Jerry Cain
CS 106AX

Handout #24
November 21st, 2025

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 **`lrs`** (short for **l**ongest **r**epeated **s**ubstring), which searches a string for all of its substrings and returns the longest substring appearing two or more times. If there are two or more such substrings, then your implementation can return any one of them. If there are no repeated substrings, then your **`lrs`** function should return the empty string.

A properly implemented **`lrs`** will produce the following results:

- **`lrs("hello")`** returns **`"l"`**
- **`lrs("fauna")`** returns **`"a"`**
- **`lrs("banana")`** returns **`"ana"`**
- **`lrs("abracadabra")`** returns **`"abra"`**
- **`lrs("rationalizations")`** returns **`"ation"`**
- **`lrs("whippersnappers")`** returns **`"ppers"`**
- **`lrs("subdermatoglyphic")`** returns **`""`**

Note that each of the two **`"ana"`**s within **`"banana"`** overlap, and that's okay.
And **`"subdermatoglyphic"`** happens to be the longest word in the English language that doesn't use the same letter twice—hence, the empty string return value.

Present your implementation of **`lrs`** below.

```
def lrs(s):
   """
   Analyzes the incoming string called s and returns
   the longest substring that appears two or more times.
   """
```

Problem 2—Python Strings and Lists (20 points)

Given a list of words, write a function called **`group_anagrams`**, which returns a list of lists, where each inner list contains all of the words in the original that are anagrams of one another. For instance,

```
group_anagrams(["eat", "tea", "inch",
                "ate", "chin", "bat"])
```

might return

```
[["eat", "tea", "ate"], ["inch", "chin"], ["bat"]]
```

The order of the words within each group and the order of the groups themselves doesn't matter, so there are several different orderings of the above that are also acceptable return values as well.

Another example? Here's a call to

```
group_anagrams(["dusty", "players", "bling", "parsley"
           "study", "glean", "arc", "angel"])
```

which might return

```
[["dusty", "study"], ["players", "parsley"],
  ["bling"], ["glean", "angel"], ["arc"]]
```

Note that **`group_anagrams([])`** returns **`[]`**.

You can rely on the following **`sort_string`** function to return a copy of the supplied string, except that its characters have been reordered so they're all sorted a to z.

```
def sort_string(s):
  """
  Examples:
     sort_string("bookkeeper") returns "beeekkoopr"
     sort_string("players") returns "aelprsy"
     sort_string("players") returns "aelprsy"
     sort_string("begins") returns "begins"
  """
  return "".join(sorted(s))
```

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

```
def group_anagrams(strings):
  """
  Returns a list of anagram groups as described on the
  previous page.
  """
```

Problem 3—Working with Python Dictionaries and Objects (20 points)

At this point in the quarter, the data structures you know fairly well are the ones from the Adventure assignment, which drive the operation of the **`play`** method in **`AdvGame`**. You could, of course, do something else with those same structures. For example, you might want to write a function that would print a cheat sheet for solving a particular Adventure game by listing all the objects that are available in the game along with where they are located at the beginning of the game and where they are required to traverse a locked passage.

Your job in this problem is to implement the function

```
def printCheatSheet(objects, rooms)
```

that takes the data structures for the objects and the rooms and displays a cheat sheet showing the name of each object, its short description in parentheses, and the short description of its initial location (unless it's **`"PLAYER"`**, in which case you should just print **`"PLAYER"`**). After each object in the list, the **`printCheatSheet`** function should go through the rooms data structure and print out a line for each entry in which that object acts as a key to a locked passage. Refer to Handout 38 for the list of methods available to **`AdvRoom`** and **`AdvObject`**. Recall that passages are structured are tuples: (**`verb`**, **`destination`**, **`key`**).

For example, if you call the function

```
def AdventureCheatSheet():
   rooms = AdvGame.readRooms("SmallRooms.txt")
   objects = AdvGame.readObjects("SmallObjects.txt")
   printCheatSheet(objects, rooms)
```

your implementation of **`printCheatSheet`** might generate the following output:

```
KEYS (a set of keys) starts: Inside building
  Needed for DOWN from Outside grate
LAMP (a brightly shining brass lamp) starts: Beneath grate
  Needed for XYZZY from Inside building
  Needed for WEST from Cobble crawl
ROD (a black rod with a rusty star) starts: Debris room
WATER (a bottle of water) starts: PLAYER
```

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

One of the political events of interest in the next week is the British general election scheduled for December 12th. In each parliamentary district, which are called constituencies in Britain, several candidates from different parties will be running, although not all parties will compete in every election. Each candidate will receive some number of votes in the election. Under Britain's "first past the post" system, the candidate who wins the greatest number of votes wins the seat for that constituency, even if that candidate does not win a majority of the votes cast.

Imagine that you have been hired by a consulting firm that seeks to analyze the results of the election, which have been delivered in a large file that looks like this, which shows the results of the 2015 general election:

BritishElectionData.txt

```
Aberavon
Stephen Kinnock (Labour) 15416
Peter Bush (UKIP) 4971
Edward Yi He (Conservative) 3742

Aberconwy
Guto Bebb (Conservative) 12513
Mary Wimbury (Labour) 8514
Andrew Haigh (UKIP) 3467

Aberdeen North
Kirsty Blackman (SNP) 24793
Richard Baker (Labour) 11397
Sanjoy Sen (Conservative) 5304
Euan Davidson (LibDem) 2050
  .
  .
  .
Brighton Pavilion
Caroline Lucas (Green) 22871
Purna Sen (Labour) 14904
Clarence Mitchell (Conservative) 12448
  .
  .
  .
York Outer
Julian Sturdy (Conservative) 26477
Joe Riches (Labour) 13348
James Blanchard (LibDem) 6269
Paul Abbott (UKIP) 5251
```

Each entry in the file consists of the name of the constituency on the first line, followed by as many lines as there were candidates in that constituency. The end of each candidate list, including the last one) is marked with a blank line. Each candidate line consists of the candidate name, the party name in parentheses, and the number of votes cast for that candidate. You may assume that the file exists and is properly formatted, which means that you don't have to include any error checking.

Implement a Python class called **`ElectionData`** whose constructor looks like this:

```
class ElectionData:
   def __init__(self, filename):
```

The **`ElectionData`** constructor should read the contents of the named file and transform its contents into a suitable internal form. Your implementation of the **`ElectionData`** class, beyond its constructor, should include the following methods:

- A method **`getConstituencyNames`** that returns a Python list containing the constituency names that appear in the file.
- A method **`getResults(name)`** that returns the results of the election for the constituency with the specified name. The result is another list, each of whose elements is a dictionary with fields **`candidate`**, **`party`**, and **`votes`**. For example, calling **`getResults("Aberavon")`** should return the following list of dictionaries (or the empty list if the constituency isn't named in the file):

```
[
   { "candidate": "Stephen Kinnock", "party": "Labour", "votes": 15416 },
   { "candidate": "Peter Bush", "party": "UKIP", "votes": 4971 },
   { "candidate": "Edward Yi He", "party": "Conservative", "votes": 3742 }
]
```

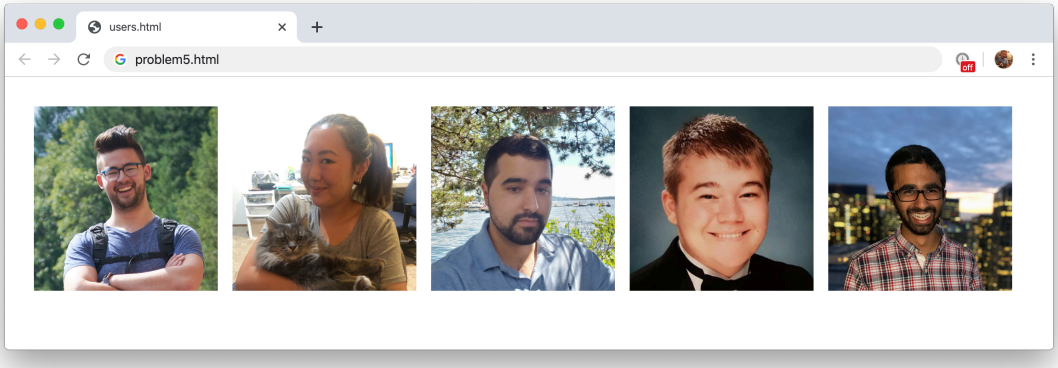
In this problem, all you have to do is read the data into the internal structure. Any actual analyses of results are the responsibility of the clients of your **`ElectionData`** class. For example, someone might use your **`ElectionData`** class to compute the total number of votes for the Labour party, as with:

```
def TestElectionData():
   count = 0
   electionData = ElectionData("BritishElectionData.txt")
   constituencies = electionData.getConstituencyNames()
   for i in range(len(constituencies)):
      results = electionData.getResults(constituencies[i])
      for j in range(len(results)):
         if results[j]["party"] == "Labour":
            count += results[j]["votes"]
   print("Total Labour vote -> {}".format(count))
```

Present your full implementation of the **`ElectionData`** class, which includes the constructor and the two methods described above. The vast majority of your code should reside within the constructor, and the implementations of the two additional methods will be very, very short.

Problem 5—Client-Side JavaScript (20 points)

Flutterer has gained some traction, so you've decided to ride some Assignment 8 momentum, raise a couple million dollars, and build a new HTML component that presents the images of its five users in a clean little row, just like this:



Assume the Flutterer API has been extended to support one new endpoint, which can be accessed via **`GET /api/images`**. This endpoint, when properly invoked, responds with a payload JSON string that's structured as follows:

```
"[
   {
      'name': 'Ryan Eberhardt',
      'url': '/images/ryan.jpg'
   },
   {
      'name': 'Suzanne Joh',
      'url': '/images/suzanne.jpg'
   },
   {
      'name': 'Esteban Rey',
      'url': '/images/esteban.jpg'
   },
   {
      'name': 'Jonathan Kula',
      'url': '/images/jonathan.jpg'
   },
   {
      'name': 'Anand Shankar',
      'url': '/images/anand.jpg'
   }
]"
```

The HTML component is programmatically constructed by initiating a **`GET`** request to **`/api/images`** after installing a success handler that knows how to process the response payload—which can vary as new images are uploaded and new image URLs are generated—and populate an initially empty **`div`** with a id of "**`user-images`**", as with:

```
<body>
   <div id="user-images"></div>
</body>
```

For each **`img`** tag, you should ensure the **`src`** attribute is set equal to an image URL, the **`alt`** attribute is set equal to the person's name, and you should ensure the **`class`** attribute is set to "**`thumbnail`**". Assume the **`.thumbnail`** CSS rule properly sizes the images and ensures they're are laid down side by side.

Present your JavaScript implementation of **`fetchAndLoadImages`**, which codes to the requirements specified above. You'll also need to implement a callback function as part of your answer. Assume **`fetchAndLoadImages`** is implemented as a top-level function, which means it doesn't have access to any **`let`** variables defined at higher scopes. Note that neither **`fetchAndLoadImages`** nor the callback you're implementing return anything.

```
function fetchAndLoadImages() {
```