

Assignment #4: Baby Names

On-time deadline: 11:59 PM on Sunday, July 28

Extended deadline: 11:59 PM on Monday, July 29

This assignment can optionally be done in pairs.

This assignment consists of a pair of warmups and one larger application that you will build in several steps. As in previous assignments, you can download the starter code for this project under the “Assignments” tab on the CS106AP website.

For this assignment, you will have to run all of the programs from the command line. Recall this means using the terminal in PyCharm and running programs via the command line format discussed in Lecture 11 and used in Assignment 3. There will be no run configurations or doctest configurations packaged with the assignment. To run doctests, you should right-click on the doctests in the function you want to run and select the ‘Run’ option from the corresponding menu.

This assignment may be done in **pairs** or may be done individually. **You may only pair up with someone in the same section time and location.** If you work as a pair, **comment both members’ names** at the top of every .py file. Make **only one** assignment submission; do not turn in two copies. If you choose to work in a pair, you should make sure to read [this Pair Programming handout](#) before beginning the assignment.

If you decide to work in a pair, **we highly recommend doing the warmup problems individually**, even though you will only be uploading a single submission to Paperless. This will help ensure that both partners have a good grasp of the underlying concepts covered on the assignment.

AN IMPORTANT NOTE ON TESTING:

For each problem, we give you specific guidelines on how to begin decomposing your solution. While you can create additional functions for decomposition, **you should not change any of the function names or parameter requirements that we already provide to you in the starter code.** Since we include doctests for these pre-decomposed functions, editing the function headers can cause existing tests to fail.

We are only requiring you to write doctests for some of the functions in this assignment. Each milestone will have instructions on how many doctests we expect you to write for each function that we have defined. If you decide to define new functions, you are expected to write **at least 1** doctest for every new function you define.

All tests for a given function should cover completely separate cases, and we encourage you to consider both common use cases and edge cases as you write your doctests. Using good testing practices and thinking through possible inputs/outputs for your code will increase your likelihood of getting full functionality credit for your work!

Warmups (warmups .py)

Election Results

It's election season at Stanford! You've been given a list of strings representing all the votes by the Stanford community for Associated Students of Stanford University (ASSU) president. For example, suppose there are five presidential candidates: Zaphod Beeblebrox, Arthur Dent, Trillian McMillian, Marvin, and Mr. Zarniwoop. The list your function takes in might look like this:

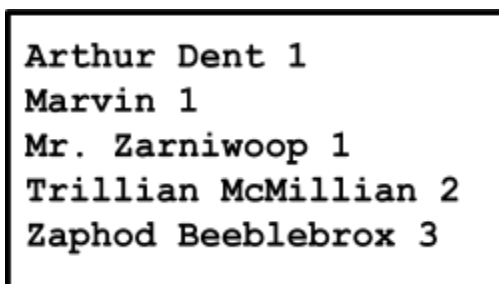
```
['Zaphod Beeblebrox', 'Arthur Dent', 'Trillian McMillian',  
'Zaphod Beeblebrox', 'Marvin', 'Mr. Zarniwoop', 'Trillian  
McMillian', 'Zaphod Beeblebrox']
```

In this list, each element represents a single vote for the presidential candidate whose name is contained in the string. Note that the list can be any length greater than zero, and there is not a fixed number of possible candidates – the five names above are just examples; there may be more or fewer candidates, and the names might be completely different. However, each candidate is guaranteed to have a unique name (elements that are the same name in the list will not be referring to different people).

Your task is to write the following function:

```
def print_vote_counts(votes)
```

The function takes in a list such as the one above and prints the number of votes that each candidate received. For example, given the list above, the program would print out the results displayed in Figure 1. **You should print out the votes per candidate in alphabetical order of the first letter of their name.**



```
Arthur Dent 1  
Marvin 1  
Mr. Zarniwoop 1  
Trillian McMillian 2  
Zaphod Beeblebrox 3
```

Figure 1: The printed results for the example list above

A dictionary dict

The problem will give you practice with lists nested inside a dictionary. In particular, we want to create a dictionary object that will act like a real-life dictionary, by grouping words based on their first letters!

Your task is to write the following function:

```
def group_by_first_letter(words)
```

The function takes in a list `words` of non-empty strings and returns a dictionary in which the keys are the unique first letters of the words within `words`. The value associated with each key is a list of the words from `words` that start with the letter indicated by the key.

For example, if your input list were

```
['Nick', 'Kylie', 'Sonja', 'kite', 'snek']
```

your function should return the following dictionary:

```
{'n': ['Nick'], 'k': ['Kylie', 'kite'], 's': ['Sonja', 'snek']}
```

Note that your function should be **case-insensitive** when associating words with keys (i.e. both “Sonja” and “snek” are both associated with the same key, lowercase “s”), but your **final dictionary should not change the casing of the original words** (i.e. we need to maintain “Sonja” rather than converting the word to “sonja”).

Files to submit: `warmups.py`

Main Program: BabyNames

BabyNames is a program that graphs the popularity of U.S. baby names from 1900 through 2010. It allows the user to analyze interesting trends in baby names over time, and it also gives you practice with data structures and simple graphics to create a large-scale application. The final, completed program that you will build is shown in Figure 2.

The rest of this handout will be broken into several sections. First, we provide an overview describing how the data itself is structured and how your program will interact with the data. All of the subsequent sections will break the problem down into more manageable milestones and further describe what you should do for each of them:

1. [Add a single name \(data processing\)](#): Write a function for adding some partial name/year/count data to a passed in dictionary.
2. [Processing a whole file \(data processing\)](#): Write a function for processing an entire data file and adding its data to a dictionary.
3. [Processing many files and enabling search \(data processing\)](#): Write one function for processing multiple data files and one function for interacting with our data (searching for data around a specific name).
4. [Run the provided graphics code \(connecting the data to the graphics\)](#): Run the provided graphics code to ensure it interacts properly with your data processing code.
5. [Draw the background grid \(data visualization\)](#): Write a function that draws an initial grid where the name data will be displayed.

- 6. [Plot the baby name data \(data visualization\)](#): Write a function for plotting the data for an inputted name.

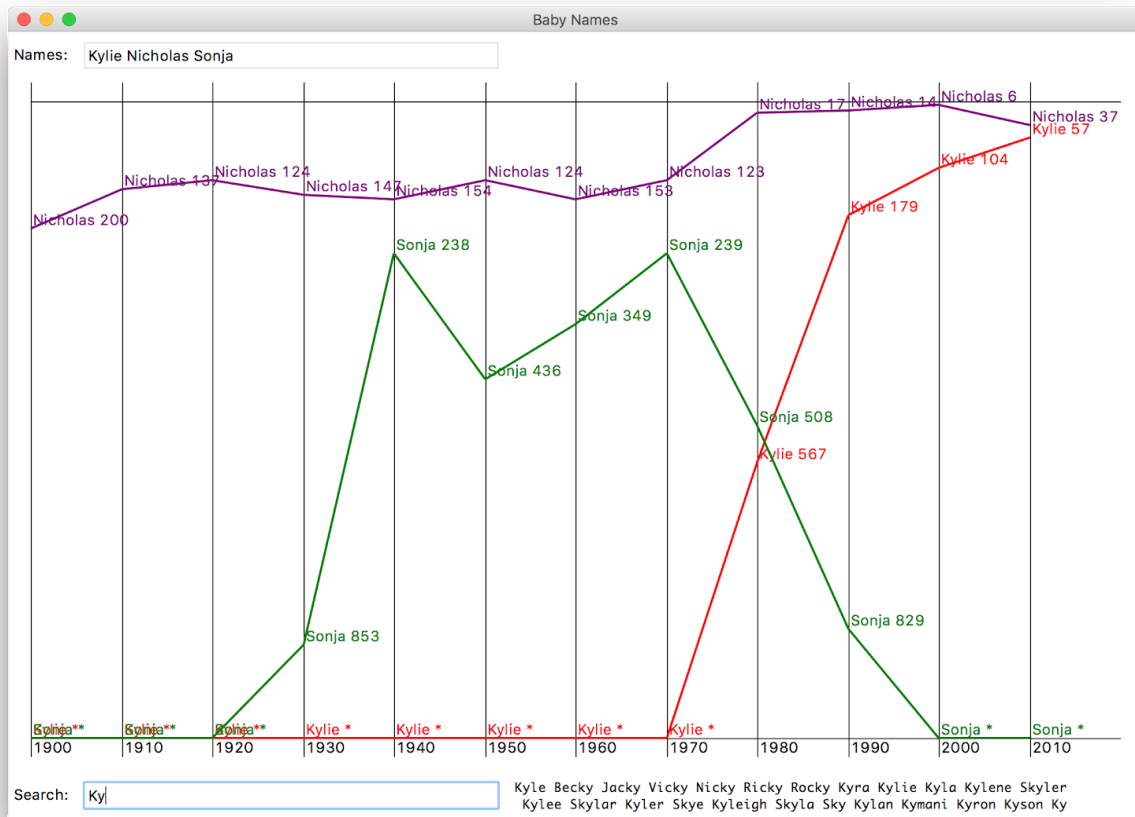


Figure 2: Sample run of the Baby Names program (with plotted names "Kylie," "Nicholas," and "Sonja"). The bottom of the window shows names that appear when searching "Ky."

The work in this assignment is divided across two files: `babynames.py` for data processing and `babygraphics.py` for data visualization. In `babynames.py`, you will write the code to build and populate the `name_data` dictionary for storing our data. In `babygraphics.py`, you will write code to use the Tkinter graphics library to build a powerful visualization of the data contained in `name_data`. We've divided the assignment this way so that you can get started on the data processing milestones (`babynames.py`) before learning about graphics.

The starter code provides empty function definitions for all of the specified milestones. While you can add additional functions for decomposition, **you should not change any of the function names or parameter requirements that we already provide to you in the starter code**. Since we include doctests or other forms of testing for these pre-decomposed functions, editing the function headers can cause existing tests to fail. Additionally, we will be expecting the exact function definitions we have provided when we grade your code. Making any changes to these definitions will make it very difficult for us to grade your submission.

IMPLEMENTATION TIP:

We highly recommend reading over all of the parts of this assignment first to get a sense of what you're being asked to do before you start coding. It's much harder to write the program if you just implement each separate milestone without understanding how it fits into the larger picture (e.g. It's difficult to understand why milestone 1 is asking you to add a name to a dictionary without understanding what the dictionary will be used for or where the data will come from).

If you're not sure about why you're being asked to do something, we recommend drawing a program diagram (covered at the beginning of Lecture 13) to map out the individual functions, their inputs and outputs, and the callers/callees. Writing the purpose of each function out in your own words allows you to confirm that you understand the problem and also helps you retain the conceptual material.

Overview

Every year, the Social Security Administration releases data about the 1000 most popular names for babies born in the U.S. at <http://www.ssa.gov/OACT/babynames/>. If you go and explore the website, you can see that the data for a single year is presented in tabular form that looks something like the data in Figure 3 (we chose the year 2000 because that is close to the year that many of the people currently in the class were born!):

Name popularity in 2000

Rank	Male name	Female name
1	Jacob	Emily
2	Michael	Hannah
3	Matthew	Madison
4	Joshua	Ashley
5	Christopher	Sarah
...		

Figure 3: Social Security Administration baby data from the year 2000 in tabular form

In this data set, rank 1 means the most popular name, rank 2 means next most popular, and so on down through rank 1000. While we hope the application of visualizing real-world data will be exciting for you, we want to acknowledge two limitations of the government dataset we're using:

- The data is divided into "male" and "female" columns to reflect the practice of assigning a biological sex to babies at birth. Unfortunately, babies who are intersex at birth are not included in the dataset due to the way in which the data has been historically collected.

- Since this data is drawn from the names of babies born in the United States, it does not capture the names of many people living in the United States who have immigrated here.

A good potential extension to this assignment might include finding and displaying datasets that have data about a wider range of people!

Like many datasets that you will encounter in real life, this data can be boiled down to a single text file that looks something like Figure 4 (data shown for 1980 and 2000). The files are included in the `data` folder of the project's starter code so you can also take a look at them yourself!

You should note the following about the structure of each file:

- Each file begins with a single line that contains the year for which the data was collected, followed by many lines containing the actual name rankings for that year.
- Each line of the file (except the first one) contains an integer rank, a male name, and a female name, all of which are separated by commas.
- Each line may also contain arbitrary whitespace around the names and ranks.

<code>baby-1980.txt</code>	<code>baby-2000.txt</code>
1980	2000
1,Michael, Jennifer	1, Jacob, Emily
2, Christopher, Amanda	2, Michael, Hannah
3, Jason, Jessica	3, Matthew, Madison
4, David, Melissa	4, Joshua, Ashley
5, James, Sarah	5, Christopher, Sarah
.
780, Jessica, Juliana	240, Marcos, Gianna
781, Mikel, Charissa	241, Cooper, Juliana
782, Osvaldo, Fatima	242, Elias, Fatima
783, Eduardo, Shara	243, Brenden, Allyson
784, Raymundo, Corrie	244, Israel, Gracie
.

Figure 4: File format for Social Security Administration baby name data

A rank of 1 indicates the most popular name that year, while a rank of 997 indicates a name that is not very popular. As you can see from the two small file excerpts in Figure 4, the popularity of names evolves over time. The most popular women's name in 1980 (Jennifer) doesn't even appear in the top five names in 2000, only 20 years later. Fatima barely appears in the 1980s (at rank #782) but by 2000 is up to #242.

If a name does not appear in a file, then it was not in the top 1000 rankings for that year. The lines in the file happen to be in order of decreasing popularity (rank), but nothing in the assignment depends on that fact.

However, data in the real world is very frequently not in the form you need it to be. Reasonably, for the Social Security Administration, their data is organized by **year**. Each year they get all those forms filled out by parents, crunch the data together, and eventually

publish the data for that year, such as we have in `baby-2000.txt`. There's a problem though; the interesting analysis and visualization of the data described above requires organizing it by **name**, across many years. This is a highly realistic data problem, and it will be the main challenge for this project.

The goal of this assignment is to create a program that graphs this name data over time, as shown in the sample run in Figure 2. In this diagram, the user has typed the string `"Kylie Nicholas Sonja"` into the box marked "Names" and then hit `<Enter/Return>` on their keyboard, to indicate that they want to see the name data for the three names "Kylie," "Nicholas," and "Sonja." Whenever the user enters names to plot, the program creates a plot line for each name that shows the name's popularity over the decades. This visualization functionality allows us to understand the data much more effectively!

Effectively structuring data

In order to help you with the challenge of structuring and organizing the name data that is stored across many different files, we will define a nested data format that we'll stick to in the rest of this assignment. The data structure for this program (which we will refer to as **name_data**) is a dictionary that has a key for every name (a **string**) that we come across in the dataset. The value associated with each name is another dictionary (i.e. a nested dictionary), which maps a year (an **int**) to a rank (an **int**). A partial example of this data structure would look something like this:

```
{
  'Aaden': {2010: 560},
  'Aaliyah': {2000: 211, 2010: 56},
  ...
}
```

Each name has data for one or more years. In the above data, "Aaliyah" jumped from rank 211 in 2000 to 56 in 2010. The reason that "Aaden" and "Aaliyah" show up first in the dataset is that they are alphabetically the first names that show up in our entire dataset of names.

(Note that although dictionaries don't guarantee that keys are in sorted order, **you won't need to worry about this**. We handle all of the printing of **name_data** for you so you are not required to sort the keys in the outer dictionary alphabetically or in the inner dictionary chronologically. But if you're interested in seeing how it works, feel free to check out the provided `print_names()` function that we call for you for testing!)

The subsequent milestones and functions will allow us to build, populate, and display our nested dictionary data structure, which we will refer to as **name_data** throughout the handout. They are broken down into two main parts: data processing in `babynames.py` (milestones 1-3) and data visualization `babygraphics.py` (milestones 4-6).

Milestone 1: Add a single name (`add_data_for_name()` in `babynames.py`)

The `add_data_for_name()` function takes in the `name_data` dictionary, a single name, a year, and the rank associated with that name for the given year. The function then stores this information in the `name_data` dict. Eventually, we will call this function many times to build up the whole data set, but for now we will focus on just being able to add a single entry to our dictionary, as demonstrated in Figure 5. This function is short but dense.

```
{
'Kylie': {2010: 57},
'Nick': {2010: 37},
}
_____→
{
'Kylie': {2010: 57},
'Nick': {2010: 37},
'Kate': {2010: 208}
}

add_data_for_name(name_data, 2010, 208, 'Kate')
```

Figure 5: The dictionary on the left represents the `name_data` dict passed into the `add_data_for_name()` function. The dictionary on the right represents the `name_data` dict after the function has added a single name, year, and rank entry specified by the other parameters.

The `name_data` dictionary is passed in as a parameter to our `add_data_for_name()` function. The `name_data` variable created inside our function will point to the same `name_data` dictionary elsewhere in our program (i.e. the baggage tag created by our parameter points to where our dictionary is stored in memory). Since dictionaries in Python are mutable, when we modify the `name_data` dictionary inside our function, those changes will persist even after the function finishes. Therefore, we do not need to return the `name_data` dict at the end of `add_data_for_name()`. (This is also called “passing the dictionary by object reference” into the function!)

Testing milestone 1

The starter code includes two doctests to help you test your code. The tests pass in an empty dictionary to represent an empty `name_data` structure. **You should write at least 4 additional tests for `add_data_for_name()`.** Pass in a non-empty `name_data` dictionary for at least 2 of these doctests to confirm that names and years accumulate in the dictionary correctly (including one to handle the “Sammy issue” explained below).

Take a look at how the existing doctests are formatted. As we mentioned in class, writing doctests is just like running each line of code in the Python Console/Interpreter. Therefore, you will first need to create a dictionary on one doctest line before passing it into your function (line 57 of the empty starter code inside the doctests for `add_file()` has an example of creating a non-empty dictionary). Then, call your `add_data_for_name()` function with the appropriate parameters. Lastly, put `name_data` on the final doctest line, followed by the expected contents in order to evaluate your function.

We have modeled this 3-step process for you in the doctests that we have provided. You do not necessarily need to create a new dictionary for every doctest, but you need to have

at least 4 additional calls to `add_data_for_name()`. Note when writing doctests for this function, the format in which you write the expected output matters. Make sure to match the spacing and formatting we have provided, including a single space after each colon and each comma. The keys should be written in the order they were inserted into the dict.

Improperly formatted doctest output:

```
{ 'Sam' : {2010:2}, 'Annie':{2010: 30}}
```

Correctly formatted doctest output:

```
{ 'Sam': {2010: 2}, 'Annie': {2010: 30}}
```

The Sammy Issue

In rare cases, a name like 'Sammy' appears twice in a given year: once as a male name and once as a female name. We need a policy for how to handle that case. Our policy will be to store whichever rank number is smaller. That is, if 'Sammy' shows up as both rank 100 (from the male data) and rank 200 (from the female data) in 1990, you should only store 'Sammy' as having rank 100 in the year 1990. This example is illustrated in Figure 6.

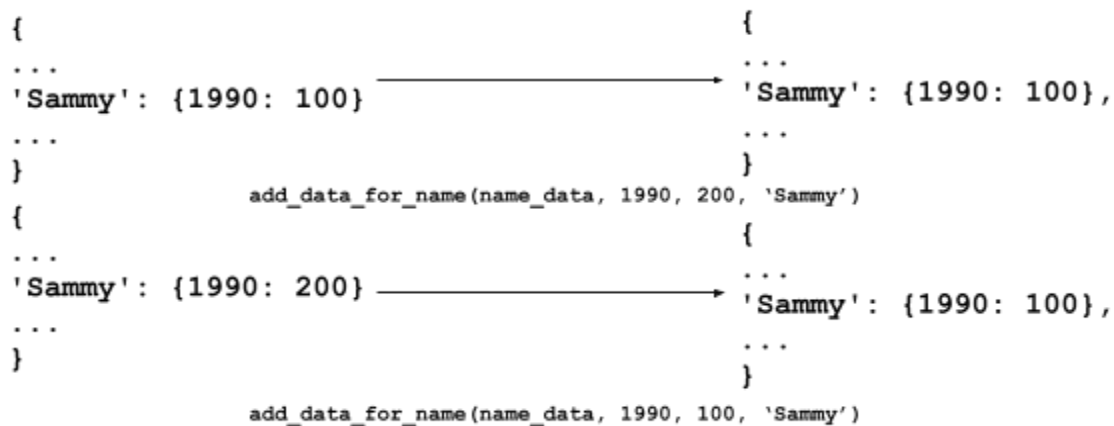


Figure 6: Two examples of how to handle the **Sammy** issue: The top example shows that if the dictionary already has a lower rank (100) for a name in a given year, then the attempt to add in a higher rank (200) would not do anything. The bottom example shows that if the dictionary is currently storing a higher rank (200) for a name in a given year, adding in a lower rank (100) would replace the old rank.

At least one of your doctests for `add_data_for_name()` should include a case that demonstrates you correctly handle the “Sammy issue.” In `baby-2000.txt` the name 'Christian' exhibits this phenomenon, and there are other such names in the dataset.

Milestone 2: Processing a whole file of data (`add_file()` in `babynames.py`)

Now that we can add a single entry to our data structure, we will focus on being able to process a whole file’s worth of baby name data. Fill in the `add_file()` function, which takes in a reference to a `name_data` dictionary and a filename. You should add the

contents of the file, one line at a time, leveraging the `add_data_for_name()` function we wrote in the previous milestone.

The format of a text file containing baby name data is shown in Figure 4 and is described again for your convenience here. The year is on the first line. The following lines each have the rank, male name, and female name separated from each other by commas. There may be some extra whitespace chars separating the data we actually care about, as seen in many of the lines in Figure 4. You should process all the lines in the file. Do not make any assumptions about the number of lines of data contained in the file. Again, since we are able to modify the `name_data` dict directly, this function does not return any values.

Tests are provided for this function, using the relatively small test files `small-2000.txt` and `small-2010.txt` to build a rudimentary `name_data` dict. **You do not need to write any additional doctests for this function.**

Milestone 3: Processing many files and searching for names in the dataset (`read_files()` and `search_names()` in `babynames.py`)

Now that we have the capability to store all of the information from a single file, we can apply our powers to read in decades worth of baby name information! Fill in the function `read_files()`, which takes in a list of filenames and builds up one big `name_data` dict that contains all of the baby name data from all the files, which is then returned. There are no doctests for this function (and we do not expect you to write any), but we will discuss how to test its output below.

Now that we have a data structure that stores lots of baby name data, organized by name, it might be good to be able to search through all the names in our dataset and return all those that we might potentially be interested in. You should write the `search_names()` function, which is given a `name_data` dict and a target string, and returns a list of all names from our dataset that contain the target string. This search should be **case-insensitive**, which means that the target strings `'aa'` and `'AA'` both match the name `'Aaliyah'`. We have not provided any doctests, so **you should write at least three doctests to test the output of this function.**

Testing Milestones 1-3

You've made it halfway! It's now time to test all the functions you've written above with some real data and then move on to crafting a display for the data.

We have provided a `main()` function in `babynames.py` for you to be able to test that all of the functions we've written so far are working properly. Perhaps more importantly, you'll be able to flex your new data organization skills on the masses of data provided by the Social Security Administration. The given `main()` function we have provided can be run in two different ways.

The first way that your program can be run is by providing one or more baby data file arguments, all of which will be passed into the `read_files()` function you have written. This data is then printed to the console by the `print_names()` function we have

provided, which prints the names in alphabetical order, along with their ranking data. A sample output can be seen below, when running the `babynames.py` program on the `small-2000.txt` and `small-2010.txt` files. These files are just for sanity-checking purposes and don't actually contain any real names.

```
$ python3 babynames.py data/small/small-2000.txt data/small/small-2010.txt
A [(2000, 1), (2010, 2)]
B [(2000, 1)]
C [(2000, 2), (2010, 1)]
D [(2010, 1)]
E [(2010, 2)]
```

The small files test that the code is working correctly, but they're just an appetizer to the main course, which is running your code on the real baby name files! You can take a look at 4 decades of data with the following command in the terminal (use the tab-key to complete file names without all the typing).

```
$ python3 babynames.py data/full/baby-1980.txt data/full/baby-1990.txt
data/full/baby-2000.txt data/full/baby-2010.txt
...output!...
```

Organizing all the data and dumping it out is impressive, but it's not the only thing we can do! The `main()` function that we have provided can also call the `search_names()` function we have written to help us filter the large amounts of data we are now able to read and store. If the first 2 command line arguments are `"-search target"`, then `main()` reads in all the data, calls your `search_names()` function to find names that have matches with the target string, and prints those names. Since the `main()` function directly prints the output of the `search_names()` function, note that the names are not printed in alphabetical order, but rather in the order in which they were added to the dictionary. Here is an example with the search target `"aa"`:

```
$ python3 babynames.py -search aa data/full/baby-2000.txt
data/full/baby-2010.txt
Aaron
Isaac
Aaliyah
Isaak
Aaden
Aarav
Ayaan
Sanaa
Ishaan
Aarush
```

You've now solved the key challenge of reading and organizing a realistic mass of data! Now we can move on to building a cool visualization for this data!

Note: Milestones 4 to 6 are built using the graphical tools discussed in the Tuesday, July 23 lecture (Lecture 17: Graphics 1.0). This means that all graphical components of this assignment can be created using basic Tkinter tools. Therefore, you will not need and are not allowed to use the campy library for this assignment.

Milestone 4: Run the provided graphics code (`babygraphics.py`)

We have provided some starter code in the `babygraphics.py` file to set up the Tkinter canvas, and then add the interactive capability to enter names to graph and target strings to search the names in the database. The code we have provided is located across two files; the Tkinter code to draw the main graphical window and set up the interactive capability is located in `babygraphicsgui.py` and the code that actually makes everything run is located in `main` in `babygraphics.py`. The provided `main()` function takes care of calling your `babynames.read_files()` function to read in the baby name data and populate the `name_data` dict. The challenge of this part of the assignment is figuring out how to write functions to graph the contents of the `name_data` dict. For this milestone, you should run the program from the command line in the usual way.

```
$ python3 babygraphics.py
```

This should bring up a window as seen in Figure 7.

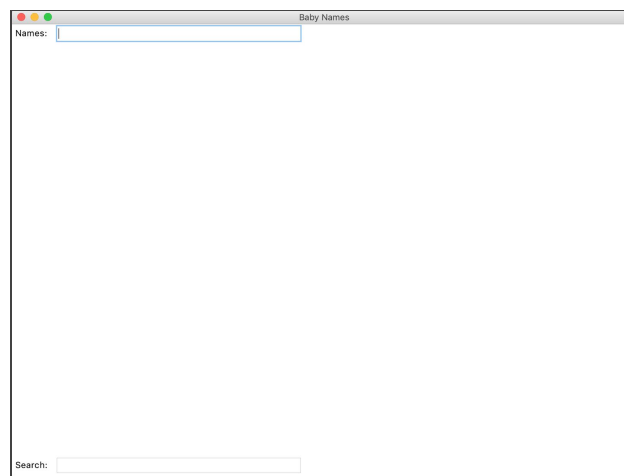


Figure 7: Blank Baby Name graphical window

Although this program might appear boring, it has actually loaded all of the baby name data information behind the scenes. To test this (and to test out the `search_name()` function you wrote in Milestone 3, try typing a search string into the text field at the bottom of the window and then hit **<Enter>**. You should see a text field pop up in the bottom of the screen showing all names in the data set that match the search string, as seen in Figure 8.



Figure 8: Bottom portion of the Baby Names window after entering the search string 'aa'

Once you have tested that you can run the graphical window and search for names, you have completed Milestone 4.

Milestone 5: Draw the background grid (`draw_fixed_lines()` in `babygraphics.py`)

Constants

Here are the constants that you will need to use for Milestones 5 and 6:

```
YEARS = [1900, 1910, 1920, 1930, 1940, 1950, 1960, 1970, 1980, 1990,
2000, 2010]
GRAPH_MARGIN_SIZE = 20
COLORS = ['red', 'purple', 'green', 'blue']
TEXT_DX = 2
LINE_WIDTH = 2
MAX_RANK = 1000
```

In this milestone, our goal is to draw the horizontal lines at the top and bottom of the canvas, the decade lines, and the labels that make up the background of the graph. To do so, you will implement the `draw_fixed_lines()` function. We have provided three lines of code in this function that clear the existing canvas and get the canvas width and height. Once this function is done, your graphical window should look like Figure 9 (minus the red highlighting).

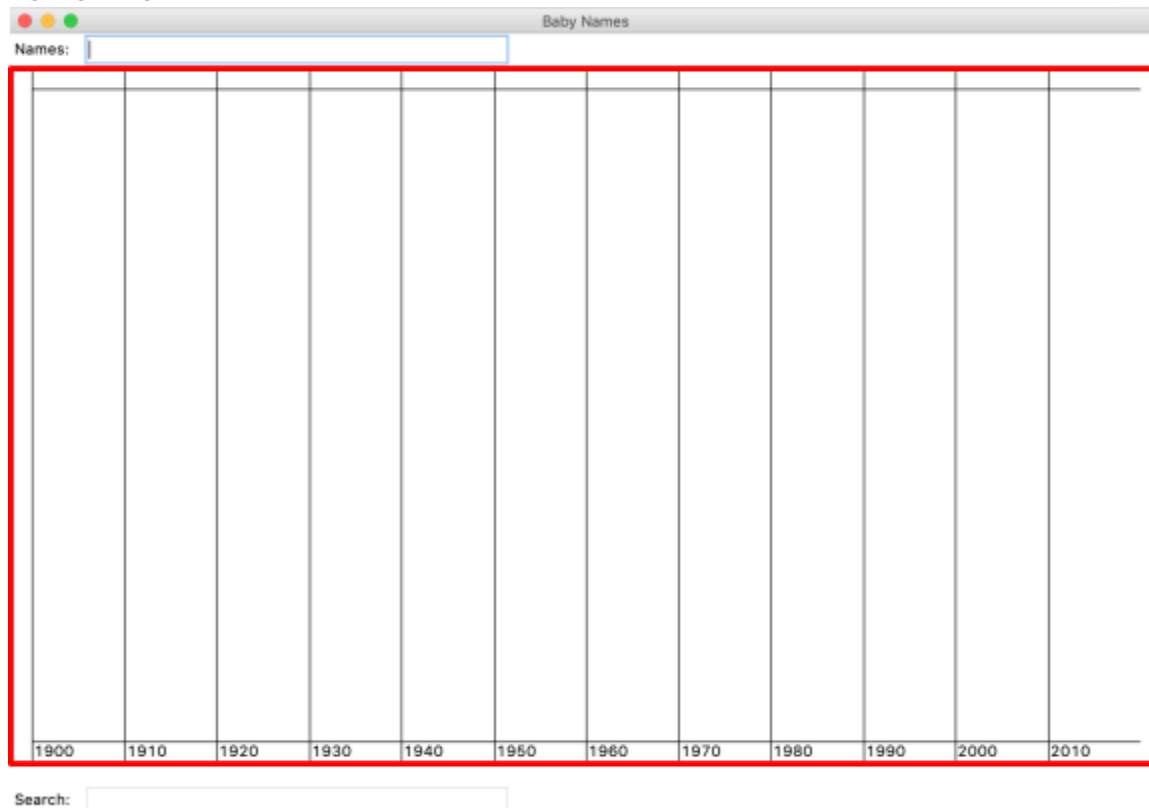


Figure 9: View of the Baby Names window after `draw_fixed()` has been called, with the canvas area highlighted in red.

Note: In the GUI (Graphical User Interface), the text field takes up the top of the window, and the canvas is a big rectangle below it. The search text field is at the bottom of the window below the canvas. When we refer to the canvas, we are talking about the red highlighted area in Figure 9.

To complete this milestone you should follow the following steps.

- There should be a **GRAPH_MARGIN_SIZE** sized margin area on the top, bottom, left, and right of the canvas (so that the year labels are always visible) as indicated in Figure 10. Draw a horizontal line at the top and the bottom of the canvas to create this margin area at the top and bottom.
- For each decade in **YEARS**, draw a vertical line from the top of the canvas to the bottom. The lines should start **GRAPH_MARGIN_SIZE** pixels in from the left edge of the window and be evenly-spaced to fill the entire width of the graph.
 - The trickiest math here is computing the **x** value for each year. For this reason, we ask you to decompose a short helper function called **get_x_coordinate()**, which takes in the width of the canvas and the index of the year you are drawing (where 1900 has index 0, 1910 has index 1, and so on) and returns the appropriate **x** coordinate. You'll need to account for the **GRAPH_MARGIN_SIZE** when writing this function, so the first line will be placed at the x-coordinate **GRAPH_MARGIN_SIZE** instead of 0. Both **draw_fixed()** and **draw_names()** (written in Milestone 6) need to have the exact same **x** coordinate for each year, which makes this a valuable function to decompose. We have provided several doctests for the **get_x_coordinate()** function, so you are not required to write more.
- For each decade in **YEARS**, add a decade label that displays that year as a string in the bottom margin area. The labels should be positioned such that their **x** coordinates are offset by **TEXT_DX** pixels from their corresponding decade line, and their **y** coordinates should be **GRAPH_MARGIN_SIZE** from the bottom of the canvas.
 - The tkinter **create_text()** function takes in an optional **anchor** parameter that indicates which corner of the text should be placed at the specified **x,y** coordinate. When calling the **create_text()** function for these labels, you should specify **anchor=tkinter.NW** as the last parameter to indicate that the **x,y** point is at the north-west corner relative to the text.

For your convenience, Figure 10 is a diagram of the line spacing for the output of the **draw_fixed_lines()** function. While this diagram has four vertical lines in it, your canvas should have twelve vertical lines. The outer edge of the canvas is shown as a rectangle, with the various lines drawn within it. Each double-arrow marks a distance of **GRAPH_MARGIN_SIZE** pixels.

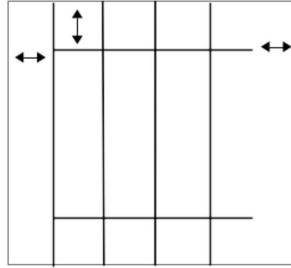


Figure 10: Line spacing diagram for Baby Names canvas

Note: The TK `create_line()` function truncates coordinates from `float` to `int` internally, so you can do your computations using floats.

By default, `main()` creates a window with a 1000 x 600 canvas in it. Try changing the constant values `CANVAS_WIDTH` and `CANVAS_HEIGHT` that are defined at the top of the program to change the size of the canvas that is drawn. Your line-drawing math should still look right for different width/height values. Note that if you specify a height and width for the canvas, the actual window itself will be larger than that because it needs space to display the text entry fields and the text field that shows the results of the targeted name search. You should also be able to temporarily change the `GRAPH_MARGIN_SIZE` constant to a value like 100, and your drawing should be able to use the new value. This shows the utility of defining constants— they allow you to define values that can be easily changed and all the lines that rely on this value remain consistent with one another.

Once your code can draw the fixed lines and year labels, for various heights and widths, you can move on to the next milestone.

Milestone 6: Plot the baby name data (`draw_names()` in `babygraphics.py`)

We have reached the final part of the assignment! Now it is time to plot the actual name data that we have worked so hard to organize. To do so, you will fill in the `draw_names()` function. We have provided three lines of code, which use your `draw_fixed_lines()` function to fill in the background and then fetch the width and height of the canvas. The parameter `lookup_names` contains a list of names like `['Kylie', 'Nick', 'Sonja']` and the parameter `name_data` contains the dict that we built up in Milestones 1-3. The `draw_names()` function is called every time that you enter some space-separated names into the top text entry field and hit `<Enter>`. The list `lookup_names` will contain the names that you enter into the top text entry field, formatted with the correct casing (uppercase first letter, lowercase other letters), which we handle for you. If you enter one or more names that are not contained in `name_data` into the top text entry field, an error message will display on the top right, and those names will be omitted from `lookup_names`.

When plotting the name data, you should consider the following tips:

- The x-coordinates of each line segment start at the vertical grid line for a particular decade and end at the vertical grid line for the following decade. Think about using a helper function that you wrote earlier in this assignment.
- The **y** coordinate value for a name's rank in a given year is defined as follows. If the name does not have a rank stored in **name_data** for a given year, you should assign it **MAX_RANK** for that year. If the rank is 1 (the best possible rank), the **y** coordinate should be equal to the **y** coordinate of the top horizontal line in the canvas. If the rank is **MAX_RANK**, the **y** coordinate should be equal to the **y** coordinate of the bottom horizontal line. All other ranks should be evenly spaced between the top and the bottom of the graph area.
- To make the data easier to read, the lines on the graph are shown in different colors. The colors that lines should be plotted in are defined in the **COLORS** constant. The first data entry is plotted in red, the second in purple, the third in green, and the fourth in blue. After that, the colors cycle around again through the same sequence. Hint: consider using the mod (%) operator. The color of a line segment can be specified with the optional **fill=color-str** parameter to **create_line()**
- For each year, you should additionally draw a rank label next to the endpoint of the line that displays that entry's name and rank for that year. If there is no rank or if the rank is 1000, you should display "*" instead of the rank (see Figure 11 for an example). The label's color should match the line's color. The x-coordinate of the ranking label is the x-coordinate of that decade's vertical line plus **TEXT_DX** pixels, and the y-coordinate is the same as the y-coordinate of that corresponding year's plot line segment. The call to **create_text()** in this part of the assignment should use **anchor=tkinter.SW** as the last optional parameter.
- When drawing line segments, you want them to appear thicker than the lines that define the boundary. To accomplish this, include the optional **width=LINE_WIDTH** parameter in any calls to **create_line()**

If you are having trouble getting the right coordinates for your lines or labels in your graph, try printing the **x/y** coordinates to verify them.

Once you think you have your code working, you should test by entering some names (separated by spaces) in the top text entry bar of the Baby Names window. '**Jennifer**' is a good test, since that name hits both the very bottom and the very top (a rags-to-riches story of baby names!). A graph with the two names '**Jennifer**' and '**Lucy**' is shown in Figure 11.

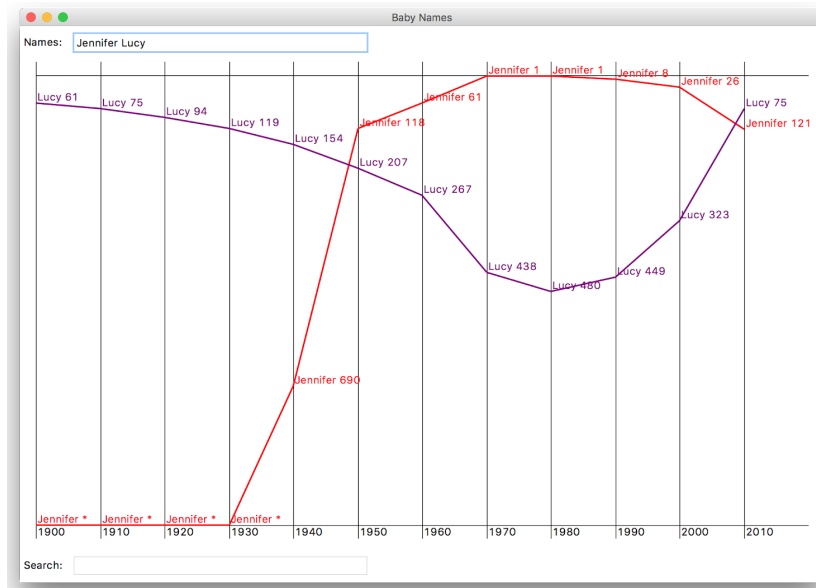


Figure 11: A fully-implemented Baby Name program displaying historical name data for the names ‘Jennifer’ and ‘Lucy’

Congratulations! You’ve wrangled a tough real-world dataset and built a cool visualization!

Files to submit: `babynames.py`, `babygraphics.py`

Optional Extra Features

There are many possibilities for optional extra features that you can add if you like, potentially for a small amount of extra credit. If you are going to do this, please *submit two versions of your program*: one that meets all the assignment requirements, and a second extended version. At the top of your extended files, in your comment header, you must **comment** what extra features you completed. Here are a few ideas:

- *Try to minimize the overprinting problem.* If the popularity of a name is improving slowly, the graph for that name will cross the label for that point, making it harder to read. You could reduce this problem by positioning the label more intelligently. If a name were increasing in popularity, you could display the label below the point; conversely, for names that are falling in popularity, you could place the label above the point. An even more challenging task is to try to reduce the problem of having labels for different names collide.
- *Plot the data differently.* Right now, your program visualizes the data by showing its popularity over time. What other information about the names could you display? Consider plotting the rate of change over time, the correlation of various names, or other interesting trends that aren’t apparent purely through their popularity.

- *Visualize another dataset.* Can you use your program to visualize a dataset of names from a different data source? What other data sets are you able to plot? The world is your oyster!