Project Description: Analyzing Baseball Data

Data Science is a rapidly growing discipline in which large amounts of data are analyzed to extract knowledge and insight from that data. That insight can be used to better explain the past, predict the future, or otherwise make decisions based on data rather than intuition. In this project, we will introduce you to some of the basic tools of data analysis. We will do some basic analyses on Baseball statistics. Large amounts of data on baseball is readily available, making it an ideal topic to explore the ideas behind large scale data analysis. While the particular analyses you will perform are specific to baseball, the underlying ideas and strategies for analyzing data are not.

The first project in this course required you to develop code for reading and writing CSV files using dictionaries. For this project, we will provide you with a several CSV files that contain data on the performance of Major League Baseball (MLB) player over a span of more than a century. You will build upon the work you did in the previous project to statistically analyze this data. This historical baseball data can be found at seanlehman.com in his baseball archive. The archive includes the raw data (stored in CSV files) used in computing most important baseball statistics.

This zip file includes a collection of CSV files from this archive with data that spans the years 1871-2016. The zip files includes two CSV files "Master.csv" and "Batting.csv" that contain player information and batting statistics. Since this data is being updated regularly, we ask that you use the 2016 versions of this two files linked here: Master_2016.csv and Batting_2016.csv. Using our provided version of the files allows us all to work from the same raw data.

Each line in the file Master.csv (and Master_2016.csv) is indexed by a unique field, `"playerID"`, that corresponds to each player that has played in Major League Baseball. Other fields in the file include the player's first and last names. The file Batting.csv (and Batting_2016.csv) includes season-by-season batting data for each player. The first field identifies the player via his ID while the rightmost fields contain integers that correspond to the player's performance in various basic statistical categories.

This project will focus on writing code that will compute several common batting statistics from the data in these CSV files.

Preliminaries: Working on the Project

Coding Style

In this class, you will be asked to strictly follow a set of coding style guidelines. Good programmers not only get their code to work, but they also write it in a way that enables others to easily read and understand their code. Please read the style guidelines carefully and get into the habit of following them right from the start. A portion of your grade on the project will be based upon coding style.

Testing

You should always test your code as you write it. Do not try to solve the entire project before running it! If you do this, you will have lots of errors that interact in unexpected ways making your program very hard to debug. Instead, as you write each function, make sure you test it to ensure that it is working properly before moving on to the next function.

Throughout this course, we will be using a machine grader (OwlTest) to help you assess your code. You can submit your code to this Owltest page to receive a preliminary grade and feedback on your project. The OwlTest page has a pale yellow background and does **not** submit your project to Coursera. OwlTest is just meant to allow you to test your project automatically. Note that trying to debug your project using the tests in OwlTest can be very tedious since they are slow and give limited feedback. Instead, we *strongly* suggest that you first test your program using your own tests. Also, note that each OwlTest link is specific to a particular project. You need to come back to this page and click the link above to ensure that you are running the tests for this project.

When you are ready to submit your code to be graded formally, submit your code to the assignment page for this project. You will be prompted to open a tool which will take you to the Coursera LTITest page. Note that the Coursera LTITest page looks similar to the OwlTest page, but they are *not* the same! The CourseraLTI Test page has a **white** background and does submit your grade to Coursera.Provided code

Project: Analyzing Baseball Data

We have provided the following template that you can use to get you started on the project. It includes the signatures (name, parameters, and docstrings) for all of the functions that you will need to write. The code however, simply returns some arbitrary value no matter what the inputs are, so you will need to modify the body of the function to work correctly. You should not change the signature of any of the functions in the template, but you may add any code that you need to. The provided code also includes implementations of the functions `read_csv_as_list_dict()` and `read_csv_as_nested_dict()` from the first project, as you will need to use these functions in this project. You can also download all of the files used by OwlTest when testing your code as a zip file.

Working with the CSV files

As the format of the CSV files that store the baseball data could change (or you could acquire data from somewhere else), the functions that operate directly on the data will all take an "info" dictionary that provides information about the files. That way, you do not need to use constants within your code to access the CSV files and their columns. If the name of a particular column changes, for instance, you can simply update the info structure appropriately and all of your code will continue to work. Furthermore, if you have a CSV file that uses different field separators, you can tailor the info structure appropriately to deal with that without needing to change any of your other code. The info dictionaries contain the following keys, all of which are strings (the use of these keys will become apparent as you work on the project, you may want to refer back to this information as you work on the different parts of the project):

- "masterfile": the name of the master CSV file that includes columns with player IDs and names.
- "battingfile": the name of the CSV file that includes columns with player IDs and batting data.
- "separator": the delimiter character used in the two CSV files.
- "quote": the quote character used in the two CSV files.
- "playerid": the name of the column header for player IDs in both the master and batting CSV files.
- "firstname": the name of the column header for player's first names in the master CSV file.
- "lastname": the name of the column header for player's last names in the master CSV file.
- "yearid": the name of the column header for the year in the batting CSV file.
- "atbats": the name of the column header for at-bats data in the batting CSV file.
- "hits": the name of the column header for hits data in the batting CSV file.
- "doubles": the name of the column header for doubles data in the batting CSV file.
- "triples": the name of the column header for triples data in the batting CSV file.
- "homeruns": the name of the column header for home runs data in the batting CSV file.
- "walks": the name of the column header for walks data in the batting CSV file.
- "battingfields": a list of column header names that correspond to batting data in the batting CSV file.

If you look in the template file, you will see an example of such an "info" dictionary that is used to access the baseball data archive discussed above. It looks as follows:

```
    baseballdatainfo = {
        "masterfile": "Master_2016.csv",   # Name of Master CSV file
        "battingfile": "Batting_2016.csv", # Name of Batting CSV file
        "separator": ",",                  # Separator character in CSV files
        "quote": '"',                      # Quote character in CSV files
        "playerid": "playerID",            # Player ID field name
        "firstname": "nameFirst",          # First name field name
        "lastname": "nameLast",            # Last name field name
        "yearid": "yearID",                # Year field name
        "atbats": "AB",                    # At bats field name
        "hits": "H",                       # Hits field name
        "doubles": "2B",                   # Doubles field name
        "triples": "3B",                   # Triples field name
        "homeruns": "HR",                  # Home runs field name
        "walks": "BB",                     # Walks field name
        "battingfields": ["AB", "H", "2B", "3B", "HR", "BB"]
    }
```

As you can see, if you wanted to access the master CSV file, you would therefore open the file named `baseballdatainfo["masterfile"]`. Once you have read the CSV file, you could access the column containing player's first names using the key `baseballdatainfo["firstname"]`. If you look at the batting statistics formulas (discussed below) in the template, you can also see how they use the "info" dictionary in order to access particular batting statistics.

Provided baseball statistical functions

We have also provided implementations of functions that compute three important statistics from the batting data. These functions are

- `batting_average(baseball_info, batting_stats)` - Takes dictionary with batting statistics and computes the player's batting average.
- `onbase_percentage(baseball_info, batting_stats)` - Takes dictionary with batting statistics and computes the player's on-base percentage.
- `slugging_percentage(baseball_info, batting_stats)` - Takes dictionary with batting statistics and computes the player's slugging percentage.

Note that if the player has fewer than 500 batting attempts (at-bats), each function returns zero to eliminate statistical outliers with a small number of at-bats. You can also create additional statistical functions using `lambda`. For instance, you could build a `lambda` function that computes a player's on-base plus slugging percentage (OPS) by adding together the results of calling `onbase_percentage` and `slugging_percentage`.

Part 1 - Compute players with top batting statistics by year

Your task for this part of the project will be to write four functions that can used in combination to compute the top players based on a provided statistical formula for a given year. These functions will select a subset of the data and compute the provided statistic on this data.

First, you will write `filter_by_year`. This function should filter a list of batting statistics dictionaries to return a new list of batting statistics dictionaries that consist only of those statistics in the input which correspond to the given year. Each batting statistics dictionary in the input corresponds to the statistics for a single player for a single year. A batting statistics dictionary is a dictionary whose keys (all strings) include a player id, a year, various batting statistics, and possibly other information. As you only need the name of the "year" column for this function, it is given as an input (`yearid`). This function should not modify the batting statistics dictionaries in any way, rather it should simply return a list that is similar to the input list of statistics, only it is potentially smaller (assuming that the input contains statistics from multiple years). The `filter_by_year` function has the following signature:

```
def filter_by_year(statistics, year, yearid):
    """
    Inputs:
      statistics - List of batting statistics dictionaries
      year       - Year to filter by
      yearid     - Year ID field in statistics
    Outputs:
      Returns a list of batting statistics dictionaries that
      are from the input year.
    """
```

Hints:

1. The year will be passed to this function as an integer, such as `1999`. Note, however, that when you read the baseball data from the CSV file, the year will be read as a string, such as `"1999"`. Make sure that you convert these to the same type before comparing them in order to do the filtering.

Next, you will write `top_player_ids`. This function should compute the top `numplayers` players with the given compound statistic computed by `formula`. The input `formula` function will return a floating point number corresponding to the compound statistic for the given input batting statistics dictionary. The batting statistics are again given as a list of batting statistics dictionaries of the same format that was used by `filter_by_year`. You will need to pass the baseball `info` dictionary to the statistics formula so that it can access the appropriate data out of the batting statistics dictionaries. The `top_player_ids` function should return a list of tuples, where the first element of the tuple is a player ID and the second element is the statistic for that player computed by `formula`. This list should be sorted in descending order based upon the value of the computed statistic.

Note, that in general, there could be ties whereby two players have exactly the same value for the computed statistic. In general, you would have to decide what to do in that case. For baseball statistics, returning the tied players in any order is probably not a problem. However, if you are computing the top 10 players and the 10th and 11th player are tied, you would arguably want to return a list with both of them in it as tied for 10th place. For the purposes of this project, however, we are going to ignore ties and the machine grader (OwlTest) will not test any situations in which there are ties. In fact, the values of the computed statistic will always be different by at least 0.00001. So, you do not need to write any code to deal with the case where there is a tie. Just keep in mind that if you are doing similar types of analyses in the future, you should think about what you want to do if there are ties.

The `top_player_ids` function has the following signature:

```
def top_player_ids(info, statistics, formula, numplayers):
    """
    Inputs:
      info       - Baseball data information dictionary
      statistics - List of batting statistics dictionaries
      formula    - function that takes an info dictionary and a
                   batting statistics dictionary as input and
                   computes a compound statistic
      numplayers - Number of top players to return
    Outputs:
      Returns a list of tuples, player ID and compound statistic
      computed by formula, of the top numplayers players sorted in
      decreasing order of the computed statistic.
    """
```

Hints:

1. You should first create a list of all player IDs and their computed statistic, by calling formula for every player in the `statistics` list.
2. You can then sort this resulting list appropriately and select the top `numplayers` items from the list to return.

Next, you will write `lookup_player_names`. This function should take a list of tuples in the same form that is returned from `top_player_ids`. From that information, this function should create a list of strings of the form "x.xxx --- FirstName LastName". For example, "0.325 --- Scott Rixner". The floating point statistics must be converted to a consistent format with one digit before and three digits after the decimal point. The `lookup_player_names` function has the following signature:

```
def lookup_player_names(info, top_ids_and_stats):
    """
    Inputs:
      info              - Baseball data information dictionary
      top_ids_and_stats - list of tuples containing player IDs and
                          computed statistics
    Outputs:
      List of strings of the form "x.xxx --- FirstName LastName",
      where "x.xxx" is a string conversion of the float stat in
      the input and "FirstName LastName" is the name of the player
      corresponding to the player ID in the input.
    """
```

Hints:

1. The `info` dictionary contains all of the information you need to access the appropriate CSV file(s) to convert player IDs into player names.

2. You may want to review the string processing and formatting ideas from our previous class and/or review the Python documentation on strings. In particular, make sure you use the f type to format floating point numbers.

Finally, you will write **compute_top_stats_year**. This function takes a baseball **info** dictionary, a compound statistics **formula**, the number of top players to find, **numplayers**, and a **year**. It should use that information to return a list of strings of the same form as returned by **lookup_player_names** that correspond to the **numplayers** players from the given year with the highest compound statistic computed by **formula**. The **compute_top_stats_year** function has the following signature:

```
def compute_top_stats_year(info, formula, numplayers, year):
    """
    Inputs:
      info        - Baseball data information dictionary
      formula     - function that takes an info dictionary and a
                    batting statistics dictionary as input and
                    computes a compound statistic
      numplayers  - Number of top players to return
      year        - Year to compute top statistics for
    Outputs:
      Returns a list of strings for the top numplayers in the given year
      according to the given formula.
    """
```

Hints:

1. This function should make use of the previous functions you wrote and the provided functions. There should not be a lot of code in this function, rather you are just putting together what you have already done.
2. Notice there are no statistics given as input to this function. You need to read them from the appropriate CSV file(s). Make sure you read them in the format that is needed by the functions you have already written.

Part 2 - Compute players with top batting statistics by career

Your task for this part of the project will be to write two more functions that can used along with the other four functions to compute the top players based on a provided statistical formula for their entire career. These functions will aggregate the yearly data in data that spans and player's career and then compute the provided statistic on this data.

First, you will write **aggregate_by_player_id**. The batting statistics are again given as a list of batting statistics dictionaries of the same format that was used by the functions in Part 1. The column name for the player ID field is given as **playerid**, and the **fields** input is a list of the names of the columns that should be aggregated. Note that the **fields** input is necessary because not all of the batting statistics can (or should) be aggregated. For example, it doesn't make sense to sum up the years of the statistics! This function should produce a dictionary of dictionaries. The outer dictionary should map player IDs to batting statistics dictionaries. The batting statistics dictionaries should have keys for **playerid** and all of the field names in **fields**, all mapped to the appropriate values. The batting statistics should be the sum of all of the statistics within the **statistics** input that correspond to each **playerid**. So, for example, if the input contains data for two years of a particular player, the output should contain one entry for that player with the statistics that are the sum of those two years. The **aggregate_by_player_id** function has the following signature:

```
def aggregate_by_player_id(statistics, playerid, fields):
    """
    Inputs:
      statistics - List of batting statistics dictionaries
      playerid   - Player ID field name
      fields     - List of fields to aggregate
    Output:
      Returns a nested dictionary whose keys are player IDs and whose values
      are dictionaries of aggregated stats.  Only the fields from the fields
      input will be aggregated in the aggregated stats dictionaries.
    """
```

Hints:

1. The output format is a dictionary with keys that are player IDs to simplify the processing required within this function. The first time you see a particular player, you will need to create a new entry in the dictionary. Thereafter, you can simply update the statistics that are already there.

2. Be careful about how you access and update the dictionaries in this function. There are a lot of dictionaries and it can be difficult to keep everything straight.

Finally, you will write `compute_top_stats_career`. This function is very similar to `compute_top_stats_year`, but you will first need to aggregate the statistics for each player so that you are operating on career statistics, instead of statistics for a particular year. It should return a list of strings of the same form as returned by `lookup_player_names` that correspond to the `numplayers` players with the highest compound statistic computed by `formula` for their careers. The `compute_top_stats_career` function has the following signature:

```
def compute_top_stats_career(info, formula, numplayers):
    """
    Inputs:
      info        - Baseball data information dictionary
      formula     - function that takes an info dictionary and a
                    batting statistics dictionary as input and
                    computes a compound statistic
      numplayers  - Number of top players to return
    """
```

Hints:

1. This function should make use of the previous functions you wrote and the provided functions. There should not be a lot of code in this function, rather you are just putting together what you have already done.
2. Notice there are no statistics given as input to this function. You need to read them from the appropriate CSV file(s). Make sure you read them in the format that is needed by the functions you have already written.
3. Notice that the `aggregate_by_player_id` returns statistics in a different structure than is used by the other functions in this project. You will need to convert the data into the appropriate format before passing it to other functions.

Testing your code

Notice that the provided template includes a `test_baseball_statistics` function at the end. This code calls the functions you have written to compute top players based on various statistics. However, this code only works once you have written all of the functions and it operates on the full baseball data set. We strongly recommend you write smaller tests and utilize OwlTest to test each function individually. If something goes wrong, you will likely want to write smaller tests to help you understand how your code is working anyway. OwlTest uses smaller files to allow more targeted and understandable testing. You can use those files on your own, as well.

Mark as completed