Unit 4 Review

15-110 – Wednesday 04/28

Announcements

- **Check6-1** revision deadline was today
- **Check6-2** deadline is **Friday at noon**

Agenda

- Unit 4 Overview
- Reading and Reformatting Data
- Matplotlib Coding
- Simulation Coding

Unit 4 Overview

Unit 4 Goals

Our fourth unit explored how computer science could be used as a **tool** in other domains. We discussed this in two stages: how to **model** domain-specific data, and how to **answer questions** about that data.

How did the topics we discussed fit into these themes?

Modeling Data

In data analysis, we discussed how you can **read and write files** and **interpret data with protocols** to load real data into a program. We also learned how to **reformat** data to meet the task's needs.

In simulation, we learned about the **model-view-controller** framework, where we store data in a shared structure, represent it graphically with a view, and update it **over time** or based on **events**.

In machine learning, we talked about how to choose different **categories of learning and reasoning** based on the **features** being analyzed. We also used **game decision trees** to model data for AI agents.

Answering Questions

In data analysis, we discussed a few common **analysis methods**, and used **Matplotlib** to visualize data as charts.

In simulation, we used **Monte Carlo methods** to run **experiments** over simulations and find the expected results.

In machine learning, we discussed how we use data to **train, validate, and test** a reasoning model, and how an AI can **perceive, reason, and act** to accomplish a goal. We also used the **Minimax** algorithm and **heuristics** to help an AI find a good next action quickly.

Upcoming Topics

Our final unit will address how computer science affects the world by diving into **history**, exploring questions regarding **ethics** in the present day, and looking forward at the **future**.

We'll wrap up the course with a single lecture exploring different opportunities in the **School of Computer Science**.

Reading and Reformatting Data

Opening Files

To read a file from your computer into a string, you first need to call the `open()` built-in function with the **filepath** of the file.

If the file is in the same directory as the Python file, the filepath is just its name. If it's in a folder, use `"folder/name"` instead, with `"/"` separating the folder's name from the file's name.

This can be repeated for nested folders - for example, `"project/data/all-icecream.csv"`.

Reading Files

Once you've opened a file properly, you'll have a **file object** stored in a variable. The simplest way to get data from that file object is to call the `read()` method on it:

```
text = f.read()
```

This reads *all* the text from the file into a string and stores that string in the variable. You can then parse the variable to separate lines or parts of lines as needed.

Rearrange Data to Solve Problems

We often need to rearrange the data that is loaded in from a file in order to answer specific questions about it.

Often this involves identifying patterns that locate the part of the data we care about, then ignoring the rest.

Slicing, `str.find()`**,** `str.split()`**,** and `str.strip()` are useful tools for reformatting data.

Example: Parsing Website Data

We want to generate a list of all the lectures in 15-110 by date and topic based on the text on the Schedule page. Copying that information by hand is tedious; instead, let's read and reformat the data!

Copy all the text from https://www.cs.cmu.edu/~110/schedule.html into a .txt file in the same directory as the code file.

Inspect the data

How can we parse out the date and lecture topic from the data? We need to look for **groupings** of data and **patterns** that always occur around that information.

WEEK	DATES	DUE DATES	TOPICS	MATERIALS	RECORDINGS	PRACTICE	EXERCISE
UNIT: Programming Skills and Computer Organization							
1	02/01 Mon		Lecture: Course Intro & Algorithms and Abstraction		Mon slides	Mon recording	Mon practice Ex1-1
02/02 Tue							
02/03 Wed		Lecture: Programming Basics		Wed slides - code		Wed recording	Wed practice Ex1-2
02/04 Thu		Recitation	Rec problems				
02/05 Fri		Lecture: Data Representation		Fri slides	Fri recording	Fri practice	Ex1-3
2	02/08 Mon	Check1	Lecture: Functions		Mon slides - code	Mon recording	Mon practice Ex2-1
02/09 Tue							
02/10 Wed		Lecture: Functions II & How Python Works			Wed slides - code		
Bonus Graphics slides - graphics starter code				Wed recording		Wed practice Ex2-2	
(Optional)							
02/11 Thu		Recitation	Rec problems - code - extra problems				
02/12 Fri		Lecture: Booleans and Conditionals		Fri slides - code		Fri recording	Fri practice Ex2-3
…							

Inspect the Data

How is the data grouped? Each date and lecture are paired on the same **line**. We should start by splitting the data across lines. (Not all lines have a date and lecture, though).

How to lines separate points of information? Look closely and you'll see that each column is separated by **tabs**. Split each line by tabs to get the items.

Load the data

```
f = open("schedule.txt", "r")
text = f.read()
f.close()

lines = text.split("\n")
data = []
for line in lines:
    data.append(line.split("\t"))
```

Find the Relevant Information

Each line is now a list of tokens. We want to find the token that holds the **date** and the token that holds the **lecture title**.

How is the date different from the rest of the data? It always takes the format **XX/XX DAY**. Check whether characters 0-1 and 3-4 of the string are numbers.

Find the Date

```
for line in data:
    date = None
    for token in line:
        if token[:2].isdigit() and token[3:5].isdigit():
            date = token[:token.find(" ")] # remove day
```

Find the Relevant Information Pt. 2

How can we find the lecture title?

Each lecture title starts with **Lecture:** . Check whether the token starts with that text.

Slice off the "Lecture: " before storing the information to make the data cleaner.

Find the Lecture Title

```
for line in data:
    ...
    lecture = None
    for token in line:
        ...
        elif token[:8] == "Lecture:":
            lecture = token[9:]
```

Store the results

Now we just have to combine the results in a string and store them in a new list.

We can finally print out a nice, cleaned list of lectures with their dates. Nice!

Put it Together

```
lectures = []
for line in data:
    ...
    if date != None and lecture != None:
        lectures.append(date + " " + lecture)

for item in lectures:
    print(item)
```

Every Problem is Different

Every data analysis problem will require a different approach to reformat the data.

What's most important is that you look for the **patterns** in the data, and identify **algorithmic techniques** that will let you recognize those patterns.

Matplotlib Coding

Coding with Matplotlib

Writing code with Matplotlib isn't like writing code to solve a homework problem.

Instead of solving everything from scratch, you may need to **refer to examples** to determine how to add certain features to your charts.

Why is this different? The Matplotlib library is **huge**. It isn't efficient to memorize every possible function- it's better to look up functions when you need them.

Example: Making a Scatter Plot

Goal: we want to visually compare the most popular ice cream flavors in our dataset to determine if there are any interesting trends in which flavors were chosen most often as the 1st vs 2nd favorite flavor.

Start with the code from the Data Analysis II lecture that creates a dictionary from the ice cream flavors. Since we want to investigate the correspondence between 1st and 2nd favorites, we'll update the code to only read data from one column.

Updated getIceCreamCounts

```
def getIceCreamCounts(data, colName):
    iceCreamDict = { }
    col = data[0].index(colName)
    for i in range(1, len(data)): # skip header
        flavor = data[i][col]
        if flavor not in iceCreamDict:
            iceCreamDict[flavor] = 0 # start new count
        iceCreamDict[flavor] += 1
    return iceCreamDict

import csv
f = open("all-icecream.csv", "r")
data = list(csv.reader(f))
f.close()

firstCounts = getIceCreamCounts(data, "#1 cleaned")
secondCounts = getIceCreamCounts(data, "#2 cleaned")
```

Reformat Data

For each flavor, we want to compare the number of #1 preferences for that flavor to the number of #2 preferences for that flavor.

First, we need to write some code to restructure the data from a dictionary to three lists- the flavors, #1 counts, and #2 counts. Let's also narrow down our data to only include flavors that appear at least 10 times.

We need to account for three possibilities – a flavor might occur in *both* dictionaries, in *just* the first dictionary, or in *just* the second dictionary.

Reformatting Data

```
flavors = []
firstPrefs = []
secondPrefs = []

for flavor in firstCounts:
    first = firstCounts[flavor]
    if flavor in secondCounts:
        second = secondCounts[flavor]
    else:
        second = 0

    if first + second >= 10:
        flavors.append(flavor)
        firstPrefs.append(first)
        secondPrefs.append(second)
...
```

```
...

for flavor in secondCounts:
    if flavor not in firstCounts:
        second = secondCounts[flavor]
        if second >= 10:
            flavors.append(flavor)
            firstPrefs.append(0)
            secondPrefs.append(second)
```

Scatter Plot Demo

Each of these data types is **numerical,** and there are **two** dimensions, so we need a **scatterplot**. Search the `plt` documentation to find the right method.

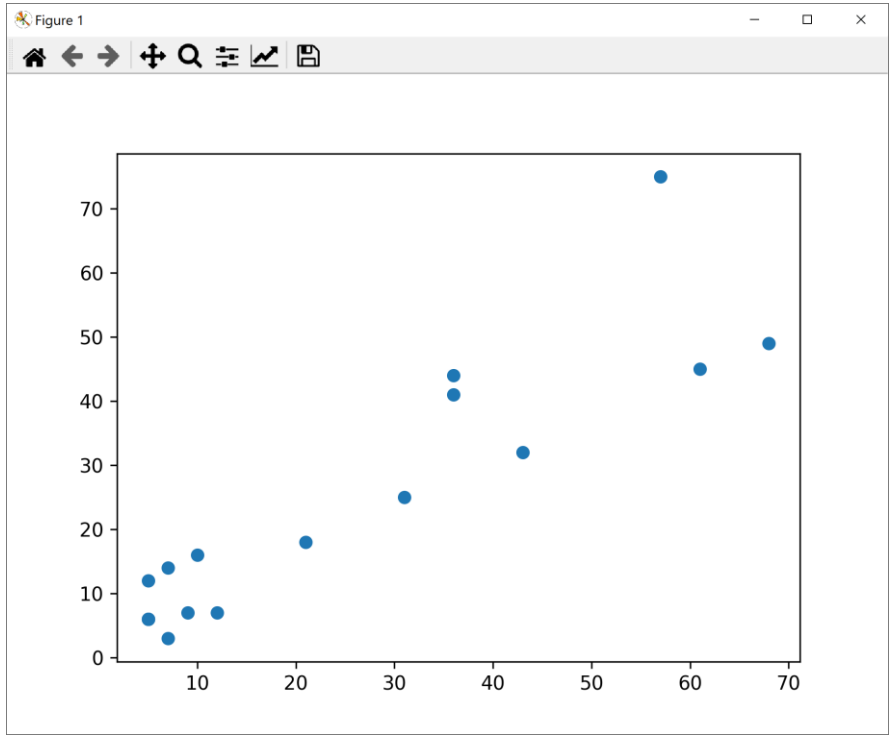
This shows that the function we need is `plt.scatter()`. We can read about its arguments here:

https://matplotlib.org/stable/api/_as_gen/matplotlib.pyplot.scatter.html

Setting up a Basic Chart

The core `plt.scatter()` function requires two lists - one of x values, one of y values. Pass in `firstPrefs` and `secondPrefs`.

```
import matplotlib.pyplot as plt
plt.scatter(firstPrefs, secondPrefs)
plt.show()
```



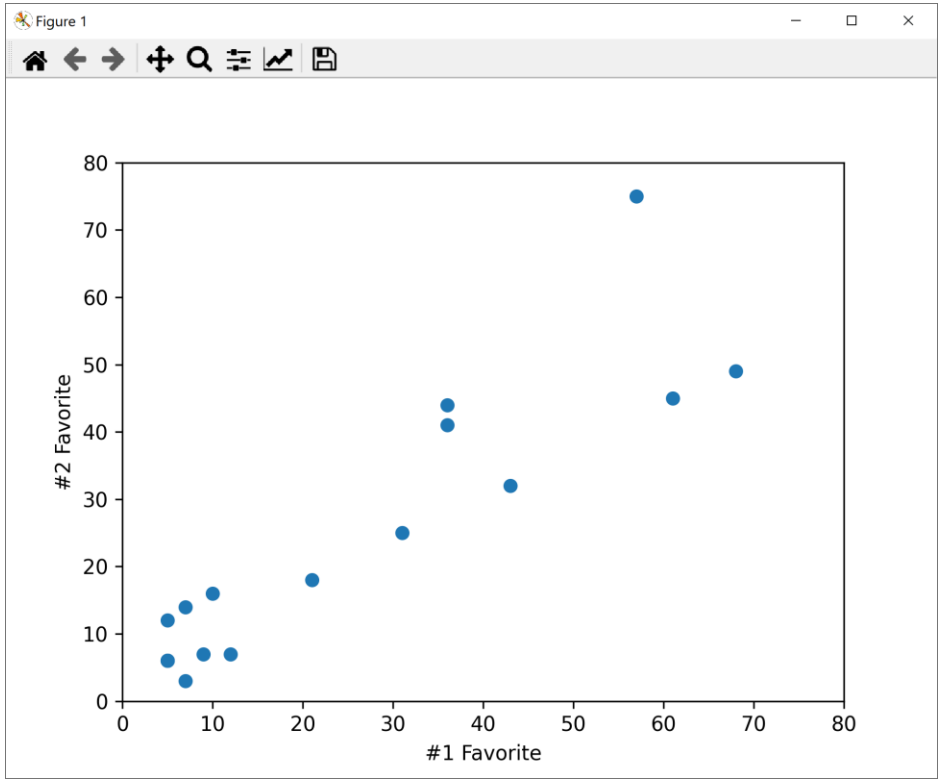
Modifying the Axes

If we want to make the chart look nicer, we might want to add **labels** and set **limits** to the axis lengths.

Searching the documentation shows us the way again!

https://matplotlib.org/stable/api/_as_gen/matplotlib.pyplot.xlabel.html

https://matplotlib.org/stable/api/_as_gen/matplotlib.pyplot.xlim.html




```
plt.xlabel("#1 Favorite")
plt.ylabel("#2 Favorite")
plt.xlim(left=0, right=80)
plt.ylim(bottom=0, top=80)
```

Finding Scatterplot Labels

Finally, to make the data easier to understand, we might want to add labels to the data points.

Nothing immediately pops out as relevant in the `scatter` API, and there are no relevant examples. Try searching the internet!

'add labels to scatterplot matplotlib' gives us: https://stackoverflow.com/questions/14432557/matplotlib-scatter-plot-with-different-text-at-each-data-point , which mentions the `annotate` function. Jackpot!

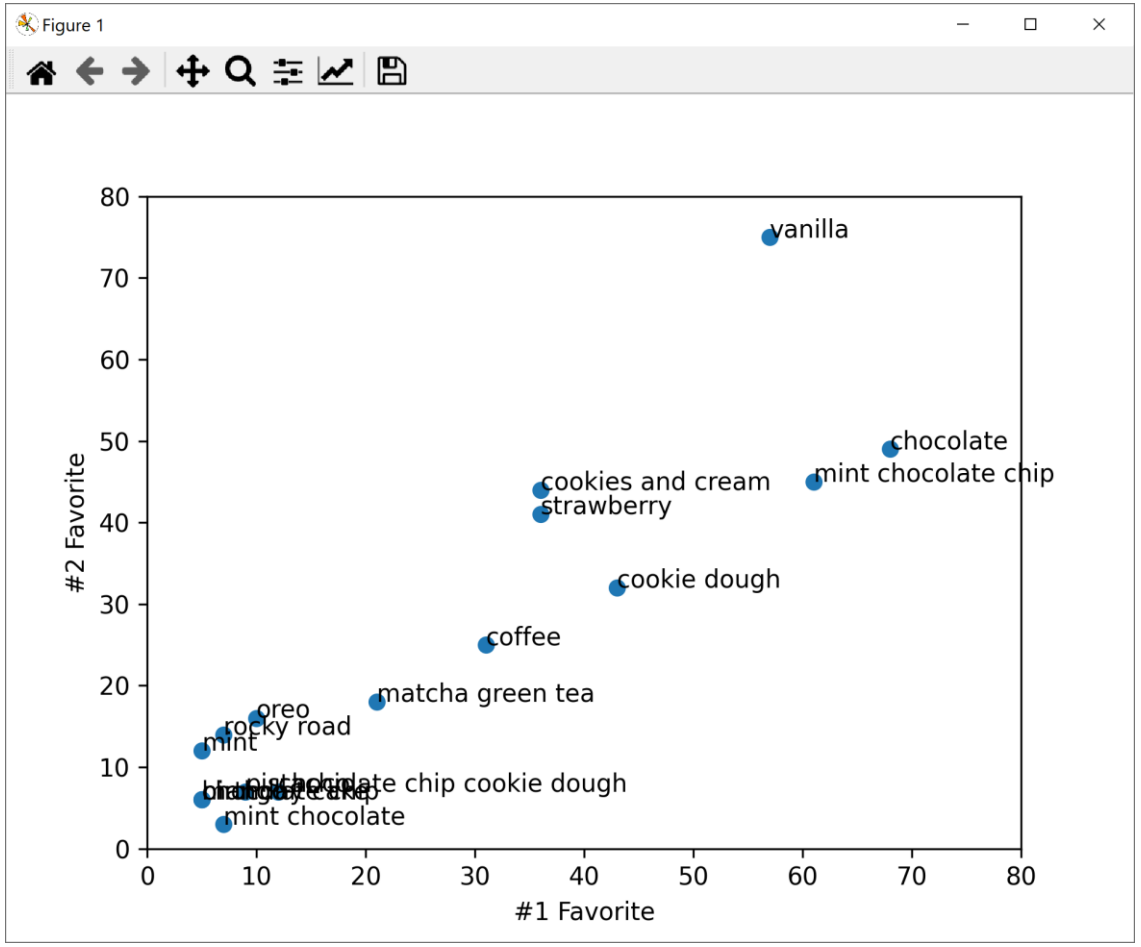
Look it up in the documentation: https://matplotlib.org/stable/api/_as_gen/matplotlib.pyplot.annotate.html

Adding Annotations

Iterate over all three lists to add the labels.

Now our graph is complete!

```
for i in range(len(flavors)):
    pos = [ firstPrefs[i],
            secondPrefs[i] ]
    plt.annotate(flavors[i], pos)
```



Simulation Coding

Model, View, Controller

Recall that our simulation framework separates the simulation into three parts.

Model: represents the current state of the simulation (as a dictionary mapping key-names to value-values and a set of rules).

View: draws the state of the world graphically

Controller: tells the model when to run rules based on **time passing** or **events**.

Example: Generating Bubbles

Let's program a basic simulation that makes bubbles appear in random locations as time passes, or when the user clicks on the screen

At any given moment, the **state** of the simulation is the number of bubbles and where they're located. That means the bubbles will need to be stored in the **model**.

Represent each bubble as an x-y location and a color. Maybe we can start with a single bubble in the middle of the screen

```
def makeModel(data):
    data["bubbles"] = [ [200, 200, "blue"] ]
```

View

To see the model, we'll need to implement the **view**. The canvas is constantly erased and re-painted with the current state of the model so that it is always up-to-date.

```
def makeView(data, canvas):
    for bubble in data["bubbles"]:
        x = bubble[0]
        y = bubble[1]
        color = bubble[2]
        radius = 20
        canvas.create_oval(x - radius, y - radius,
                           x + radius, y + radius, fill=color)
```

Time-Based Simulation

We want bubbles to show up in random locations as time passes. To do this, we'll use a function that is repeatedly called by the framework every time a certain amount of time has passed.

By adding code to this function (`runRules`), we can make the simulation smoothly update over time.

The function will need to change **the model's dictionary** for the effects to stick over time.

Time Rules

Every time a certain amount of time passes, we want to add a single bubble to the screen. To do this, **add a single bubble to the model**. Note that we **must** update the `data` variable to save the changes.

```
def runRules(data):
    x = random.randint(0, 400)
    y = random.randint(0, 400)
    color = random.choice(["red", "orange", "yellow", "green",
                           "blue", "purple"])
    bubble = [x, y, color]
    data["bubbles"].append(bubble)
```

Event-Based Simulation

We also want bubbles to show up when the user clicks on the screen. To do this, we'll use a function that is called whenever the simulation receives a mouse-click event from the computer system.

Again, this function will change the **model's dictionary** in order to represent change in the system.

Event Rules

When the user clicks on the screen, add a circle centered at that location with no color.

```
def mousePressed(data, event):
    bubble = [ event.x, event.y, None ]
    data["bubbles"].append(bubble)
```

More Event Rules

Maybe we want to let the user select the color of the circle they'll add by typing the first letter of the color. We can do with a different kind of event and a different function that is called.

However, we'll need to **store** the chosen color in `data` to share it between the two event methods.

Updated Event Rules

```
def makeModel(data):
    data["bubbles"] = [ [ 200, 200, "green" ] ]
    data["color"] = None

def keyPressed(data, event):
    colorMap = { "r" : "red", "o" : "orange", "y" : "yellow",
                 "g" : "green", "b" : "blue", "p" : "purple" }
    if event.char in colorMap:
        data["color"] = colorMap[event.char]

def mousePressed(data, event):
    bubble = [ event.x, event.y, data["color"] ]
    data["bubbles"].append(bubble)
```

Agenda

- Unit 4 Overview
- Reading and Reformatting Data
- Matplotlib Coding
- Simulation Coding