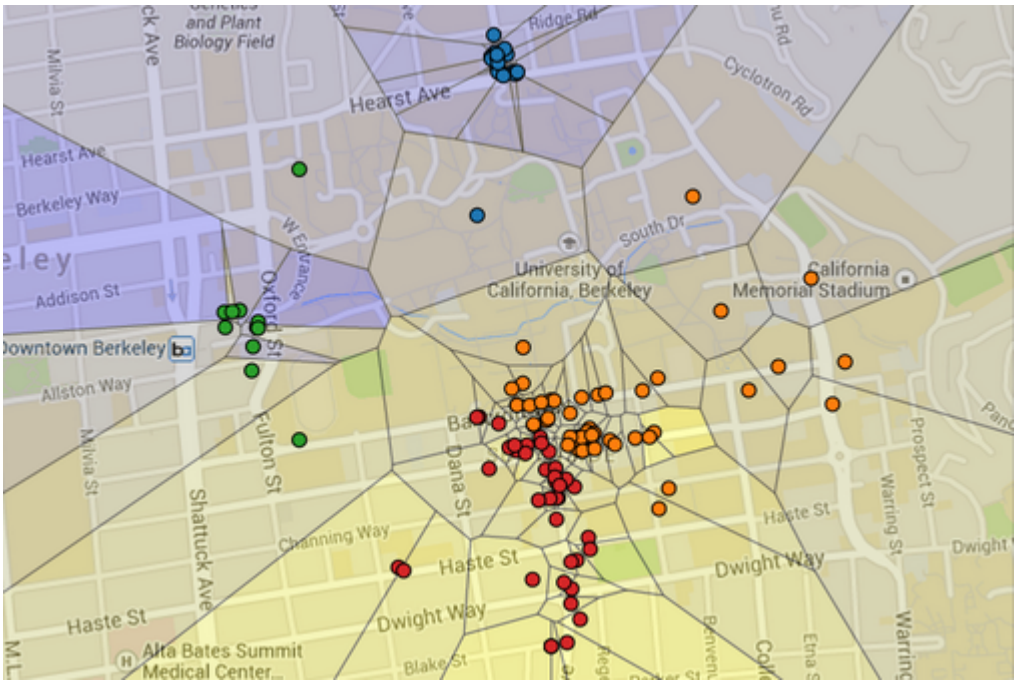
Project 2: Yelp Maps **maps.zip (maps.zip)**



Let's go out to eat!
Show me places I would like
By learning my tastes.

Introduction

In this project, you will create a visualization of restaurant ratings using machine learning and the Yelp academic dataset (https://www.yelp.com/academic_dataset). In this visualization, Berkeley is segmented into regions, where each region is shaded by the predicted rating of the closest restaurant (yellow is 5 stars, blue is 1 star). Specifically, the visualization you will be constructing is a Voronoi diagram (https://en.wikipedia.org/wiki/Voronoi_diagram).

In the map above, each dot represents a restaurant. The color of the dot is determined by the restaurant's location. For example, downtown restaurants are colored green. The user that generated this map has a strong preference for Southside restaurants, and so the southern regions are colored yellow.

This project uses concepts from Sections 2.1 (http://composingprograms.com/pages/21-introduction.html), 2.2 (http://composingprograms.com/pages/22-data-abstraction.html), 2.3 (http://composingprograms.com/pages/23-sequences.html), and 2.4.3 (http://composingprograms.com/pages/24-mutable-data.html#dictionaries) of Composing

Programs (http://composingprograms.com/). It also introduces techniques and concepts from *machine learning*, a growing field at the intersection of computer science and statistics that analyzes data to find patterns and make predictions.

Download starter files

The maps.zip (maps.zip) archive contains all the starter code and data sets. The project uses several files, but all of your changes will be made to `utils.py`, `abstractions.py`, and `recommend.py`.

- `abstractions.py`: Data abstractions used in the project
- `recommend.py`: Machine learning algorithms and data processing
- `utils.py`: Utility functions for data processing
- `ucb.py`: Utility functions for CS 61A
- `data`: A directory of Yelp users, restaurants, and reviews
- `users`: A directory of user files
- `ok`: The autograder
- `proj2.ok`: The `ok` configuration file
- `tests`: A directory of tests used by `ok`
- `visualize`: A directory of tools for drawing the final visualization

Logistics

This is a 10-day project, due on **Thursday, 02/28/19 at 11:59 PM**. This is a solo project, so you will complete this project without a partner. You should not share your code with any other students, or copy from anyone else's solutions.

Remember that you can earn an additional bonus point by submitting the project at least 24 hours before the deadline.

The project is worth 20 points. 18 points are assigned for correctness, and 2 points for the overall composition (../../articles/composition.html) of your program.

You will turn in the following files:

- `utils.py`
- `abstractions.py`
- `recommend.py`

You do not need to modify or turn in any other files to complete the project. To submit the project, run the following command:

```
python3 ok --submit
```

You will be able to view your submissions on the Ok dashboard (http://ok.cs61a.org).

For the functions that we ask you to complete, there may be some initial code that we provide. If you would rather not use that code, feel free to delete it and start from scratch. You may also add new function definitions as you see fit.

However, please do **not** modify any other functions. Doing so may result in your code failing our autograder tests. Also, please do not change any function signatures (names, argument order, or number of arguments).

Testing

Throughout this project, you should be testing the correctness of your code. It is good practice to test often, so that it is easy to isolate any problems. However, you should not be testing *too* often, to allow yourself time to think through problems.

We have provided an **autograder** called `ok` to help you with testing your code and tracking your progress. The first time you run the autograder, you will be asked to **log in with your Ok account using your web browser**. Please do so. Each time you run `ok`, it will back up your work and progress on our servers.

The primary purpose of `ok` is to test your implementations, but there are two things you should be aware of.

First, some of the test cases are *locked*. To unlock tests, run the following command from your terminal:

```
python3 ok -u
```

This command will start an interactive prompt that looks like:

```
=====================================================================
Assignment: Yelp Maps
Ok, version ...
=====================================================================

~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~
Unlocking tests

At each "? ", type what you would expect the output to be.
Type exit() to quit

---------------------------------------------------------------------
Question 0 > Suite 1 > Case 1
(cases remaining: 1)

>>> Code here
?
```

At the `?`, you can type what you expect the output to be. If you are correct, then this test case will be available the next time you run the autograder.

The idea is to understand *conceptually* what your program should do first, before you start writing any code.

Once you have unlocked some tests and written some code, you can check the correctness of your program using the tests that you have unlocked:

```
python3 ok
```

Most of the time, you will want to focus on a particular question. Use the `-q` option as directed in the problems below.

We recommend that you submit **after you finish each problem**. Only your last submission will be graded. It is also useful for us to have more backups of your code in case you run into a submission issue.

The `tests` folder is used to store autograder tests, so **do not modify it**. You may lose all your unlocking progress if you do. If you need to get a fresh copy, you can download the zip archive (maps.zip) and copy it over, but you will need to start unlocking from scratch.

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

Phase 0: Utilities

> All changes in this phase will be made to `utils.py`.

Problem 0 (2 pt)

Before starting the core project, familiarize yourself with some Python features by implementing some functions in `utils.py`. Each function described below can be implemented in one line. These will be helpful in later parts of the project.

As you work through this phase, check your understanding of the problems by unlocking the test cases:

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

Then, check your implementation by running the tests:

```
python3 ok -q 00
```

Problem 0.1: Using list comprehensions

A list comprehension constructs a new list from an existing sequence by first filtering the given sequence, and then computing an element of the result for each remaining element that is not filtered out. A list comprehension has the following syntax:

```
[<map expression> for <name> in <sequence expression> if <filter expression>]
```

For example, if we wanted to square every even integer in `range(10)`, we could write:

```
>>> [x * x for x in range(10) if x % 2 == 0]
[0, 4, 16, 36, 64]
```

Implement `map_and_filter`, which takes in a sequence `s`, a one-argument function `map_fn`, and a one-argument function `filter_fn`. It returns a new list containing the result of calling `map_fn` on each element of `s` for which `filter_fn` returns a true value. *Make sure your solution is only one line and uses a list comprehension.*

Problem 0.2: Using `min`

The built-in `min` function takes a collection of elements (such as a list or a dictionary) and returns the collection's smallest element. The `min` function also takes in an optional keyword argument called `key`, which must be a one-argument function. The `key` function is called on each element of the collection, and the return values are used for comparison. For example:

```
>>> min([-1, 0, 1])                    # no key argument; return smallest input
-1
>>> min([-1, 0, 1], key=lambda x: x*x) # return input with the smallest square
0
```

Implement `key_of_min_value`, which takes in a dictionary `d` and returns the key that corresponds to the minimum value in `d`. This behavior differs from just calling `min` on a dictionary, which would return the smallest key. *Make sure your solution is only one line and uses the `min` function.*

Problem 0.3: Using `zip`

The `zip` function defined in `utils.py` takes multiple sequences as arguments and returns a list of lists, where the i-th list contains the i-th element of each original sequence. For example:

```
>>> zip([1, 2, 3], [4, 5, 6])
[[1, 4], [2, 5], [3, 6]]
>>> for triple in zip(['a', 'b', 'c'], [1, 2, 3], ['do', 're', 'mi']):
...     print(triple)
['a', 1, 'do']
['b', 2, 're']
['c', 3, 'mi']
```

Use the `zip` function to implement `enumerate`, which pairs the elements of a sequence with their indices, offset by a starting value. `enumerate` takes a sequence `s` and a starting value `start`. It returns a list of pairs, in which the i-th element is `i + start` paired with the i-th element of `s`. For example:

```
>>> enumerate(['maps', 21, 47], start=1)
>>> [[1, 'maps'], [2, 21], [3, 47]]
```

Make sure your solution is only one line and uses the `zip` function and a `range`.

> *Note*: `zip` and `enumerate` are also built-in Python functions, but their behavior is slightly different than the versions provided in `utils.py`. The behavior of the built-in variants will be described later in the course.

Problem 1 (1 pt)

Implement the `mean` function which takes in a sequence of numbers, `s`, and returns the arithmetic mean, or average, of that sequence. The sequence cannot be empty: add an `assert` statement to ensure that empty sequences are not allowed.

As a reminder, here is an example of how an `assert` statement might be used to ensure that the input to a function is positive:

```
>>> def factorial(n):
...     assert n > 0, "Input must be positive"
...     if n <= 1:
...         return 1
...     return n * factorial(n - 1)
>>> factorial(-1)
AssertionError: Input must be positive
```

Use Ok to unlock and test your code:

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

Phase 1: Data Abstraction

> All changes in this phase will be made to `abstractions.py`.

In this phase, we will implement data abstractions to represent a restaurant and its relevant features (such as name, location, and reviews).

Problem 2 (1 pt)

Complete the implementations of the constructor and selectors for the *restaurant* data abstraction in `abstractions.py`. Two of the data abstractions have already been completed for you: the *review* data abstraction and the *user* data abstraction. Make sure that you understand how they work.

You can use any implementation you choose, but the constructor and selectors must be defined together such that the restaurant selectors return the correct field from the constructed restaurant.

- `make_restaurant`: return a restaurant constructed from five arguments:
  - `name` (a string)
  - `location` (a list containing latitude and longitude)
  - `categories` (a list of strings)
  - `price` (a number)
  - `reviews` (a list of review data abstractions created by `make_review`)
- `restaurant_name`: return the name of a `restaurant`
- `restaurant_location`: return the location of a `restaurant`
- `restaurant_categories`: return the categories of a `restaurant`
- `restaurant_price`: return the price of a `restaurant`
- `restaurant_ratings`: return a list of ratings (numbers)

Use Ok to unlock and test your code:

```
python3 ok -q 02 -u
python3 ok -q 02
```

When you finish, you should be able to generate a visualization of all restaurants rated by a user. Use `-u` to select a user from the `users` directory. You can even create your own by copying one of the `.dat` files!

```
python3 recommend.py
python3 recommend.py -u one_cluster
```

Omitting the `-u` argument will default to user `test_user`.

You may have to refresh your browser to update the visualization.

Phase 2: Unsupervised Learning

> All changes in this phase will be made to `recommend.py`.

Restaurants tend to appear in *clusters* (e.g. Southside restaurants, Downtown Berkeley restaurants, Gourmet Ghetto (http://www.gourmetghetto.org/), etc.). In this phase, we will devise a way to group together restaurants that are close to each other into these clusters.

To do so, you will be implementing the **k-means algorithm**, a method for grouping data points into clusters by determining their center positions (which are called *centroids*).

K-means is considered an *unsupervised* learning method because the algorithm is not told what the correct clusters are; it must infer the clusters from the data alone.

The k-means algorithm begins by choosing `k` centroids at random. Then, it alternates between the following two steps:

1. **Update clusters.** Group the restaurants into clusters, where each cluster contains all restaurants that are closest to the same centroid. *In this step, centroid positions remain the same, but which cluster each restaurant belongs to can change.*
2. **Update centroids.** Compute a new centroid (average position) for each new cluster. *In this step, restaurant clusters remain the same, but the centroid positions can move.*

These steps are repeated to update the centroids until either an optimal list of centroids is found, or the centroid locations no longer change significantly each time (based on a maximum update threshold).

This visualization (http://tech.nitoyon.com/en/blog/2013/11/07/k-means/) is a good way to understand how the algorithm works.

Glossary

As you complete the remaining questions, you will encounter the following terminology. Be sure to refer back here if you're ever confused about what a question is asking.

- **location**: A pair containing latitude and longitude. Note that this is not a data abstraction, so we can assume its implementation is a two-element list.
- **centroid**: A location (see above) that represents the center of a cluster.
- **restaurant**: A restaurant data abstraction, as defined in `abstractions.py`.
- **cluster**: A list of restaurants grouped around a centroid.
- **user**: A user data abstraction, as defined in `abstractions.py`.
- **review**: A review data abstraction, as defined in `abstractions.py`.
- **feature function**: A single-argument function that takes a restaurant and returns a particular feature as a number, such as its mean rating or price.

Problem 3 (1 pt)

Implement `find_closest`, which takes a `location` and a sequence of `centroids` (locations). It returns the element of `centroids` closest to `location`.

You should use the `distance` function from `utils.py` to measure distance between locations. The `distance` function calculates the Euclidean distance (http://mathworld.wolfram.com/Distance.html) between two locations. It has been imported for you.

If two centroids are equally close, return the one that occurs first in the sequence of `centroids`.

> *Hint*: Use the `min` function to find the centroid with the minimum distance to `location`.
>
> `min` will automatically return the first element in a sequence if multiple elements have the same value.

Use Ok to unlock and test your code:

```
python3 ok -q 03 -u
python3 ok -q 03
```

Problem 4 (2 pt)

Now, implement a helper function for the first step in the loop of the k-means algorithm, `group_by_centroid`, which takes a sequence of `restaurants` and a sequence of `centroids` (locations) and returns a list of clusters. Each cluster of the result is a list of restaurants that are closer to a specific centroid in `centroids` than any other centroid. The order of the list of clusters returned does not matter.

If a restaurant is equally close to two centroids, it is associated with the centroid that appears first in the sequence of `centroids`.

> *Hint*: Use the provided `group_by_first` function to group together all values for the same key in a list of `[key, value]` pairs. You can look at the doctests to see how to use it.

Be sure not to violate abstraction barriers! Test your implementation before moving on:

```
python3 ok -q 04 -u
python3 ok -q 04
```

Problem 5 (1 pt)

Implement `find_centroid`, which finds the centroid of a `cluster` (a list of restaurants) based on the locations of the restaurants. The centroid latitude is computed by averaging the latitudes of the restaurant locations. The centroid longitude is computed by averaging the longitudes.

> *Hint*: Use the `mean` function from `utils.py` to compute the average value of a sequence of numbers.

Be sure not to violate abstraction barriers! Test your implementation before moving on:

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

Problem 6 (2 pt)

Finally, implement the full `k_means` algorithm. We've already filled out the first step of the algorithm, which was to randomly initialize a list of `centroids`. The rest of the algorithm consists of iteratively updating `centroids` by grouping the restaurants into clusters based on the current list of centroids and recomputing the new centroid for each cluster.

To complete the implementation, do the following in the body of the `while` statement:

1. Group `restaurants` into clusters, where each cluster contains all restaurants closest to a particular centroid in `centroids`.
2. Update `centroids` to contain the true centroid (i.e. average location) for each cluster.

> *Hint*: Use the `group_by_centroid` and `find_centroid` helper functions.

These two steps repeat until an update doesn't change the list of centroids *or* after `max_updates` iterations.

Use Ok to unlock and test your code:

```
python3 ok -q 06 -u
python3 ok -q 06
```

Your visualization can indicate which restaurants are close to each other (e.g. Southside restaurants, Northside restaurants). Dots that have the same color on your map belong to the same cluster of restaurants. You can get more fine-grained groupings by increasing the number of clusters with the `-k` option.

```
python3 recommend.py -k 2
python3 recommend.py -u likes_everything -k 3
```

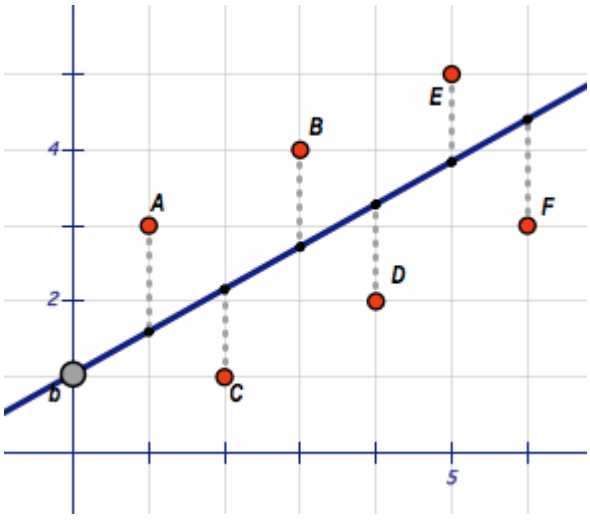
Congratulations! You've now implemented an unsupervised learning algorithm.

Phase 3: Supervised Learning

> All changes in this phase will be made to `recommend.py`.

In this phase, you will predict what rating a user would give for a restaurant. You will implement a *supervised* learning algorithm that attempts to generalize from examples for which the correct rating is known, which are all of the restaurants that the user has already rated. By analyzing a user's past ratings, we can then try to predict what rating the user might give to a new restaurant. When you complete this phase, your visualization will include all restaurants, not just the restaurants that were rated by a user.

To predict ratings, you will implement **simple least-squares (https://www.wikiwand.com/en/Least_squares) linear regression**, a widely used statistical method that approximates a relationship between some input feature (such as price) and an output value (the rating) with a line. The algorithm takes a sequence of input-output pairs and computes the slope and intercept of the line that minimizes the mean of the squared difference between the line and the outputs.



Problem 7 (3 pt)

First, let's use least-squares linear regression to write a function `find_predictor` that computes a predictor function for a user based on their existing restaurant ratings. A predictor function predicts a restaurant's rating based on a given feature, such as price or location. `find_predictor` takes in a `user`, a list of `restaurants` that have been reviewed by the user, and a feature function called `feature_fn` and returns two values: a `predictor` function and an `r_squared` value.

The `predictor` function is represented as the line `y = a + b * feature_fn(r)`, where `y` is the predicted rating for a restaurant given `r`, a restaurant. The `r_squared` value measures how accurately this line describes the original data.

To compute `a`, `b`, and `r_squared`, start by calculating the sums of squares `S_xx`, `S_yy`, and `S_xy` of the existing data (i.e. the feature values and corresponding user ratings for each restaurant in `restaurants`).

- $S_{xx} = \Sigma_i (x_i - \text{mean}(x))^2$
- $S_{yy} = \Sigma_i (y_i - \text{mean}(y))^2$
- $S_{xy} = \Sigma_i (x_i - \text{mean}(x)) (y_i - \text{mean}(y))$

After calculating the sums of squares, the regression coefficients (`a` and `b`) and `r_squared` are defined as follows:

- $b = S_{xy} / S_{xx}$
- $a = \text{mean}(y) - b * \text{mean}(x)$
- $R^2 = S_{xy}^2 / (S_{xx} S_{yy})$

> *Hint*: The `mean` and `zip` functions can be helpful here.

Use Ok to unlock and test your code:

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

Problem 8 (2 pt)

Now we need a way to decide which feature is the best predictor of restaurant ratings. This can differ from user to user; for example, some users may base ratings mostly on location, while others may base them more on price.

Implement `best_predictor`, which takes a `user`, a list of `restaurants`, and a sequence of `feature_fns`. It computes a predictor function for each feature function and returns the predictor that has the highest `r_squared` value. All predictors are learned from the subset of `restaurants` reviewed by the user (called `reviewed` in the starter implementation).

> *Hint*: The `max` function can also take a `key` argument, just like `min`.

Use Ok to unlock and test your code:

```
python3 ok -q 08 -u
python3 ok -q 08
```

Problem 9 (2 pt)

Now that we are able to find an optimal predictor function for a given user, we can compile a full collection of restaurant ratings for the user, including ratings for restaurants they haven't actually rated yet!

Implement `rate_all`, which takes a `user`, a list of `restaurants`, and a sequence of `feature_fns`. It returns a dictionary where the keys are the names of each restaurant in `restaurants` and the values are the corresponding ratings (numbers).

If a restaurant has already been rated by the user, `rate_all` will assign the restaurant the user's rating. Otherwise, `rate_all` will assign the restaurant the rating computed by the best predictor for the user. The best predictor is chosen using a list of `feature_fns`.

> *Hint*: `user_rating`, implemented in `abstractions.py`, returns a user's rating for a restaurant given the `user` and `restaurant_name`.

Be sure not to violate abstraction barriers! Test your implementation before moving on:

```
python3 ok -q 09 -u
python3 ok -q 09
```

In your visualization, you can now predict what rating a user would give a restaurant, even if they haven't rated the restaurant before. To do this, add the `-p` option:

```
python3 recommend.py -u likes_southside -k 5 -p
```

If you hover over each dot (a restaurant) in the visualization, you'll see a rating in parentheses next to the restaurant name.

Problem 10 (1 pt)

As a final addition to our visualization, let's add the ability to focus the visualization on a particular restaurant category by implemening `search`. The `search` function takes a category `query` and a sequence of restaurants. It returns all restaurants that have `query` as a category.

Be sure not to violate abstraction barriers! Test your implementation:

```
python3 ok -q 10 -u
python3 ok -q 10
```

Congratulations, you've completed the project! The `-q` option allows you to filter based on a category. For example, the following command visualizes all sandwich restaurants and their predicted ratings for the user who `likes_expensive` restaurants:

```
python3 recommend.py -u likes_expensive -k 2 -p -q Sandwiches
```

Predicting your own ratings

Once you're done, you can use your project to predict your own ratings too! Here's how:

1. In the `users` directory, you'll see a couple of `.dat` files. Copy one of them and rename the new file to `yourname.dat` (for example, `john.dat`).
2. In the new file (e.g. `john.dat`), you'll see something like the following:

   ```
   make_user(
       'John DoeNero',     # name
       [                   # reviews
           make_review('Jasmine Thai', 4.0),
           ...
       ]
   ```

   Replace the second line with your name (as a string).
3. Replace the existing reviews with reviews of your own! You can get a list of Berkeley restaurants with the following command:

   ```
   python3 recommend.py -r
   ```

   Rate a couple of your favorite (or least favorite) restaurants.
4. Use `recommend.py` to predict ratings for you:

   ```
   python3 recommend.py -u john -k 2 -p -q Sandwiches
   ```

   (Replace `john` with your name.) Play around with the number of clusters (the `-k` option) and try different queries (with the `-q` option)!

How accurate is your predictor?

Conclusion

Submit to Ok to complete the project.

```
python3 ok --submit
```

You can also check your score on the project (not including composition) using

```
python3 ok --score
```

Weekly Schedule (/weekly.html)

Office Hours (/office-hours.html)

Staff (/staff.html)

Resources (/resources.html)

Studying Guide (/articles/studying.html)

Debugging Guide (/articles/debugging.html)

Composition Guide (/articles/composition.html)

Policies (/articles/about.html)

Assignments (/articles/about.html#assignments)

Exams (/articles/about.html#exams)

Grading (/articles/about.html#grading)