**Scenario**

We've got some data about registered baby names in certain years in a comma-separated-values file; only **read\_file** needs to interact with files, and all others will use our required structure to describe the names, genders, and counts per year

**CSV file**:

This is our name for a file containing ascii text where each line in the file represents one record of information; each piece of info is surrounded by double quotes, and each of these quoted things is separated by a single comma. The very first line is the "header" row, which names the columns but is not part of the data. Here is a very small sample file that can be used in our project.

"YEAR","GENDER","NAME","COUNT"

"2009","MALE","DANIEL","3423"

"2009","MALE","ANTHONY","3106"

"2009","MALE","ANGEL","3058"

"2010","MALE","JACOB","3368"

"2010","MALE","DANIEL","3175"

"2010","MALE","ANTHONY","2882"

**Note:** a file's extension has no actual effect on its contents. These are ascii files, so you can edit them with your code editor just as easily as a .txt or .py file

**Popularity:**

We refer to a dictionary whose keys are years, and whose values are tuples of (count, rank), as a popularity record. We use this representation to store information about a particular name for multiple years. Note – though a record belongs to a particular name (as the example below is the record for name ‘DANIEL’), notice we don't see it here – that's because it will be represented elsewhere in a database, and duplicated information is rarely a good idea in a database.

sample\_pop\_record = {2009: (3423, 1), 2010: (3175, 2)}

**Database:** a "database" of names can store multiple names from multiple years. Our database is a dictionary whose keys are tuples of (name, gender), and whose values are popularity records in the form of dictionary as defined above.

sample\_db = { ('DANIEL', 'MALE'): {2009: (3423,1), 2010: (3175, 2)}, ('ANTHONY', 'MALE'): {2009: (3106, 2), 2010: (2882, 3)}, ('ANGEL', 'MALE'): {2009: (3058, 3)}, ('JACOB', 'MALE'): {2009: (2978, 4), 2010: (3368, 1) }

We either call a database ranked, where all ranks have been correctly filled in, or unranked, where ranks are either None or no longer correct due to an addition.

**NOTE:** We will use a few different csv files as our examples and in testing. They come from the shared zip file (we will provide)

**What's allowed?**

Here is the exhaustive list of things you can use on the project.   
• all basic expressions and statements   
• functions: len(), range(), min(), max(), enumerate(), int(), and any functions you write   
• file reading: open(), close(), read(), readline(), readlines(), with syntax   
• dictionaries: len, get, clear, copy, keys, values, items, pop, popitem, update   
• methods: lists: remove(), insert(), append(), extend(), pop(), index() strings: split(), endswith(), startswith(), join(), replace(), insert()   
• sorted(), sort(), reversed(), reverse() This means that…   
• you can't call anything not listed above. Focus on applying these functions to solve the task.   
• you can't import any modules for this project. (so you can't import csv either – it isn't that helpful)

**Functions dealing with unranked databases**

**These functions either work on, or create, unranked databases.**

1. **read\_file (filename):** This is the \*only\* function that needs to deal with reading a file. It will accept the file name as a string, and assumes it is a CSV file as described above (with our name data in the same format as the example, but with any number of rows after the header row). It will open the file, read all the name entries, and correctly create the unranked database.   
   • Return the resulting database.   
   • Set all rankings to None.   
   • Hint: How can you break this task down into multiple phases, each one taking a pass over the data and making something slightly more useful towards getting the result? How can you use any functions that you have to write, or could write?   
   • If you get stuck on this one, you can still attempt others – just test with the tester or with manually built database values.
2. **add\_name (db, name, gender, year, count):** This function accepts an existing database db, name, gender, and year, and then it updates the database to include that information.

• Return None.   
• the rank for that entry is set to be None.   
• The resulting database is considered unranked, as the addition means the rankings aren't up-to-date. It's acceptable that there are some None ranks and some numeric ranks.

1. **new\_names(db, gender, old\_year, new\_year)**: accepts a database db, gender, and a pair of years. It searches in the database for names of that gender that do not make the top list for old\_year, but do show up in the list for new\_year. Return the qualified names as a list of strings, alphabetically sorted

**Functions dealing with ranked databases**

These functions either make a database become ranked, or they rely upon ranked databases. If a function expects to receive a ranked database, you are not required to handle receiving an unranked database in any particular way – that behavior is undefined and won't be tested. You're welcome to still try to handle it if you want.

1. **rank\_names\_for\_one\_year (db, year):** This function accepts an existing database and a year. It calculates the ranking of names according to their counts and updates that information into the database. Rank boys’ and girls’ names separately. The most popular name for each gender (with the highest count) gets a rank value of 1.   
   • This function should return None.
2. **rank\_names (db)** : This function accepts an existing database and ranks all names for all years of data present, making the database become ranked.   
   • This function should return None.   
   • rank male and female names separately.   
   • If there is a tie in counts, assign all names with the same rank and make sure the next rank is adjusted accordingly. Given counts of A:10, B:5, C:5, D:5, E:1, they'd get rankings of A=1, B=2, C=2, D=2, E=5.   
   • Hint: use previous functions!
3. **popularity\_by\_name(rdb, name, gender):** accepts a ranked database, name, and gender. It finds the popularity counts for all years included in the db for name, assemble them in a list of tuples (year, rank), and return the list. If db has no records for name, return []. Sort multiple years’ records (tuples) by year
4. **popularity\_by\_year(rdb, gender, year, top=10):** accepts a ranked database, gender, year, and top. It finds for the specified year, the top popular names and returns them in a list of tuples (rank, name). Sort multiple tuples in your return list by rank most common first). You can assume top is always a positive integer. If top is not provided, use default value and report the top 10 popular names. If top is larger than the number of stored names for that year, report all names in the right order.
5. **always\_popular\_names(rdb, gender, years=None, top=10):** accepts a ranked database, gender, a list of years and a top threshold. It searches in the database for (name, gender) records that are always ranked within (and including) top for all the years in the given list and return them as a list of strings.   
   • If years==None, use all years present anywhere in the database.   
   • If top is not provided, default to 10.   
   • in the answer, sort the names alphabetically
6. **increasing\_rank\_names(rdb, gender, old\_year, new\_year):** accepts a ranked database, gender, and two years. It searches in the database for names whose rank gets promoted from old\_year to new\_year, and returns them as a list of strings. If there are multiple names, sort them alphabetically. If a name does not get ranked for a year, do not include that name in the return list