CS 341: Programming Languages Design and Implementation
Spring 2023

# Project 01 : Analyzing CTA2 L data in Python (doc v1.0)

**Assignment:** **Python program to analyze CTA2 L data stored in SQLite**
**Evaluation:** **Gradescope followed by manual execution & review**
**Policy:** **Individual work only**
**Complete By:** **Thursday February 9th @ 11:59pm CDT**
Late submissions: 10% penalty if submitted late, but before Friday 2/10 @ 11:59pm
30% penalty if submitted late, but before Saturday 2/11 @ 11:59pm

**Pre-requisites:** **Homework 01 and 02**

## Overview

The goal in project 01 is to write a console-based Python program that inputs commands from the user and outputs data from the CTA2 L daily ridership database. The program starts by outputting some basic stats retrieved from the database:

```
** Welcome to CTA L analysis app **

General stats:
  # of stations: 147
  # of stops: 302
  # of ride entries: 1,070,894
  date range: 2001-01-01 - 2021-07-31
  Total ridership: 3,377,404,512
  Weekday ridership: 2,778,644,946 (82.27%)
  Saturday ridership: 330,165,977 (9.78%)
  Sunday/holiday ridership: 268,593,589 (7.95%)

Please enter a command (1-9, x to exit):
```

The percentages shown in the last 3 outputs are computed using Python; all other data shown is retrieved / computed using SQL. This is a project requirement. You'll also need to match the output exactly. Note that these values may change based on the database tested against; when you submit to Gradescope for testing, we reserve the right to change the contents of the database (the schema will remain the same, but the underlying data may change to confirm you are writing a general-purpose program).

After the stats, the program starts a command-loop, inputting string-based commands "1" – "9" or "x" to exit. All other inputs should yield an error message:

```
Please enter a command (1-9, x to exit): 0
**Error, unknown command, try again...
```

```
Please enter a command (1-9, x to exit): 10
**Error, unknown command, try again...

Please enter a command (1-9, x to exit): help
**Error, unknown command, try again...
```

Some of the commands are simpler, e.g. command “3” outputs the top-10 stations in terms of ridership:

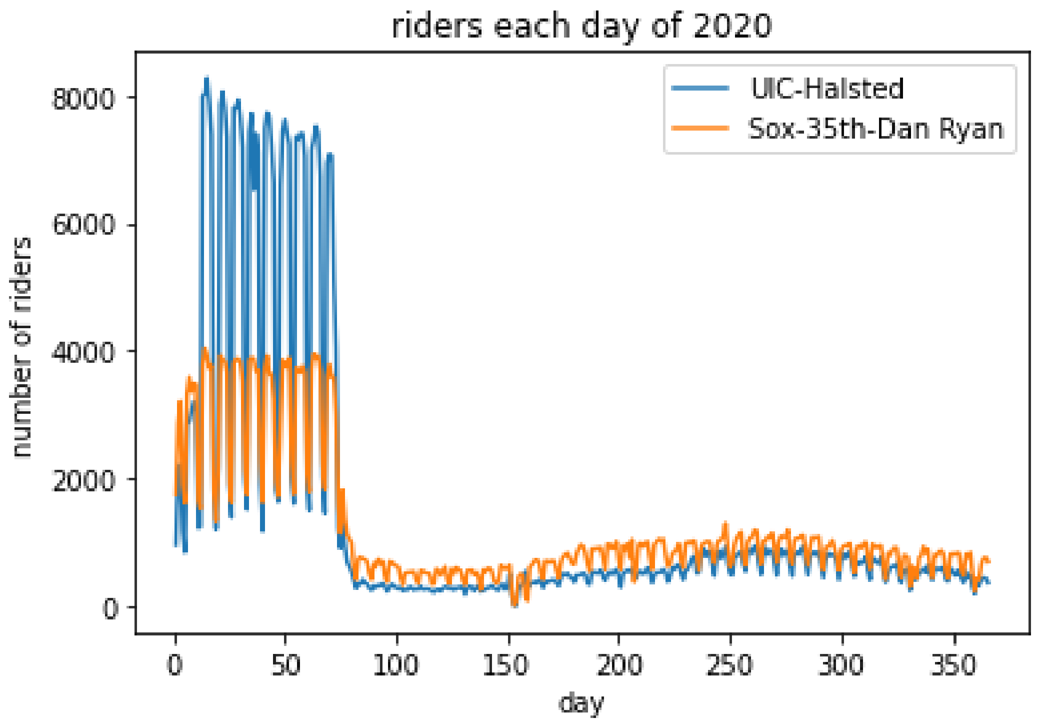
```
Please enter a command (1-9, x to exit): 3
** top-10 stations **
Lake/State : 100,419,088 (2.97%)
Clark/Lake : 100,088,085 (2.96%)
Chicago/State : 91,899,932 (2.72%)
Belmont-North Main : 74,452,064 (2.20%)
95th/Dan Ryan : 74,235,360 (2.20%)
Fullerton : 72,888,906 (2.16%)
Grand/State : 68,379,115 (2.02%)
O'Hare Airport : 66,363,838 (1.96%)
Jackson/State : 61,803,911 (1.83%)
Roosevelt : 61,487,262 (1.82%)
```

Note these answers may change based on the database tested against. Some of the commands are more challenging, and will require plotting. For example, command “8” compares ridership at two different stations for a given year:

```
Please enter a command (1-9, x to exit): 8

Year to compare against? 2020

Enter station 1 (wildcards _ and %): %uic%

Enter station 2 (wildcards _ and %): %sox%
Station 1: 40350 UIC-Halsted
2020-01-01 958
2020-01-02 2143
2020-01-03 2215
2020-01-04 1170
2020-01-05 840
2020-12-27 327
2020-12-28 426
2020-12-29 438
2020-12-30 429
2020-12-31 363
Station 2: 40190 Sox-35th-Dan Ryan
2020-01-01 1747
2020-01-02 2865
2020-01-03 3206
2020-01-04 1907
2020-01-05 1612
2020-12-27 424
2020-12-28 664
2020-12-29 749
```

```
2020-12-30 757
2020-12-31 694

Plot? (y/n) y
```

If the user responds with "y" your program should plot as follows (with appropriate title, legend, and axis labels):

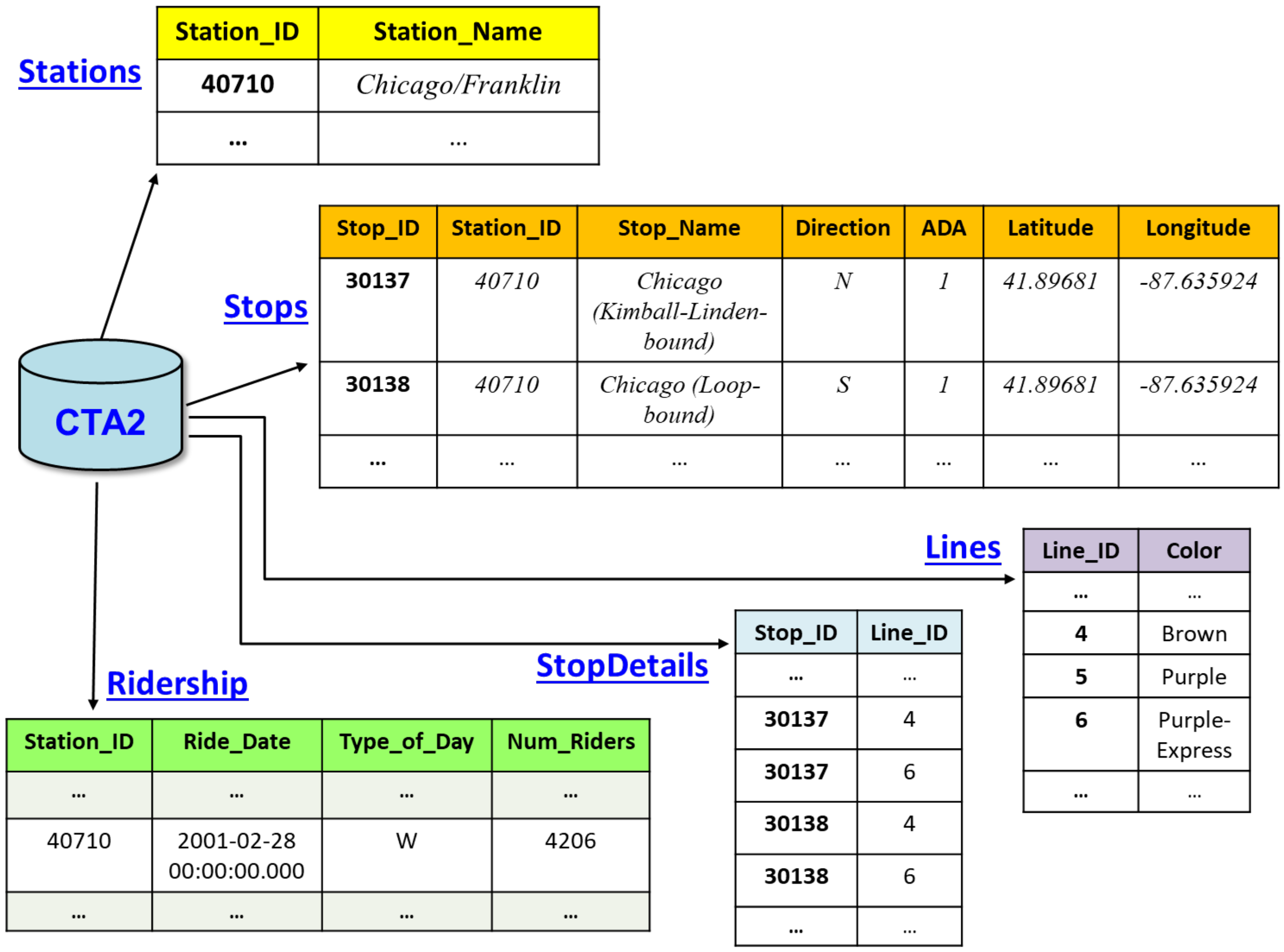


If the user responds with any other input, do not plot. [ This will be important for Gradescope testing. ]

The program should repeat until the user inputs "x" to exit the command loop. The user can input commands in any order, and may repeat commands as often as they want. The next section will detail the expected output from each command "1" – "9".

## CTA2: an update to the original CTA database

The **CTA2** database consists of 5 tables: **Stations**, **Stops**, **Ridership**, **StopDetails**, and **Lines**. This is the same database that was used in Homework 2. The following information about the database is the same information that was provided in Homework 2 (i.e. we haven't snuck anything in to attempt to trip you up). This provides information about both stations and stops in the L system:



Stations

| Station_ID | Station_Name |
|---|---|
| **40710** | *Chicago/Franklin* |
| ... | ... |

Stops

| Stop_ID | Station_ID | Stop_Name | Direction | ADA | Latitude | Longitude |
|---|---|---|---|---|---|---|
| **30137** | *40710* | *Chicago (Kimball-Linden-bound)* | *N* | *1* | *41.89681* | *-87.635924* |
| **30138** | *40710* | *Chicago (Loop-bound)* | *S* | *1* | *41.89681* | *-87.635924* |
| ... | ... | ... | ... | ... | ... | ... |

Lines

| Line_ID | Color |
|---|---|
| ... | ... |
| **4** | Brown |
| **5** | Purple |
| **6** | Purple-Express |
| ... | ... |

StopDetails

| Stop_ID | Line_ID |
|---|---|
| ... | ... |
| **30137** | 4 |
| **30137** | 6 |
| **30138** | 4 |
| **30138** | 6 |
| ... | ... |

Ridership

| Station_ID | Ride_Date | Type_of_Day | Num_Riders |
|---|---|---|---|
| ... | ... | ... | ... |
| 40710 | 2001-02-28 00:00:00.000 | W | 4206 |
| ... | ... | ... | ... |

Here's more detail about the CTA2 database:

**Stations**:

- Denotes the stations on the CTA system. A station can have one or more stops, e.g. "Chicago/Franklin" has 2 stops
- **Station_ID**: primary key, integer
- **Station_Name**: string

**Stops**:

- Denotes the stops on the CTA system. For example, "Chicago (Loop-bound)" is one of the stops at the "Chicago/Franklin" station; a Southbound stop, and handicap-accessible (ADA)
- **Stop_ID**: primary key, integer
- **Station_ID** of station this stop is associated with: foreign key, integer
- **Stop_Name**: string
- **Direction**: a string that is one of N, E, S, W
- **ADA**: integer, 1 if the stop is handicap-accessible, 0 if not
- **Latitude** and **Longitude**: position, real numbers

**Lines:**

- Denotes the CTA lines, e.g. "Red" line or "Blue" line

- **Line_ID**: primary key, integer
- **Color**: string

**StopDetails**:

- A stop may be on one or more CTA lines — e.g. "Chicago (Loop-bound)", stop 30138, is on the *Brown* and *Purple-Express* lines
- One row of StopDetail denotes one unique pair (stop_id, line_id) --- if a stop is on multiple lines such as "Chicago (Loop-bound)", it will have multiple StopDetail pairs
- **Stop_ID**: foreign key, integer
- **Line_ID**: foreign key, integer
- The pair (Stop_ID, Line_ID) forms a composite primary key

**Ridership:**

- Denotes how many riders went through the turnstile at this station on this date
- **Station_ID**: foreign key, integer
- **Ride_Date**: string in format "yyyy-mm-dd hh:mm:ss.sss"
- **Type_of_Day**: string, where 'W' denotes a weekday, 'A' denotes Saturday, and 'U' denotes Sunday or Holiday
- **Num_Riders**: integer, total # of riders who went through the turnstile on this date
- The pair (Station_ID, Ride_Date) forms a composite primary key

What is a foreign key? A foreign key is a primary key stored in another table, typically used to join those tables. Think of a foreign key as a pointer to the table where it's a primary key. Example: **Station_ID** is the primary key of the **Stations** table, and a foreign key in the **Stops** and **Ridership** tables --- this allows the Stops and Ridership tables to point to the station name in case it's needed.

## Program functionality

### Command "1"

Input a partial station name from the user (SQL wildcards _ and % allowed) and retrieve the stations that are "like" the user's input. Output station names in ascending order. If no stations are found, say so:

```
Please enter a command (1-9, x to exit): 1

Enter partial station name (wildcards _ and %): %uic%
40350 : UIC-Halsted

Please enter a command (1-9, x to exit): 1

Enter partial station name (wildcards _ and %): lake
**No stations found...

Please enter a command (1-9, x to exit): 1

Enter partial station name (wildcards _ and %): FOSTE_
40520 : Foster
```

```
Please enter a command (1-9, x to exit): 1

Enter partial station name (wildcards _ and %): %lake
40170 : Ashland-Lake
41260 : Austin-Lake
41360 : California-Lake
40280 : Central-Lake
40480 : Cicero-Lake
40380 : Clark/Lake
41160 : Clinton-Lake
40020 : Harlem-Lake
41070 : Kedzie-Lake
41510 : Morgan-Lake
41350 : Oak Park-Lake
40030 : Pulaski-Lake
40260 : State/Lake
```

Note that the SQL command you need to execute is parameterized based on the user's input. How do you do this? The simplest way is to write the SQL query with a ? where you would normally place the value, e.g.

```
sql = "Select … Where Station_Name like ? …"
```

Then, when you execute the query, you provide the value to the execute( ) function:

```
dbCursor.execute(sql, [value_to_insert_into_query])
```

## Command "2"

Output the ridership at each station, in ascending order by station name. Along with each value, output the percentage this value represents across the total L ridership. The totals must be computed using SQL, the percentages can be computed using Python (output values on the next page are omitted for brevity):

```
Please enter a command (1-9, x to exit): 2
** ridership all stations **
18th : 9,248,879 (0.27%)
35-Bronzeville-IIT : 12,800,451 (0.38%)
35th/Archer : 16,229,257 (0.48%)
43rd : 5,973,445 (0.18%)
47th-Dan Ryan : 18,648,390 (0.55%)
.
.
.
Western-Cermak : 5,803,238 (0.17%)
Western-Forest Park : 8,814,396 (0.26%)
Western-Orange : 20,905,905 (0.62%)
Western/Milwaukee : 26,646,085 (0.79%)
Wilson : 35,862,893 (1.06%)
```

I would recommend using Python to format the output. Here's one way to format each line of the output:

```
print(stationname, ":", f"{ridership:,}", f"({percentage:.2f}%)")
```

In a print statement, the "f" stands for formatted output. The ":," after the *ridership* variable means format that value with "," separators. The ":.2f" after the *percentage* variable means output the value with 2 digits following the decimal point.

## Command "3"

Output the top-10 busiest stations in terms of ridership, in descending order by ridership:

```
Please enter a command (1-9, x to exit): 3
** top-10 stations **
Lake/State : 100,419,088 (2.97%)
Clark/Lake : 100,088,085 (2.96%)
Chicago/State : 91,899,932 (2.72%)
Belmont-North Main : 74,452,064 (2.20%)
95th/Dan Ryan : 74,235,360 (2.20%)
Fullerton : 72,888,906 (2.16%)
Grand/State : 68,379,115 (2.02%)
O'Hare Airport : 66,363,838 (1.96%)
Jackson/State : 61,803,911 (1.83%)
Roosevelt : 61,487,262 (1.82%)
```

As in command #2, use SQL to compute the totals and the order, and Python to compute the percentages. Similar output formatting as in command #2.

## Command "4"

Output the least-10 busiest stations in terms of ridership, in ascending order by ridership:

```
Please enter a command (1-9, x to exit): 4
** least-10 stations **
Homan : 27 (0.00%)
Oakton-Skokie : 2,188,410 (0.06%)
Kostner : 2,471,822 (0.07%)
Cermak-McCormick Place : 2,721,448 (0.08%)
King Drive : 3,931,290 (0.12%)
Noyes : 4,488,208 (0.13%)
South Boulevard : 4,691,607 (0.14%)
Halsted/63rd : 4,707,290 (0.14%)
Foster : 4,772,450 (0.14%)
Indiana : 4,942,648 (0.15%)
```

## Command "5"

Input a line color from the user and output all stop names that are part of that line, in ascending order. If the line does not exist, say so:

```
Please enter a command (1-9, x to exit): 5

Enter a line color (e.g. Red or Yellow): yellow
Dempster-Skokie (Arrival) : direction = N (accessible? yes)
Dempster-Skokie (Howard-bound) : direction = S (accessible? yes)
Howard (Linden & Skokie-bound) : direction = N (accessible? yes)
Howard (Terminal arrival) : direction = S (accessible? yes)
Oakton-Skokie (Dempster-Skokie-bound) : direction = N (accessible? yes)
Oakton-Skokie (Howard-bound) : direction = S (accessible? yes)

Please enter a command (1-9, x to exit): 5

Enter a line color (e.g. Red or Yellow): Magenta
**No such line...
```

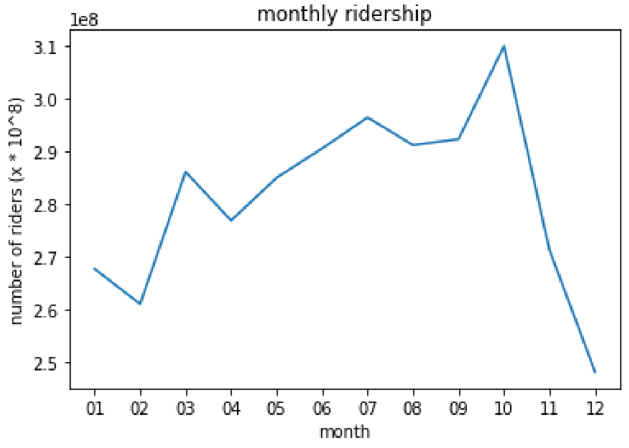
Notice the stop's direction is also output, along with whether the station is handicap-accessible or not. A stop is handicap-accessible if its ADA value is 1. The user's input should be treated as case-insensitive, e.g. "yellow" and "YELLOW" are considered the same. Do not hard-code the colors of the L lines into your Python program; the existence of a line color in the L system can easily be determined by looking at the result of your SQL query. Note that "Purple" and "Purple-Express" are considered separate lines in the L system.

## Command "6"

Outputs total ridership by month, in ascending order by month. After the output, the user is given the option to plot the data:

```
Please enter a command (1-9, x to exit): 6
** ridership by month **
01 : 267,762,005
02 : 261,119,280
03 : 286,169,400
04 : 276,946,650
05 : 285,049,263
06 : 290,602,705
07 : 296,482,279
08 : 291,240,598
09 : 292,374,159
10 : 310,014,434
11 : 271,381,428
12 : 248,262,311

Plot? (y/n) y
```

If the user responds with "y" your program should plot as follows (with appropriate title and axis labels):



If the user responds with any other input, do not plot. [ This will be important for Gradescope testing. ] Plotting in Python is straightforward using the **matplotlib.pyplot** package, which is assumed to be imported as follows:

```
import matplotlib.pyplot as plt
```

Here's the approach for plotting a series of (x, y) coordinates in a line plot:

```
x = []    # create 2 empty vectors/lists
y = []

for row in rows:  # append each (x, y) coordinate that you want to plot
    x.append(…)
    y.append(…)

plt.xlabel("…")
plt.ylabel("…")
plt.title("…")

plt.plot(x, y)
plt.show()
```

If you need to plot multiple lines, you call plot() once for each line, and then call show() at the end.

## Command "7"

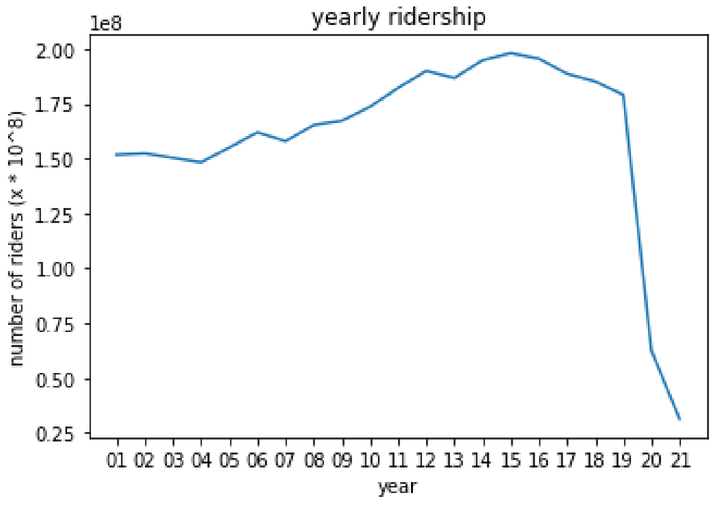
Outputs total ridership by year, in ascending order by year. After the output, the user is given the option to plot the data:

```
Please enter a command (1-9, x to exit): 7
** ridership by year **
2001 : 151,739,502
2002 : 152,364,552
2003 : 150,319,580
2004 : 148,312,412
2005 : 154,987,157
2006 : 161,966,231
```

```
2007 : 157,903,245
2008 : 165,290,763
2009 : 167,215,635
2010 : 173,561,960
2011 : 182,207,049
2012 : 189,958,315
2013 : 186,706,688
2014 : 194,826,889
2015 : 198,041,408
2016 : 195,555,726
2017 : 188,665,453
2018 : 185,146,121
2019 : 179,071,205
2020 : 62,340,303
2021 : 31,224,318
```

Plot? (y/n) **y**

If the user responds with "y" your program should plot as follows (with appropriate title and axis labels):



If the user responds with any other input, do not plot. [ This will be important for Gradescope testing. ]

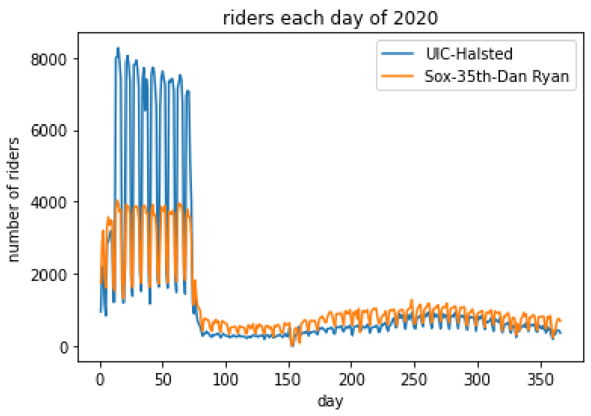
## Command "8"

Inputs a year and the names of two stations (full or partial names), and then outputs the daily ridership at each station for that year. Since the output would be quite long, you should only output the first 5 days and last 5 days of data for each station (as shown below):

```
Please enter a command (1-9, x to exit): 8

Year to compare against? 2020

Enter station 1 (wildcards _ and %): %uic%

Enter station 2 (wildcards _ and %): %sox%
```

```
Station 1: 40350 UIC-Halsted
2020-01-01 958
2020-01-02 2143
2020-01-03 2215
2020-01-04 1170
2020-01-05 840
2020-12-27 327
2020-12-28 426
2020-12-29 438
2020-12-30 429
2020-12-31 363
Station 2: 40190 Sox-35th-Dan Ryan
2020-01-01 1747
2020-01-02 2865
2020-01-03 3206
2020-01-04 1907
2020-01-05 1612
2020-12-27 424
2020-12-28 664
2020-12-29 749
2020-12-30 757
2020-12-31 694

Plot? (y/n) y
```

If the user responds with "y" your program should plot as follows (with appropriate title, legend, and axis labels) -------> If the user responds with any other input, do not plot. If the first station name entered by the user does not exist --- or yields multiple stations --- the command should abort immediately with an error message:



```
Please enter a command (1-9, x to exit): 8

Year to compare against? 2020

Enter station 1 (wildcards _ and %): uic
**No station found...

Please enter a command (1-9, x to exit):
```

Here's what should happen if the station name matches multiple stations:

```
Please enter a command (1-9, x to exit): 8

Year to compare against? 2020

Enter station 1 (wildcards _ and %): %uic%

Enter station 2 (wildcards _ and %): %lake%
**Multiple stations found...
```

```
Please enter a command (1-9, x to exit):
```

If both station names match exactly one station, the command should output the data for each station (station 1 followed by station 2), and then prompt to plot. Note that if the user enters a year for which there is no data, no error message is necessary --- the output and plot will be empty, which is sufficient.

## Command "9"

Input a line color from the user and output all station names that are part of that line, in ascending order. You will most likely get duplicates, use SQL's "distinct" to delete the duplicates for you before outputting. If the line does not exist, say so:

```
Please enter a command (1-9, x to exit): 9

Enter a line color (e.g. Red or Yellow): magenta
**No such line...

Please enter a command (1-9, x to exit): 9

Enter a line color (e.g. Red or Yellow): yellow
Dempster-Skokie : (42.038951, -87.751919)
Howard : (42.019063, -87.672892)
Oakton-Skokie : (42.02624348, -87.74722084)

Plot? (y/n) n
```

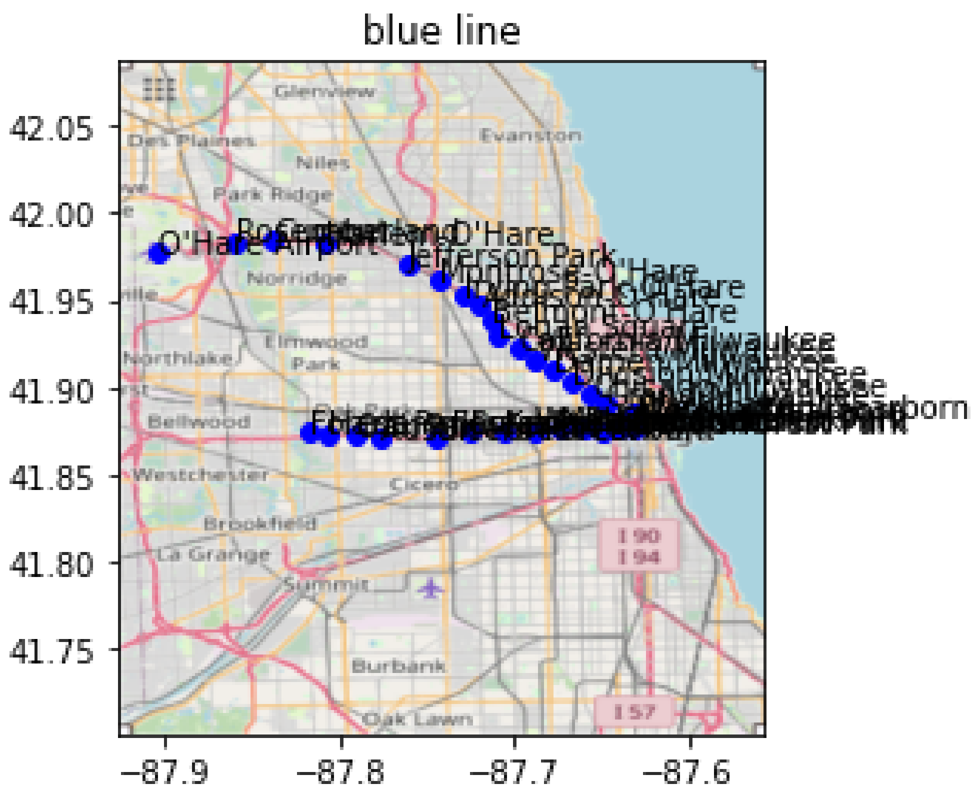
Also output the (latitude, longitude) position of each station, which are retrieved from the Stops table (assume all the stops for a given station have the same position). You'll need to use a Python formatted string to get the output to match what you see above. As in command #5, the user's input should be treated as case-insensitive, e.g. "yellow" and "YELLOW" are considered the same. Also, do not hard-code the colors of the L lines into your Python program; the existence of a line color in the L system can easily be determined by looking at the result of your SQL query.

The more interesting output is the plot, where we plot the locations of the stations overlaying a map of Chicagoland. The map is provided as an image (.png) file (on replit.com or separately as noted in the "Getting Started" section). Here's an example of the Blue line (some values are omitted for brevity):

```
Please enter a command (1-9, x to exit): 9

Enter a line color (e.g. Red or Yellow): blue
Addison-O'Hare : (41.94738, -87.71906)
Austin-Forest Park : (41.870851, -87.776812)
Belmont-O'Hare : (41.938132, -87.712359)
.
.
.
UIC-Halsted : (41.875474, -87.649707)
Washington/Dearborn : (41.883164, -87.62944)
```

```
Western-Forest Park : (41.875478, -87.688436)
Western/Milwaukee : (41.916157, -87.687364)

Plot? (y/n) y
```

Here's the plot that should be produced --- the quality will be better if you are able to use an environment such as Spyder where the plots can appear as separate figure windows (vs. inline within the environment):



This turns out to be surprisingly easy to do in Python. First, make sure the "chicago.png" image file is in the same folder as your Python program and the CTA2 database file. Then do the following:

```
#
# populate x and y lists with (x, y) coordinates --- note that longitude
# are the X values and latitude are the Y values
#
x = []
x = []
.
.
.

image = plt.imread("chicago.png")
xydims = [-87.9277, -87.5569, 41.7012, 42.0868]  # area covered by the map:
plt.imshow(image, extent=xydims)

plt.title(color + " line")
```

```
#
# color is the value input by user, we can use that to plot the
# figure *except* we need to map Purple-Express to Purple:
#
if (color.lower() == "purple-express"):
    color="Purple"  # color="#800080"

plt.plot(x, y, "o", c=color)

#
# annotate each (x, y) coordinate with its station name:
#
for row in rows:
    plt.annotate(the_station_name, (xposition, yposition))

plt.xlim([-87.9277, -87.5569])
plt.ylim([41.7012, 42.0868])

plt.show()
```

In case you're curious, here's how I created the image file, and where the xydims / xlim / ylim values came from:

```
# Map grid from min and max position values in the CTA data:
#
#   Longitude (x): -87.90422307|-87.605857
#   Latitude (y): 41.722377|42.073153
#
# How to get map image?
#   1. https://www.openstreetmap.org
#   2. search for say "Chicago"
#   3. click on export in title bar
#   4. click on "manually select a different area"
#   5. resize the box
#   6. screenshot the box, save as .png file
#   7. record the 4 coordinates of the box
```

## Getting Started

You'll be programming in Python and SQL. We've been talking about SQL in class, and you will see some examples of executing SQL in Python. But we will not be talking about Python in depth. Learning new languages is a skill you have to develop, so it's expected you'll learn the basics of Python on your own. You are free and encouraged to use internet search to help, as long as you don't hire someone or ask Chegg for the answers. Here are some references for Project 01:

- Learning python: https://www.w3schools.com/python/
- SQLite programming: https://docs.python.org/3/library/sqlite3.html#
- Plotting: https://matplotlib.org/stable/tutorials/introductory/pyplot.html

You can program using *replit.com* (see Team project "**Project 01**"), or whatever Python3 programming environment you prefer (I recommend Spyder, which runs on all platforms, but it is a large install). If you program outside of replit.com, you'll need to download the following files:

- CTA2_L_daily_ridership.db
- chicago.png

A simple Python program is available as starter code, which establishes a connection to the CTA2 database and calls a function to output one of the required stats:

```
#
# header comment? Overview, name, etc.
#
import sqlite3
import matplotlib.pyplot as plt

##################################################################
#
# print_stats
#
# Given a connection to the CTA database, executes various
# SQL queries to retrieve and output basic stats.
#
def print_stats(dbConn):
    dbCursor = dbConn.cursor()

    print("General stats:")

    dbCursor.execute("Select count(*) From Stations;")
    row = dbCursor.fetchone();
    print("  # of stations:", f"{row[0]:,}")

##################################################################
#
# main
#
print('** Welcome to CTA L analysis app **')
print()

dbConn = sqlite3.connect('CTA2_L_daily_ridership.db')

print_stats(dbConn)

#
# done
#
```

## Project requirements

The main requirements is that you use Python3 and SQL, and the provided SQLite database. We also expect good programming, i.e. the use of functions and comments. In a 3xx class this should be obvious and not require further explanation. But to be clear, if you submit a program with no functions and no comments, you will be significantly penalized --- in fact you can expect a score of 0, even if the program produces the correct results. This is not an intro class, we expect mature program design from everyone. How many functions? How many comments? Don't ask, you decide --- make reasonable decisions and you won't be penalized.

Additionally, all data retrieval and computation should be performed using explicit, string-based SQL queries executed via the **sqlite3** package; no tool-generated code is allowed (e.g. you cannot use SQLAlchemy). This implies that SQL must be used to do the vast majority of computation, e.g. all searching and summing and sorting. It is not valid to load all the data into objects and then write code to do the searching and sorting yourself. Use SQL, that's the point. For plotting, use the **matplotlib** package. [ *Some of the expected output contains percentages; computation of percentages in this case is best done using Python, so the use of SQL is not required.* ]

## Have a question? Use Piazza, not email

As discussed in the syllabus, questions should be posted to our course Piazza site — questions via email are typically ignored. Remember the guidelines for using Piazza:

1. *Look before you post* — the main advantage of Piazza is that common questions are already answered, so search for an existing answer before you post a question.
2. Post publicly — only post privately when asked by the staff, or when it's absolutely necessary (e.g. the question is of a personal nature). Private posts defeat the purpose of piazza, which is answering questions to the benefit of everyone.
3. Ask pointed questions — do not post a big chunk of code and then ask "help, please fix this". Staff and other students are willing to help, but we aren't going to type in that chunk of code to find the error. You need to narrow down the problem, and ask a pointed question, e.g. "on the 3rd line I get this error, I don't understand what that means…".
4. Post a screenshot — sometimes a picture captures the essence of your question better than text. Piazza allows the posting of images, for "how-to" see http://www.take-a-screenshot.org/ .

Don't post your entire answer / code — if you do, you just gave away the answer to the ENTIRE CLASS. When posting code, do so privately; there's an option to create a private post ("visible to staff only").

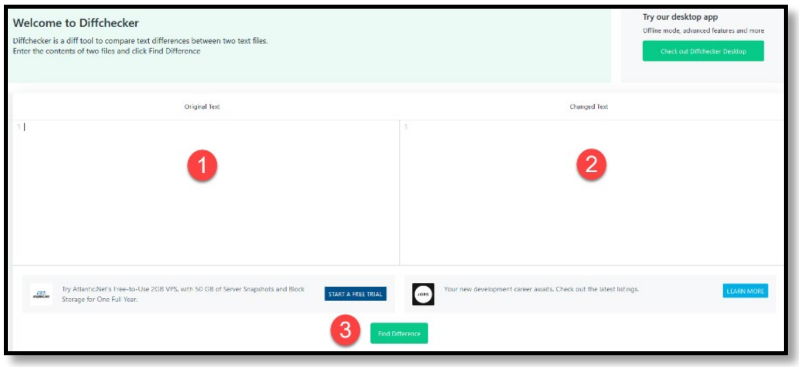
## Program submission and grading

**Submission**: submit a single "main.py" file to Gradescope under "Project 01". You have an unlimited number of submissions, though it is expected you will test locally before submitting. To encourage local testing, the Gradescope submission site will not be made available until a couple days before the early due date; do not ask when it will be released, that will only cause delay.

The score reported on Gradescope is only part of your final score (50%). After the project is due, the TAs will manually review the programs for style (10%) and adherence to requirements (0-100%), and then manually run the programs to check the required plotting functionality (40%).

**Suggestion**: when you fail a test on Gradescope, we show your output, the correct output, and the difference between the two (as computed by Linux's **diff** utility). Please study the output carefully to see where your output differs. If there are lots of differences, or you can't see the difference, here's a good tip:

1. Browse to https://www.diffchecker.com/
2. Copy your output as given by Gradescope and paste into the left window
3. Copy the correct output as given by Gradescope and paste into the right window
4. Click the "Find Difference" button



Diffchecker is a diff tool to compare text differences between two text files.
Enter the contents of two files and click Find Difference

Offline mode, advanced features and more

Try AtlanticNet's Free-to-Use 2GB VPS, with 50 GB of Server Snapshots and Block Storage for One Full Year.

Your new development career awaits. Check out the latest listings.

You'll get a visual representation of the differences. Modify your program to produce the required output, and resubmit.

In terms of grading, note that we expect all submissions to compile, run, and pass at least some of the test cases; do not expect partial credit with regards to correctness unless your program compiles and runs. By default, we grade your **last** submission. Gradescope keeps a complete submission history, so you can **activate** an earlier submission if you want us to grade a different one; this must be done before the due date. We assume *every* submission on your Gradescope account is your own work; do not submit someone else's work for any reason, otherwise it is considered academic misconduct (how else did you end up with someone else's program?).

## Academic Honesty

In this assignment, all work submitted for grading *must* be done individually. While we encourage you to talk to your peers and learn from them, this interaction must be superficial with regards to all work submitted for grading. This means you *cannot* work in teams, you cannot work side-by-side, you cannot submit someone else's work (partial or complete) as your own. The University's policy is available here:

https://dos.uic.edu/conductforstudents.shtml

In particular, note that you are guilty of academic dishonesty if you **extend or receive any kind of unauthorized assistance**. Absolutely no transfer of program code between students is permitted (paper or electronic), and you may not solicit code from family, friends, or online forums. Other examples of academic dishonesty include emailing your program to another student, screen sharing, copying-pasting code from the internet, working in a group on a homework assignment, and allowing a tutor, TA, or another individual to write an answer for you. Academic dishonesty is unacceptable, and penalties range from a letter grade drop to expulsion from the university; cases are handled via the official student conduct process described at https://dos.uic.edu/conductforstudents.shtml .