Jerry Cain
CS106AX

Handout #26
November 21st, 2025

CS106AX Practice Final Examination

Review session: Sunday, December 7th, 12:00 – 2:00 P.M., CoDa B90

Scheduled final: Monday, December 8th, 8:30 – 11:30 A.M., CoDa B90

This handout is intended to give you practice solving problems that are comparable in format and difficulty to those which will appear on the final exam.

Time of the exam

The final exam is scheduled for Monday, December 8th, 8:30–11:30 A.M. in CoDa B90, which is the room where we normally hold lecture. If you are unable to take the final exam at the scheduled time, or if you need special accommodations, please send an email message to **`jerry@cs.stanford.edu`** stating the following:

- The reason you cannot take the exam at the scheduled time.
- A list of three-hour blocks (or longer if you have OAE accommodations) on Monday or Tuesday of final exam week at which you could take the exam. These time blocks must be during the regular working day and must therefore start between 8:30 and 2:00.

To arrange special accommodations, Jerry must receive your message by 5:00 P.M. on Friday, December 5th. Replies will be sent by email on Saturday, December 6th.

Review session

The course staff will conduct a review session on Sunday, December 7th, from 12:00–2:00 P.M., in CoDa B90.

Coverage

The exam will focus on the Python and client-side JavaScript material we've learned since Week 5. In particular, all of the questions will require you write code in Python, save for the last one, which will require you code in JavaScript to exercise your client-side web programming skills. As with the midterm, the exam is open notes and open book, though it's closed computer.

The questions presented below are examples of what you might see on your own final exam.

Problem 1: Python Strings (20 points)

Implement a function called `lrnoss`—that's short for **l**ongest **r**epeating, **n**on**o**verlapping **s**ub**s**tring—which accepts a string and returns the longest substring that appears at least twice, with the added constraint that at least two copies of the substring are nonoverlapping. If there are multiple such substrings, then `lrnoss` can return any single one of them and ignore the others. And if there are no repeating substrings at all, your function should just return the empty string.

For instance, the longest repeating, nonoverlapping substring within "**aabaabaaba**" is `"aaba"` (and I've underlined the two copies of `"aaba"` as proof they're nonoverlapping). It's true that `"abaa"`, `"baab"`, and even longer substrings like `"aabaa"` and `"aabaab"` appear multiple times as well, but in all cases the first copy overlaps with the second. However, there are two copies of `"aaba"` that don't overlap, so that's the longest repeating, nonoverlapping substring!

More examples:

`lrnoss('abcabcabc')` *might return* `'abc'`
`lrnoss('aabaabaaba')` *returns* `'aaba'`
`lrnoss('banana')` *might return* `'an'`
`lrnoss('windowwasher')` *returns* `'w'`
`lrnoss('abcdefgh')` *returns* `''`

The first and third examples say *might return*, because there were other possible return values. But the function is only required to return one of them, not all of them.

Present your implementation on the next page.

(space for your answer to problem #1 appears on the next page)

```
def lrnoss(str):
    """
    Accepts an arbitrary string and returns the longest substring
    that appears at least twice, where at least two copies of the
    substring are nonoverlapping.
    """
```

Problem 2: Python Lists (20 points)

The ***Playfair cipher*** relies on a 5x5 table of letters—all letters of the alphabet minus the Q, which is omitted because it's so rare—to guide the encryption of a cleartext message. The table is constructed from a ***passphrase*** by filling in the 25 slots from left to right, top to bottom, with the letters of the passphrase in the order they first appear and then filling any remaining spots with the rest of the letters of the alphabet.

For instance, if the passphrase is `'OBSERVEOBLONGCHEWYINERT'`, the table would look like this:

```
O B S E R
V L N G C
H W Y I T
A D F J K
M P U X Z
```

For this problem, you'll implement just the `construct_playfair_table` function, which accepts the passphrase and constructs the relevant table as described above. The table is really list of length 5, where each entry itself is a list of 5, one-character strings. You may assume the passphrase is composed of uppercase letters minus the `'Q'`. So, a call to `construct_playfair_table('OBSERVEOBLONGCHEWYINERT')` might go like this:

```
>>> print(construct_playfair_table('OBSERVEOBLONGCHEWYINERT'))
  [['O', 'B', 'S', 'E', 'R'],
   ['V', 'L', 'N', 'G', 'C'],
   ['H', 'W', 'Y', 'I', 'T'],
   ['A', 'D', 'F', 'J', 'K'],
   ['M', 'P', 'U', 'X', 'Z']]
>>>
```

(space for your answer to problem #2 appears on the next page)

```
def construct_playfair_table(passphrase):
  '''
  Constructs a Playfair table as described in the problem statement.
  '''
  ALPHABET = 'ABCDEFGHIJKLMNOPRSTUVWXYZ' # note Q is omitted
```

Problem 3: Working with Python Dictionaries (20 points)

CS106A and CS106AX students are asked to use PyCharm to manage all Python files contributing to any one assignment. Interestingly enough, large portions of PyCharm are themselves implemented in Python—in particular, PyCharm maintains persistent data structures about who edited which file last and when they did so. For instance, PyCharm maintains the following information on behalf of the team of people making changes and introducing bugs!

```
project = {
    "name": "Flutterer",
    "files": {
        "api.py": {
            "size": 2312, # in bytes
            "commits": [{
                    "author": "Eugene",
                    "timestamp": "2023-12-04T14:45:30",
                    "bugs": 13
                }, {
                    "author": "Natalia",
                    "timestamp": "2023-12-03T12:10:13",
                    "bugs": 0
                }, {
                    "author": "Natalia",
                    "timestamp": "2023-12-03T11:00:01",
                    "bugs": 1
                }]
        },
        "floot.py": {
            "size": 5421,
            "commits": [{
                    "author": "Reilly",
                    "timestamp": "2023-12-05T19:44:51",
                    "bugs": 2
                }, {
                    "author": "Eugene",
                    "timestamp": "2023-12-03T10:20:15",
                    "bugs": 9
                }]
        },
        "floot_comment.py": {
            # similarly structured
        }
    }
}
```

Each of the `commits` fields you see are themselves lists of dictionaries, where each dictionary details the number of bugs the specified user introduced at the time they saved

the file. In principle, any single commits list can be empty or arbitrarily large, and each developer (e.g., Eugene, Natalia, Reilly, or anyone else) can appear multiple times within a single commits list (as Natalia clearly does) and across multiple commits lists (as Eugene clearly does).

Your job here is to implement the **`shame`** function, which accepts a project data structure much like that above and returns the name of the author who's contributed the largest number of total bugs across all commits to the project. ☺ Your implementation, of course, should work for any data structure like that above. If two or more people contribute the same maximum number of bugs, you can return any single one of them.

Use the space below and on the next page to implement your **`shame`** function. Your implementation can assume the supplied project is structured as the one described above.

```
def shame(project):
  '''
  Crawls over the entire project and returns the name of the author
  who has introduced the most bugs across all commits to all files.
  '''
```

(more space for your answer to problem #3 appears on the next page)

(more space for your answer to problem #3 appears below)

Problem 4: Defining Python Classes and Reading Files (20 points)

In 2019, the College Board announced plans to share **landscape** metrics alongside SAT scores with admission panels. These metrics were designed to be used by admissions officers to decide whether the various social and economic hardships captured by the metrics might inform their interpretation of the applicant's performance. The metrics were computed on a per-school basis, and the College Board included those metrics alongside the scores.

Imagine you are building a platform whose sole purpose is to search through these metrics. You have been granted access to the data in a single file, formatted like so:

```
Alabama
Auburn,Auburn High School,1250,43
Birmingham,Oak Mountain High School,1230,77
Birmingham,Spain Park High School,1270,64
...                              ⇐ many schools omitted
Vestavia Hills,Vestavia Hills High School,1370,64
       ⇐ blank line marks end of Alabama's data
Alaska
Anchorage,Diamond High School,1080,68
Anchorage,East High School,1020,50
Anchorage,West High School,1100,50
...                              ⇐ many schools omitted
Wasilla,Wasilla High School,1010,23
       ⇐ blank line marks end of Alaska's data
...                              ⇐ many more states
       ⇐ blank line marks end of West Virginia's data
Wisconsin
Beaver Dam,Wayland Academy,1170,9
Brookfield,Brookfield Central High School,1420,90
...                              ⇐ many schools omitted
Milwaukee,University School Milwaukee,1390,86
       ⇐ blank line marks end of Wisconsin's data
```

Each state for which data was available is included in this file, and the data available for each state (including the last one in the file) is terminated by a blank line. The first line for each state is the name of the state itself, and all other lines for that state provide details for a school where 30 or more students took the SAT. Each line beyond the first is a comma-separated list of four items: city, school name, median SAT score, and the integer percentage of students who enroll in either a two- or four-year college or university upon graduation.

For this problem, you're to provide the implementation of a Python class called `SATMetrics` whose constructor opens a file, assumes it's structured as above, and parses the entire file to build an internal representation of the information that allows it to respond to two additional methods reasonably quickly. Those methods are:

- `getCityStateData`, which accepts a city and state and returns a list of dictionaries, where each dictionary aggregates information about a high school

across three properties: name, sat, and college. So, as a call to **`getCityStateData("Anchorage", "Alaska")`** would return:

```
[
    {"name": "Diamond High School", "sat": 1080, "percent": 68},
    {"name": "East High School", "sat": 1020, "percent": 50},
    {"name": "West High School", "sat": 1100, "percent": 50},
]
```

If the city or state isn't present, your method should simply return an empty list.

- **`getHighSchoolsMeetingThreshold`**, which accepts an SAT score as argument and returns the names of the high schools whose median SAT score matches or exceeds the number supplied. Your return value should just include the names of high schools and shouldn't be worried if two high schools happen to share the same name. You can assume all median scores are multiples of 10 and range from 200 to 1600 inclusive. So, a call to **`getHighSchoolsMeetingThreshold(1220)`** would return a list structured as:

```
["Auburn High School", "Spain Park High School", # plus others
```

If no high schools meet the supplied threshold, your method should simply return the empty list.

Using the next two pages, present your implementation of the **`SATMetrics`** constructor and the two methods as described above. The constructor should do all the file processing and most of the work needed to ensure the two method implementations at most a few (e.g., perhaps 1 – 6) lines long. You shouldn't use the **`TokenScanner`** class, but you should rely on the Python string's **`split`** method, which works like this:

`'sentence without punctuation'.split(' ')` *returns* **`['sentence', 'without', 'punctuation']`**

`'172.43.100.128'.split('.')` *returns* **`['172', '43', '100', '128']`**

`'lions, tigers, bears'.split(',')` *returns* **`['lions', ' tigers', ' bears']`**

(space for your answer to problem #4 appears on the next page)

```
class SATMetrics:
   """
   Exports a class that allows one to profile various high schools
   for their median SAT scores and other metrics.

   Code for the constructor should be included on this page, and code
   for the two methods should be presented on the next page.  You may
   not always need the full amount of space that's been provided.
   """

   def __init__(self, filename):
      """
      Parses the contents of the named file and compiles
      all of the internal data structures needed to ensure
      the two other methods can run reasonably fast.
      """
```

(even more space for your answer to problem #4 appears on the next page)

```
def getCityStateData(self, city, state):
   """
   Returns a list of dictionaries, where each dictionary
   contains information about a high school within the supplied
   city and state.
   """
```

```
def getHighSchoolsMeetingThreshold(self, threshold):
   """
   Returns a list of all high schools with median SAT score at
   or above the supplied threshold.
   """
```

Problem 5: Client-Side JavaScript (20 points)

Many major airlines provide travel service between most airports within the U.S., Canada, and Mexico. When booking air travel back and forth between, say, Chicago and Mexico City, you'll typically visit the websites of many airlines—perhaps Aeroméxico, Delta, United, Viva Aerobus, Volaris—to compare prices and generally book the least expensive one. When flying between Toronto and Seattle, Aeroméxico, Viva Aerobus, and Volaris won't provide any options, but Delta, United, and Air Canada do. To simplify your travel planning needs, you've decided to write client-side JavaScript using our **`AsyncRequest`** class to invoke API endpoints provided by all major airlines to see which ones offer service between any two airports of your choosing.

Miraculously, all the airlines provide APIs with the same URL structure—beyond the domain name (e.g., **`www.delta.com`**, **`www.united.com`**, **`www.aircanada.com`**, **`www.aeromexico.com`**, etc.) the API endpoint is simply **`"/search"`**, where **`GET`** parameters **`depart`** and **`arrive`** are used to specify the airport codes (e.g., SFO, JFK, ORD, LAX, etc.) of the departure and destination cities. The response payload from each API endpoint—no matter the domain name—is always JSON including two or three fields, depending on whether the airline provides travel options between the two airports. The JSON payload contains an **`airline`** property specifying the full name of the airline, a **`found`** property reporting **`true`** if and only the airline flies between the two airports, and a **`price`** attribute supplying the best available price in US dollars (though the **`price`** property will be missing if **`found`** is reported as **`false`**).

So:

```
http://www.united.com/search?depart=SFO&arrive=YYZ
```

might respond with a payload of:

```
{
   airline: "United Airlines",
   found: true,
   price: 801.50
}
```

whereas:

```
http://www.aeromexico.com/search?depart=SFO&arrive=YYZ
```

might respond with a payload of:

```
{
   airline: "Aeroméxico",
   found: false,
}
```

You're to implement a function called **`options`** that accepts departure and arrival airport codes and eventually prints out a list of all airline names, one per line, alongside the best price that airline offers. If an airline doesn't offer service between the two airports, you should simply omit it from the listing. Once all requests have been processed and all responses have been received, you should print out the best price. You should issue all requests to all airline API servers so they are all processed simultaneously, and you may

print out each line of the list, except for the last one, in any order. So, a call to **`options("SFO", "YYZ")`** might ultimately print out the following:

```
Delta Airlines: $578.70
Alaska Airlines: $1008.00
United Airlines: $801.50
Air Canada: $477.00
American Airlines: $560.00
Air Canada offers the best deal at $477.00
```

To be clear, **`options`** queries all airlines, including those not listed presumably because those airlines responded saying they didn't provide service between the two airports. The five airlines that were listed might have been listed in any order, but the final line can only be listed after your implementation detects the last response has come in.

Present your implementation of **`options`** on the next page. Your success handler will need to be defined as an inner function—within **`options`**—so it can access any top-level variables you define. Think very carefully about how you can keep track of the best price and the number of responses you've received so you know when all responses have come in. You may assume at least one airline provides service between the two cities and that all API calls succeed.

(more space for your answer to problem #5 appears on the next page)

```
function options(depart, arrive) {
  let domains = ["www.aa.com", "www.aeromexico.com", "www.aircanada.com",
                 "www.alaskaair.com", "www.avianca.com", "delta.com",
                 … many other airlines
                 "www.united.com", "www.westjet.com"];
  let numDomains = domains.length;
  // present any additional variable needed and the rest of your
  // implementation, including the inner function serving as your
  // success handler.
```