**Using Python to analyze a multiplayer game of war**

When I was younger one of my brothers and two of my cousins would always play the game of war. He would change the rules by playing with only four cards while my cousins and I split the rest of the deck. However, the four cards he played with were the four aces. My cousins and I never seemed to win. I wanted to see how often, under these circumstances, we would win.

I decided to use Python since I have just recently learned the language from a course on LinkedIn. Writing this application would enable me to delve deeper into some of the more complex aspects of this language. This program will take the classic war game and play a defined number of hands while saving the results of this to a *dataFrame*. This file can later be saved to a file.

1. **Initial tools used for developing this class**
2. **Overview of initial findings for using these tools**

I chose to try out three different tools for this development. These were Anaconda, Komodo, and Visual Studio Code with Python (3.7). Overall, I found Anaconda to be the best both in terms of performance and initial setup. However, I really liked the way Visual Studio Code handled interactive debugging.

|  |  |
| --- | --- |
| Anaconda:  Spyder (Ver 4.0.1) | Pros:   1. Easiest to setup since it included Jupyter (support for both inline graphs), support for all the standard Python import: numpy, pandas, etc. 2. Did not need to manually close each pop out graph when a program, like this one, generates multiple graphs. (Also led to a con, see below.) 3. Had the best performance (22% faster than Komodo) 4. Supported inline graphing without additional effort   Cons:   1. Had to “reset” the graphs between executions, otherwise pop out graphs would fail if the program was executed before closing previous graphs. This was resolved by the command on line 640: *plt.figure(figsize=(10, 8))* |
| Komodo: (Ver 11) | Pros:   1. Perhaps it is just personal preference, but I liked this editor the best for entering code.   Cons:   1. The slowest of all the editors I used. 2. Although the latest updated resolved my issue, I was initially unable to get any graphs to work. 3. Could not get the interactive debugger to work. 4. Had significant difficulty getting some of pip installs to work and had to resort to importing the binaries for one of them.   <https://stackoverflow.com/questions/26575587/cant-install-scipy-through-pip> |
| Visual Studio Code: (Ver 1.41.1) | Pros:   1. The interactive debugger worked the best and was the most intuitive for me to use. (Alright, I admit I have had previous experience with Visual Studio and perhaps that helped just little… 😊)   Cons:   1. Had difficulty getting some imports to work, just like Komodo. |

1. **Detailed overview of initial findings for using these tools**

**Installation of various python IDEs**

|  |  |  |
| --- | --- | --- |
| **Application** | **Windows installation video** | **Mac installation video** |
| Anaconda (Spyder) | Install Anaconda Python, Jupyter Notebook, and Spyder on Windows 10  <https://youtu.be/5mDYijMfSzs> | Installing Anaconda in MacOS and Setting up Virtual Environment  <https://www.youtube.com/watch?v=nVlrpNf3EdM> |
| Komodo | Python Essential Training (LinkedIn) | Python Essential Training (LinkedIn) |
| Visual Studio Code | Visual Studio Code (Windows) - Setting up a Python Development Environment and Complete Overview <https://youtu.be/-nh9rCzPJ20> | Visual Studio Code (Mac) - Setting up a Python Development Environment and Complete Overview  <https://youtu.be/06I63_p-2A4> |

|  |  |
| --- | --- |
| **Application** | **Advance function imports** |
| Anaconda | All libraries pre-installed need for this program are pre-installed |
| Komodo | Need to install all the necessary libraries (see below…) |
| Visual Studio Code | Need to install all the necessary libraries (see below…) |

To get the example program function the following imports are needed:

numpy (NUMeric Python): matrices and linear algebra

pandas: Python data science tools (Series and dataframes)

scipy (SCIentific Python): many numerical routines

matplotlib: (PLOTting LIBrary) creating plots of data

**Basic Pros and cons of the various IDEs along with performance findings for the code**

|  |  |  |
| --- | --- | --- |
| **Application** | **Interactive debugging** | **Graphing** |
| Anaconda (Spyder) | Functional, not the best | Supports both inline and pop out graphs, can open multiple graphs at one time |
| Komodo | could not get to work | Does not natively support inline graphs, pop up graphs failed until updating to version 11.1.1-91089 f  Can only open one graph at a time |
| Visual Studio Code | This debugger was the easiest to use | Does not natively support inline graphs.  Can only open one graph at a time |

**Note 1**: I probably could have managed to get debugging to work in Komodo, but chose to just use Visual Studio code, since it worked so well. (Also, I found the following site on debugging Komodo, but still had problems: <http://docs.activestate.com/komodo/11/manual/debugpython.html>

**Note 2**: For Visual Studio Code, I found an article that seems to support generating graphs inline.

<https://stackoverflow.com/questions/49992300/how-to-show-graph-in-visual-studio-code-itself>

I did not test this since the functionality worked for Anaconda right out of the box.

**Performance**:

For the performance test I created class that I called *timestamp* that simplified logging the times the program took to execute. The function *getPerformance* performed all the logging. This function basically played 40 different batches of 1000 matches each. (40,000 matches total.) I limited each game to a maximum of 5000 rounds. The 5000 round limits really were only relevant when the program was not re-shuffling the discard piles. To keep the games interesting I set the number of players to range between 2 and 11 players.

1. Apply cheats before splitting the deck. (This would ensure player one always had the four aces.)
2. Apply cheats before after splitting the deck. (This would cause the batches with 6, 8, 10 and 11 players to have the potential of an ace or to not be available to assign to the player.
3. Same as option 1 without re-shuffling the player discards. The unexpected result here was that player 1 only was victorious around 52 percent of the time. IE. having the 4 aces did not yield that big an advantage when only two players were playing the game.
4. Same as option 2 without re-shuffling the player discards. This had similar statistical results for the 2-player game.

|  |  |  |
| --- | --- | --- |
| **Application** | **Performance (with processes)** | **Performance (after reboot)** |
| Anaconda (Spyder) | 5:48.353255 | 4:33.892924 |
| Komodo | 6:43.223487 (16% slower) | 5:35.277245 (22% slower) |
| Visual Studio Code | 6:01.775134 (3% slower) | 5:33.467782 (22% slower) |

Note 1: to make this test more accurate, I did two runs, one with 11 other processes running (the Chrome browser open: to the 9 links at the bottom of the article, this document, and the excel sheet discussed in this article), and one after a fresh reboot of the computer. These times are in minutes.

Note 2: I did not generate graphs for the performance tests because only Anaconda would not wait for the me close each graph as it was generated.

The main take-aways here are:

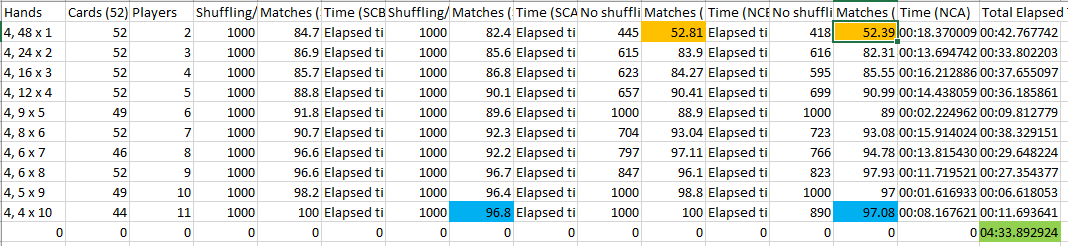
1. Anaconda performed quicker than the other tools. In fact, it was nearly as quick as the other two environments with other processes running.
2. Komodo was the most affected environment when other processes were running in the background. Otherwise its performance was nearly identical to Visual Studio Code.

The file that is generated by the program is called: War\_Stats\_Total.csv

To limit the number of columns, and since player 1 is the player that always has the advantage, I only saved the first players percentage of wins.

This file will log the percentage of times the player one won each 1000 card match up. You will also notice that typically there was less than 1000 games played when we did not shuffle the discard piles. As mentioned above this is due to limit of 5000 rounds being played per match.

Here is a screen shot of the file. Some interesting points to highlight here are:



1. When not reshuffling the discard pile and only two players were playing, the odds did not improve that much for the player with the four aces (orange highlight)
2. When elven players played, player number 1 would always win except when the aces were assigned to the player after shuffling. This happened because in this scenario, we only played with 44 cards, randomly discarding the rest. If the aces are assigned after splitting the deck, there is a good chance that 1 or more aces might not be present to assign to player 1. Hence, player 1 would need to play with 1 or more random cards. (blue highlight)
3. The last row has the total time to execute all the matches. (green highlight)
4. **Comments about the *dataFrame* / *Series* and graphing**
5. **Overview of initial findings for using the new data types**

The basic functionality of the class is to simplify statistically analyzing rounds of the game of war under various scenarios. The class will save some statistical data between runs for later evaluation.

This data includes: current match, the total number of rounds played, final player that won, the round in which each player lost the game. The player that won this round will have a zero for the round.

For example, if 3 players played the game, and player 1 won the game after 75 rounds, and players 2, 3, and 4 lost the game after 25 and 50 and 75 rounds respectively the dataFrame would look as follows:

|  |  |  |  |  |  |  |
| --- | --- | --- | --- | --- | --- | --- |
| Match | Rounds | Winner | Player 1 | Player 2 | Player 3 | Player 4 |
| 1 | 75 | 1 | 0 | 25 | 50 | 75 |

For this program I use two additional data types: *dataFrames*, and *defaultdict*.

Both types greatly expand on the features of Python.

*defaultdict*: simplifies inserting entries into a dictionary by automatically adding a default value instead of generating an error if the value does not exist.

For information on *defaultdict* watch the following video:

<https://www.youtube.com/watch?v=OSGv2VnC0go&t=1628s> (At about minute 25 he goes into using *defaultdict*. However, the whole video is pretty good so if you are starting out with Python, you might just want to watch and take notes. I did…)

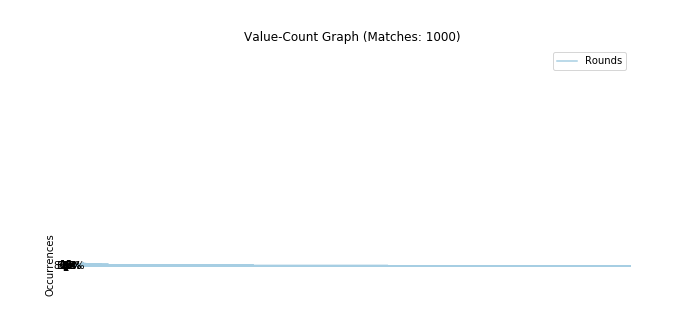
*dataFrame*: allow for more advanced storage features that provide more robust tools for data manipulation. These *dataFrames* can then be graphed. Either using *matPlotLab* or directly from the *dataFrame* class (which essentially uses *matPlotLab*). This program provides an example of both techniques. Okay, I extract a single column from the *dataFrame*, which is called a *Series*, and graph that.

The first graph, a line graph was generated directly from an element from the *dataFrame* object called a *Series*. A *Series* is single column of data pulled from a *dataFrame*

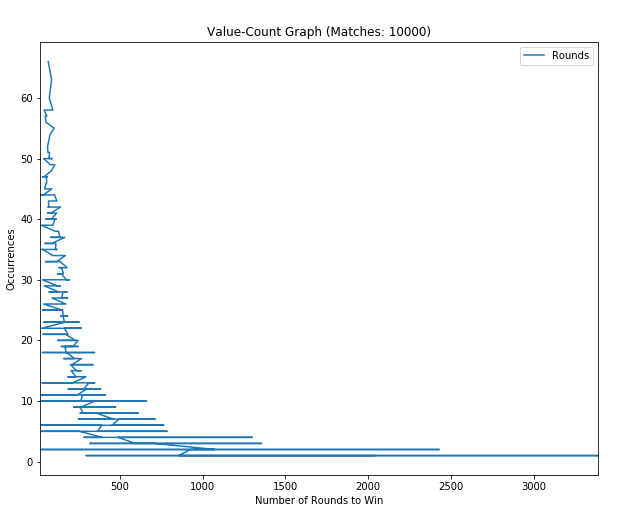
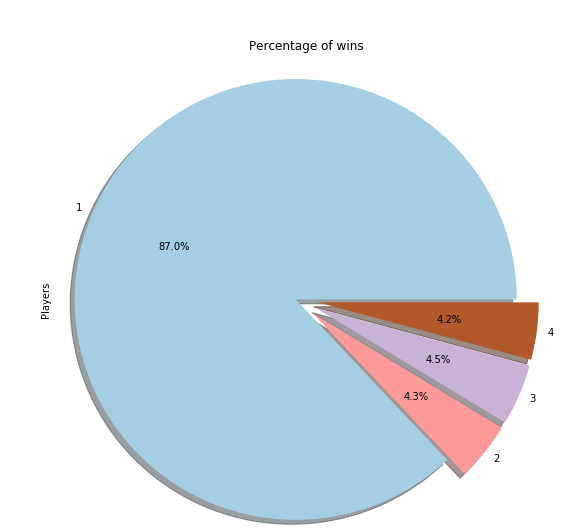
**Note:** When generating a graph from a *Series* object, it is a good idea to execute the command:

*plt.figure()* before generating a graph from the Series object, otherwise, a pre-existing plot will be used if it is still active. Please see the comments and code at lines 668 – 676.

This will lead to unexpected results as depicted below:



Instead of the graphs:



The root cause of this is that a new graph is not generated when a *Series* object type is graphed. However, the command *plt.figure*(…) will force the generation of a new graph. It should be noted that a *dataFrame* does generate a new graph.

The four graphs that are generated in the program are a line graph, scatter graph, 3d scatter graph, and a pie graph.

1. A value-count graph comparing the number of rounds to win a game to the number of occurrences.
2. A scatter plot breaking the wins for each player and apply a kernel density estimate to color shift the colors for each player to more clearly display the number of rounds it took to win.
3. A 3d scatter plot comparing the number of rounds to win a game to the number of occurrences broken out by players.
4. A pie graph that displays the percentage of wins each player had.

Some special notes about the graphing. I tried to use the same color scheme for both the 3d graph and pie graph. I found that the colors when printed using the 3d graph appeared too faded when further back in perspective within the graph.

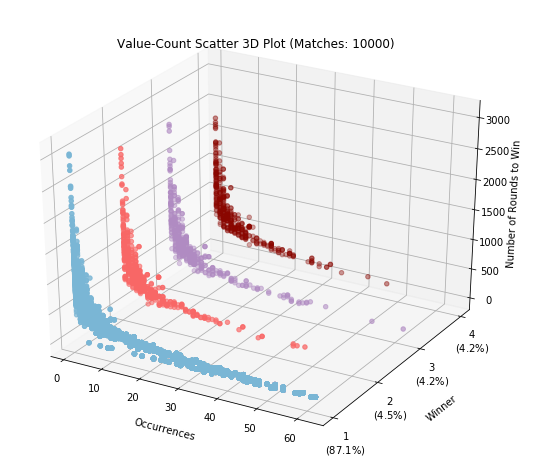
I created a function called: *\_darken* to make the colors a little more vibrant. Please see URL 2 below for where I pulled the idea for this.

I created a function called: \_makeColors to apply a specific color schema across all the values within a graph. I used similar code from the web as a basis for this code. Sadly, my computer ran into an issue and I was unable to find the specific URL again. ☹

1. **Overview of initial findings for using the new data types**

I really liked learning about and using pandas and matPlotLab. These classes have great features for manipulating data.

These classes are perfect for viewing data quickly and efficiently. The ability to generate graphs, especially 3d ones, that can be rotated and viewed from different perspectives with just a few lines of code is amazing.



For simple data manipulation pandas works fine. For example: joining two dataFrames df and dc the following code can be used:

*dn = df.merge(vc, left\_on="Rounds", right\_on="Rounds", how = "inner")*

Although, when the field names are the same it can simplify to…

*dn = df.merge(vc, on="Rounds", how = "inner")*

The equivalent SQL statement would be…

*Select \* into dn from df inner join dc on dc.Rounds=df.Rounds*

However, from the example above, it might not be clear why pandas is not quite so efficient for analyzing more complex scenarios. This is where SQL shines and allows for:

1. I found no way to limit the fields in the result set in pandas. (short of executing another command after the merge statement; which, defeats the purpose if we were trying to improve performance on a large data set.) In SQL this is performed as follows:

*Select Counts, df.Rounds, Players into dn from df inner join dc on dc.Rounds=df.Rounds*

I do not need to qualify (prepend the table name) to unique field names.

1. If we want to merge multiple tables simultaneously, which I find I need to do frequently, the merge statement can become unwieldy rather quickly. With SQL it is simply a matter of adding additional joins to the statement. Please see below.

*Select Counts, df.Rounds, Players, color into dn from (df inner join dc on dc.Rounds=df.Rounds) Inner join colors cl on dc.Players=cl.Players*

Note: The parenthesis are optional, as is assigning an alias to a table.

It should be noted that since python is fully extensible a class could be written to support these features, and maybe one already has.

1. **Notes on Anaconda and Spyder**

Since I found Anaconda and Spyder to work the best for data analysis, except when it comes to debugging, I spent the most time here trying to make sure I had the latest updates.

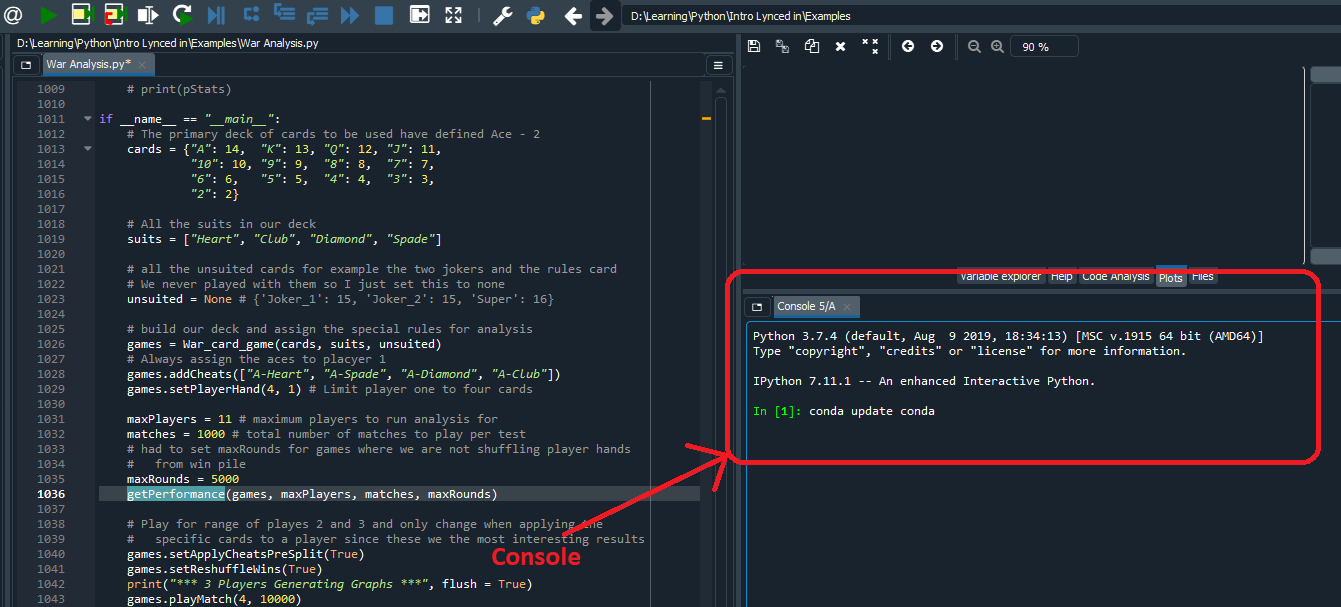
First, I wanted to make sure tools I was using were updated to the latest version.

From the console execute each of these commands. Please see below.

*conda update conda* 🡪 (first update the tool used for making updates.)

*conda update anaconda 🡪* (Second update Anaconda, the main application.)

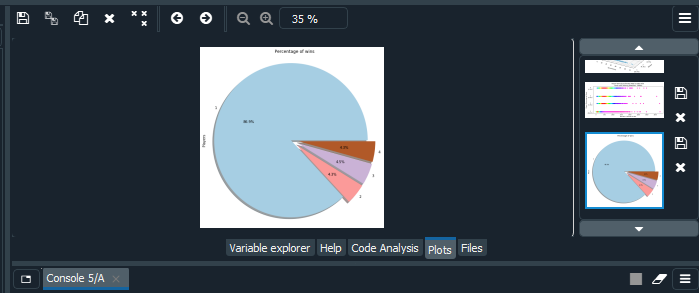
conda update spyder 🡪 (Third update Spyder.)



Make sure to reset the application to make the changes effective, best way to do this is to completely close Anaconda between each update.

Second, Anaconda is running IPython which yields greater control over graphing.

By default, Anaconda will generate the graphs in the window above the console.

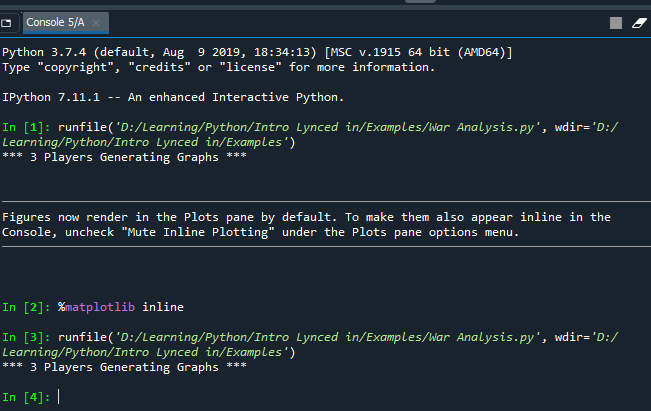


There is great flexibility to control where the plots are displayed.

If graphs are not already generating in this manner, enter the command:

%matplotlib inline

At the console



To have anaconda generate pop out graphs that can be manipulated, rotated, zoomed, etc.

At the console enter the command:

*%matplotlib qt5*

Note: you can do this programmatically in your code as well by using the command:

*matplotlib.use('Qt5Agg')*

I do not typically like doing this, unless you are generating a window with controls on it since this prevents the user from determining where their graph goes. The graph will close immediately on completion of the python code. This will mandate developing a full functional window in Python.

If you are interested in looking into this, here is a great URL to get started:

<https://matplotlib.org/gallery/user_interfaces/embedding_in_qt5_sgskip.html>

1. **Using the class for war for analysis**

The options that are permitted for this program when instantiating this class are:

1. cards: A dictionary containing a list of the cards and their corresponding values. For example {“A”: 14, “K” 13, …} will add the cards specified for each suit to your deck.
2. suits: A List that contains all the suits that will be in our deck. For example [“Spade”, “Club”… ]
3. unsuited: A dictionary of all the unsuited cards in your deck and their corresponding values. For example {“Joker 1”: 15, “Joker 2”: 15} would add two jokers to the deck.
4. reshuffleWins: This is a true/false flag that determines if we will shuffle the discard pile of cards before bringing them back into play. Since the rules are not specific here, this is an optional parameter.
5. applyCheatsPreSplit: This flag determines when the cheats if defined will be applied. The reason this is important is when the number of players specified in a deck will cause some of the cards to be excluded from the game. If the cards being excluded in a particular game are unavailable, then they will not be assigned to the specified player.

Additional options that can be updated are:

1. addCheats: This function takes a list or a single car as the first parm. The cards specified in this call must exist in the deck defined above. If now player is specified player one will be assumed to be the player that will be assigned this/these cards. A card can only be assigned to one player. The last player assigned to the card will take precedence.
2. setPlayerSize: this will force a player’s hand to be a specific size.

In the code below:

*games.addCheats(["A-Heart", "A-Spade", "A-Diamond", "A-Club"])*

*games.setPlayerHand(4, 1)*

Player one is given the four aces and then set to always contain only four cards.

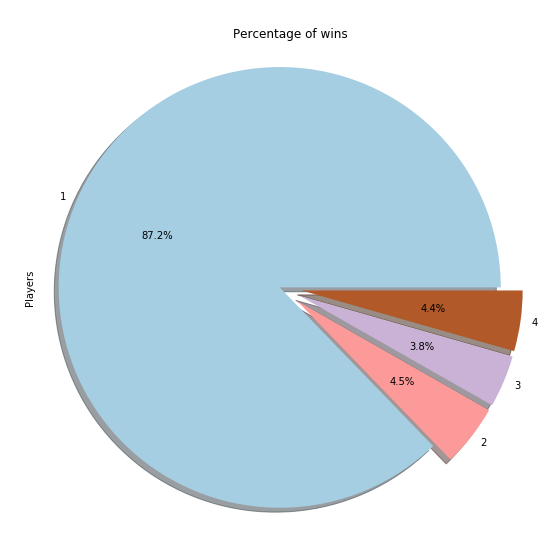
The code is well documented; hence, the reason for being more than 1000 lines in length.

It is available on GitHub, here is a link:

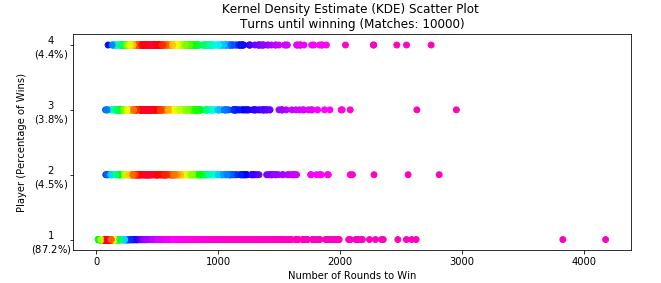
<https://github.com/RickOldenburger/War-Card-Game-Analysis>

1. **Conclusions**

In conclusion I found that we should have won occasionally. Not sure why never remember anyone ever winning but my brother.



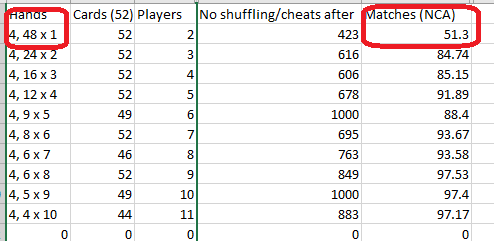
Perhaps we just gave up; since statistically, the game would last a lot longer if any of us were destined to win that match. The kernel density estimate depicts the more frequent number of rounds for each player in red. On average, it would have taken us about 500 rounds to win, compared to my brothers average of 100 rounds to attain his victory.



One last point of interest is a result I did not expect. This occurs when:

1. The discard piles are NOT re-shuffled.
2. There are only two players (Player 1 has four aces, and player two has the other 48 cards.)
3. Play is abandoned after 5000 rounds (hence only 423 games were played

As depicted below… Player 1 only won about 51% – 52% of the time (51.3%). This translates to the fact that player 1 really does not have a significant advantage over player 2.



**Sites used for developing this program:**

1. Video on Transforming Code into Beautiful, Idiomatic Python (I learned about *defaultdict* here)

<https://www.youtube.com/watch?v=OSGv2VnC0go&t=1628s>

1. Found code to partially code darken function: (However, it did not work as depicted so I had to modify the code heavily… but the idea was still presented here.)

<https://stackoverflow.com/questions/37765197/darken-or-lighten-a-color-in-matplotlib>

1. Found similar code for altering the colors in the scatter plot based on density:

<https://inneka.com/programming/python/visualization-of-scatter-plots-with-overlapping-points-in-matplotlib/>

1. Found information on using value\_count() from pandas:

<https://www.geeksforgeeks.org/python-pandas-index-value_counts/>

1. Site used for general python graphing examples:

<http://queirozf.com/entries/pandas-dataframe-plot-examples-with-matplotlib-pyplot>

1. Information for creating a pie chart:

<https://matplotlib.org/3.1.1/gallery/pie_and_polar_charts/pie_features.html>

1. More advanced string use, it basically showed me how to use raw formatting for files.

<https://www.python.org/dev/peps/pep-0498/>

1. Evaluating execution time for both the time import and timedelta call.

<https://stackoverflow.com/questions/27779677/how-to-format-elapsed-time-from-seconds-to-hours-minutes-seconds-and-milliseco>

1. Link to an example for generating an external window.

Notes: I did not use this example in my code, Jupyter is required, so it works best to use with Anaconda.

<https://matplotlib.org/gallery/user_interfaces/embedding_in_qt5_sgskip.html>