

The `is_swap` function takes two arguments: the current player's score and the opponent's score. 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 5a (3 pt)

Implement the `play` function, which simulates a full game of Hog. Players alternate turns rolling dice until one of the players reaches the `goal` score.

You can ignore the Feral Hogs rule and `feral_hogs` argument for now; You'll implement it in Problem 5b.

To determine how much 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 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.

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 except for feral hogs.
- 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.

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

```
python3 ok -q 05a -u
```

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

```
python3 ok -q 05a
```

Problem 5b (1 pt)

Now, implement the Feral Hogs rule. When `play` is called and its `feral_hogs` argument is `True` , then this rule should be imposed. If `feral_hogs` is `False` , this rule should be ignored. (That way, test cases for 5a will still pass after you solve 5b.)

> The Feral Hogs rule is reproduced below:

- Feral Hogs. If the number of dice you roll is exactly 2 away (absolute difference) from the number of points you scored on the previous turn, you get 3 extra points for the turn. Treat the turn before the first turn as scoring 0 points. Do not take into account any previous feral hog bonuses or swine swap (next rule) when calculating the number of points scored the previous turn.

  Examples

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

```
python3 ok -q 05b -u
```

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

```
python3 ok -q 05b
```

Also make sure to re-run the checks for 5a. If these don't pass anymore, perhaps you're not using the `feral_hogs` argument correctly.

```
python3 ok -q 05a
```

> The last test for Question 5b 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 the issue could be caused by errors in your answers to previous problems.

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
```

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.

Make sure to submit your work so far before the checkpoint deadline:

```
python3 ok --submit
```

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

```
python3 ok --score
```

Congratulations! You have finished Phase 1 of this project!

Phase 2: Commentary

> You can work on and submit Phase 2 and 3 with a partner! Make sure one of you submits and then adds the other as a partner on okpy.org.

In the second phase, you will implement commentary functions that print remarks about the game after each turn, such as, `"22 points! That's the biggest gain yet 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.

    NOTE: the following game is not possible under the rules, it's just
    an example for the sake of the doctest

    >>> 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, 6)
    Player 0 now has 10 and Player 1 now has 6
    >>> h3 = h2(6, 17)
    Player 0 now has 6 and Player 1 now has 17
    Player 1 takes the lead by 11
    """
    def say(score0, score1):
        return both(f(score0, score1), g(score0, score1))
    return say
```

Problem 6 (2 pt)

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.

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 (3 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. E.g., `announce_highest(1)` and its return value ignore Player 0 entirely and just print information about Player 1. 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. 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.

> Note: The doctests for `both` / `announce_highest` in hog.py might describe a game that can't occur according to the rules. This shouldn't be an issue for commentary functions since they don't implement any of the rules of the game.

> 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.

```
python3 ok -q 07 -u
```

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

```
python3 ok -q 07
```

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

```
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 `original_function` as an argument. It returns another function that takes the same number of arguments as `original_function` (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 `original_function` on the same arguments. This function should call `original_function` a total of `trials_count` 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, you can write `*args`. To call another function using exactly those arguments, you call it again with `*args`. 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
```

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 (2 pt)

Implement the `max_scoring_num_rolls` function, which runs an experiment to determine the number of rolls (from 1 to 10) that gives the maximum average score for a turn. Your implementation should use `make_averaged` and `roll_dice`.

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(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.

Problem 10 (1 pt)

A strategy can try to 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 `cutoff` points and returns `num_rolls` otherwise.

> Note it is impossible for strategies to know what number of points the current player earned on the previous turn, and thus we cannot predict feral hogs. For strategies, we do not take into account bonuses from feral hogs to calculate bonuses against the cutoff or whether a swap will occur

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. Is it better than just rolling 4?

Problem 11 (2 pt)

A strategy can also take advantage of the *Swine Swap* rule. The swap strategy always rolls 0 if doing so triggers a beneficial swap and always avoids rolling 0 if doing so triggers a detrimental swap. In other cases, it rolls 0 if rolling 0 would give at least `cutoff` points. Otherwise, the strategy rolls `num_rolls`.

> Note it is impossible for strategies to know what number of points the current player earned on the previous turn, and thus we cannot predict feral hogs. For strategies, we do not take into account bonuses from feral hogs to calculate bonuses against the cutoff or whether a swap will occur
>
> Hint: a tie is technically a "swap" (e.g., 43 being swapped with 43), but is considered neither detrimental nor beneficial for the purposes of this problem.

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 rolling more than 0 dice.
- 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.

```
python3 ok -q 12
```

> Note: `calc.py` does not currently work. We're fixing a bug with the server and will make it work soon.

You will also eventually be able to check your exact final win rate by running

```
python3 calc.py
```

This should pop up a window asking for you to confirm your identity, and then it will print out a win rate for your final strategy.

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
```

If you have a partner, make sure to add them to the submission on okpy.org.

Check to make sure that you did all the problems by running

```
python3 ok --score
```

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

```
python3 hog_gui.py
```

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.

Hog Strategy 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 (/~cs61a/su20/proj/hog_contest). Or simply check out the leaderboard (https://hog-contest.cs61a.org).

CS 61A (/~cs61a/su20/)

Weekly Schedule (/~cs61a/su20/weekly.html)

Office Hours (/~cs61a/su20/office-hours.html)

Staff (/~cs61a/su20/staff.html)

Resources (/~cs61a/su20/resources.html)

Studying Guide (/~cs61a/su20/articles/studying.html)

Debugging Guide (/~cs61a/su20/articles/debugging.html)

Composition Guide (/~cs61a/su20/articles/composition.html)

Policies (/~cs61a/su20/articles/about.html)

Assignments (/~cs61a/su20/articles/about.html#assignments)

Exams (/~cs61a/su20/articles/about.html#exams)

Grading (/~cs61a/su20/articles/about.html#grading)