**Project Inception**

Several months ago, as a first attempt at a programming project, I cobbled together a database file containing MLB results and statistics over the last twenty seasons. I accomplished this by scraping sportsreference.com, which has a generous web-scraping policy that accommodates most get requests. Given the massive amount of information I needed to scrape, however, even sportsreference’s web-scraping policy presented difficulties I struggled to overcome. Trust me, it is difficult to capture data in a timely manner capped at a request per second! Fortunately, I found a more complete, open-source set of resources – retrosheet.com.

Founded in 1989, retrosheet is a record keeping project that computerizes play-by-play accounts of Major League baseball games. Information on the site is free to use, under the condition that any project developed with the site's information properly cites the site.

Although retrosheet makes staggering amounts of information available free of charge, I quickly encountered a problem – the information largely exists in specialty files that can only be queried by the site's software applications. Further complicating matters, the applications are meant to run on a windows command line, with output redirected to text files that can be imported into database and spreadsheet software.

**What the Heck is a Command Line?**

I suspect that, after reading the last sentence, the typical windows user has a basic question: what the heck is a command line? Informally, the command line is that vaguely intimidating, minimalist screen that the IT specialist at work uses to conduct magic rituals. More formally, it is one of two command line shells for the windows operating system that provides an interface between a user and the OS. A windows user can find and open the command line by entering “Command Prompt” in the windows search bar.

As stated above, retrosheet’s software applications – “bevent,” “bgame,” and “box” – are designed to run on the windows command line. Ultimately, this approach is cumbersome, requiring a fair amount of setup that potentially discourages the use of the applications. My solution to this problem, a python module that circumvents the command line, allows python enthusiasts to start with the fun stuff – creating descriptive statistics and exploring the actual contents of the data sets. Since my project obviates the manipulation of the command line, it is worth exploring how the command line tools function.

**Legacy Implementation**

We are getting ahead of ourselves. Before we can explore how the retrosheet applications work, we must complete some basic setup. To access the site’s information, we have to create a directory comprising the retrosheet event files we want to query, in addition to the three aforementioned applications – bevent, bgame, and box. Event files may be downloaded from <https://www.retrosheet.org/game.htm>, while the applications may be downloaded from <https://www.retrosheet.org/tools.htm>. Note that event files end with either the “.EVA” or “.EVN” suffix.

Now for some bad news: Linux and Unix users are out of luck. Developed before apple products skyrocketed in popularity, the retrosheet applications are designed to run on windows OS. Although the command line can be emulated on Linux and Unix operating systems, I have written and tested this project on windows, reserving cross platform functionality for a future date. I cannot guarantee that the project module will work on a non-windows operating system. My apologies to any mac users.

Once the event files and applications are in the same directory, we can begin testing the tools. First, we call the command line, which should open a screen similar to figure 3.1. Note that right clicking on the command prompt’s header will give us access to properties, allowing us to customize the command line to our liking.

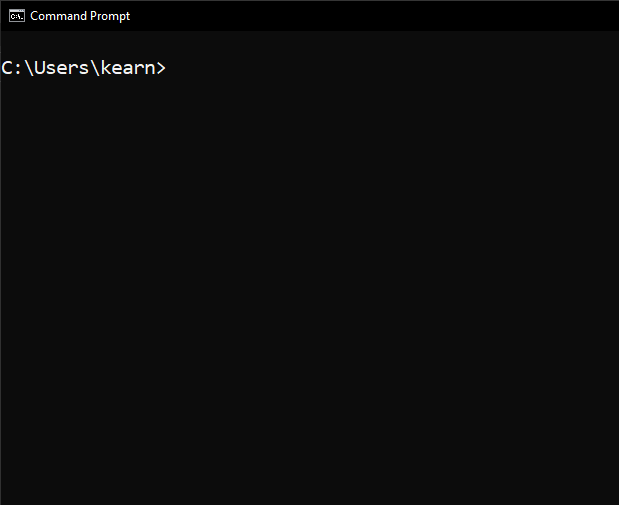


Figure 3.1 - Windows command line.

Next, using the cd command – an abbreviation for change directory – navigate to the folder where the retrosheet event files and applications are stored. Typing a retrosheet application’s name followed by the -h switch will provide a broad overview of the application.

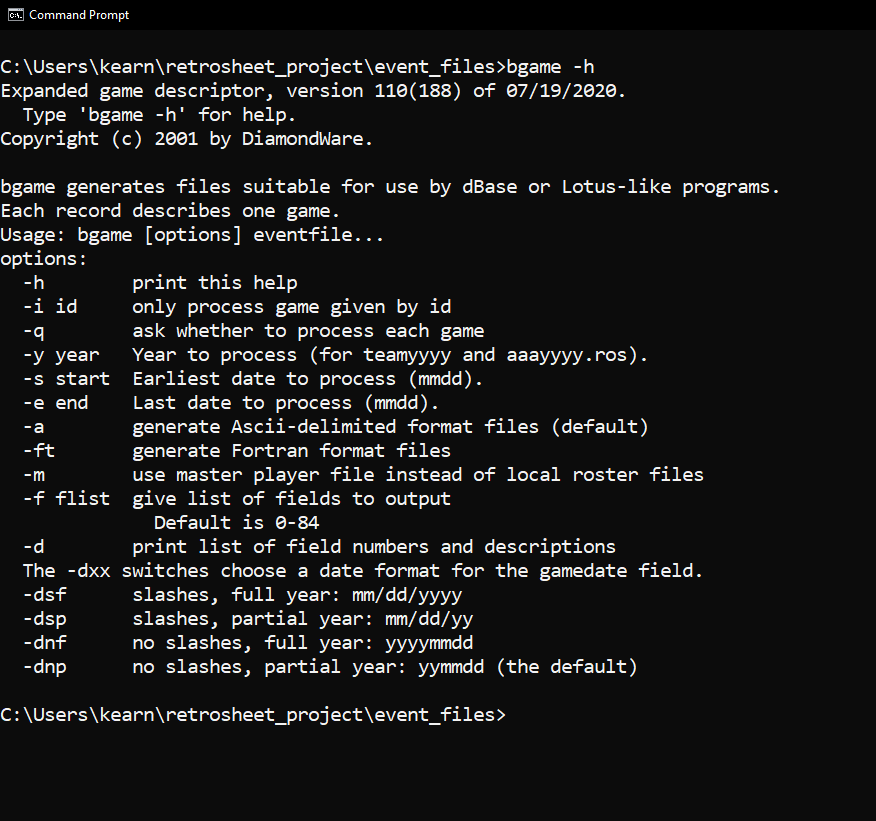


Figure 4.1 – bevent help screen.

Figure 4.1 shows the bgame help screen. It provides a brief description of the application, followed by an enumeration of the various switches that can be utilized in a query. The switches allow a user to manipulate the dates queried and the formatting of any results, and they may be combined in a variety of ways. A quick aside: is your command line too cluttered? No problem! A quick “cls” command will clear your screen.

Now that our screen is cleared, let us attempt our first query. Say we want all of the bgame fields associated with the Boston Red Sox for July 2021. As outlined by the help screen, our query will need the following information: the string “bgame”; the year to be processed, 2021; the start date, July first; the end date, July thirty-first; and the relevant event file, 2021BOS.eva. Combining these components into a single query, we get the following, “bgame -y 2021 -s 0701 -e 0731 2021BOS.eva.” Figure 5.1 shows the results.

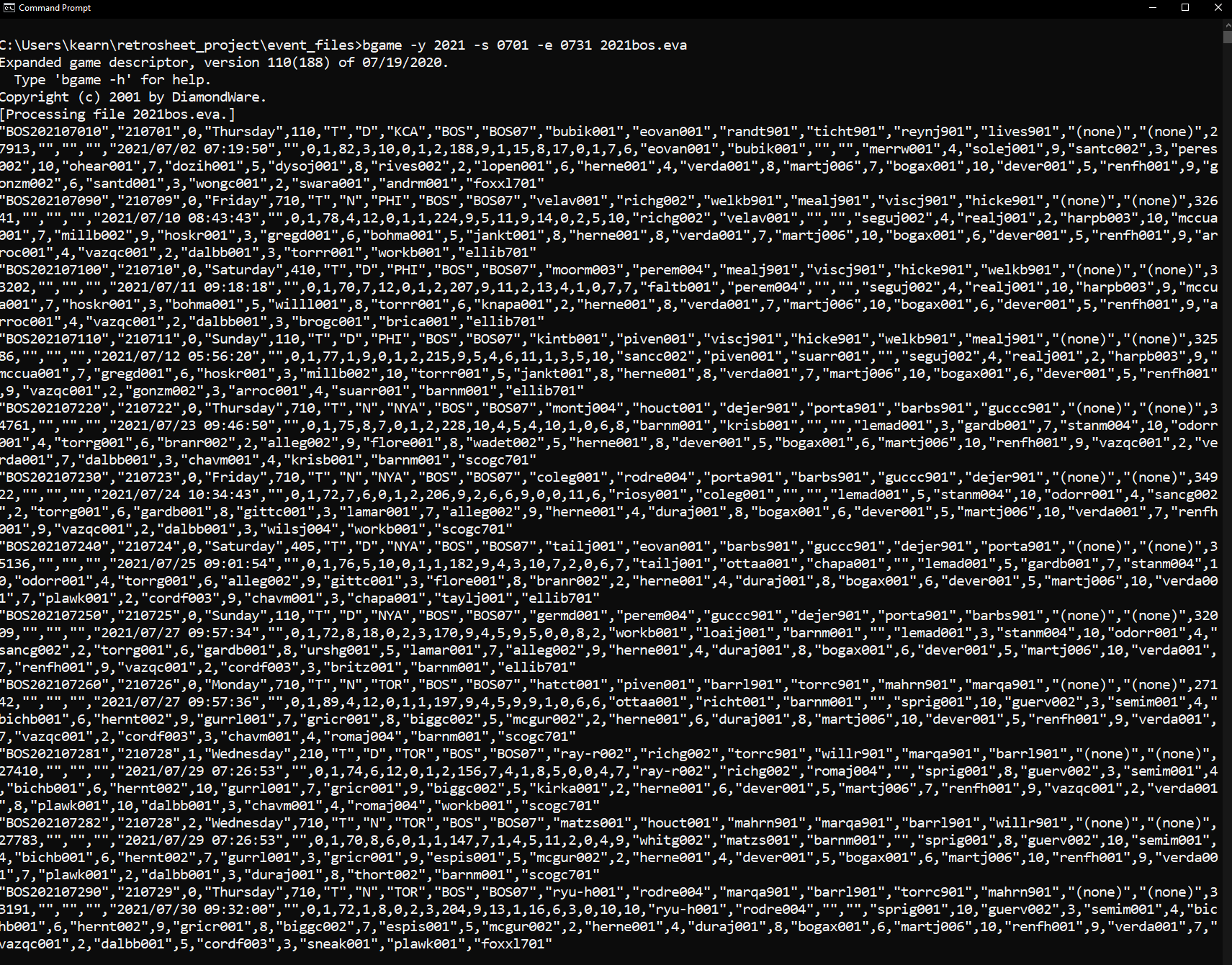


Figure 5.1 - Results from bgame query.

Notice that the dates in our query, July first and July thirty-first, are padded with zeros. Failure to pad single digit days or months with zeros will result in an empty return value from standard output. Moreover, as alluded to above, the suffix of the queried file carries meaning – event files for American League teams end with the suffix “.EVA,” while files for National League teams end with the suffix “.EVN.” Failure to provide the correct suffix will result in an empty return value, as will an incorrect team abbreviation.

The information as currently displayed is not particularly helpful. Indeed, it may induce a low-grade migraine in the unsuspecting newcomer. How can we transfer this information to a more digestible form?

Retrosheet recommends redirecting the command line’s standard output to a text or csv file. While this method may have its shortcomings, it is certainly preferable to a blob of text superimposed against a bare screen. To redirect standard output, we simply have to place the “>” symbol after our query, along with the name and location of the file we want to write. Our Boston query could be rewritten in the following manner: “bgame -y 2021 -s 0701 -e 0731 2021BOS.eva > c:\Users\kearn\retrosheet\_project\text\_files\Boston\_July\_21.csv.”

As captured by figure 6.1, a csv file has been created and placed in our target directory. Note that csv stands for comma separated values, a format that is commonly used to import data to, and export data from, relational databases.

Most queries will mirror our first example. If we wanted to extract all bgame fields up to a certain date, then we would simply drop the -s switch. For example, “bgame -y 2021 -e 0701…” will return a result set up to and including July first. Conversely, “bgame -y 2021 -s 0701…” will return a result set from July first through the end of the regular season. If no column information is specified in either a bgame or bevent query, then column defaults are provided.

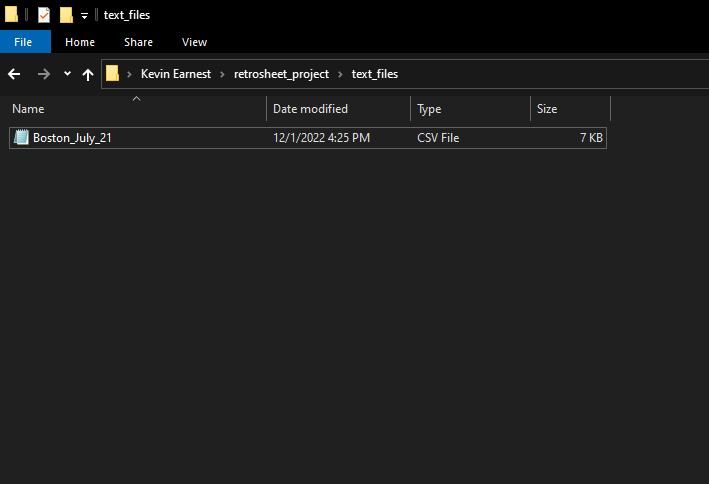


Figure 6.1 – Text file directory.

**Application Shortcomings**

This workflow has a number of limitations. Event files are organized in accordance with a team’s home games. This means that any given event file only covers half of a season. Retrieving the other eighty-one games is no trivial matter, and doing so via the command line is tedious and error prone. An algorithmic approach to this problem would seem preferable to command line mastery.

Moreover, the breadth of major league history is vast, presenting a host of problems for users. Teams change cities and even leagues, which result in new team abbreviations and file extensions for query construction. Not so long ago, the Brewers played in the American League, the Astros the National League.

Although application output can be redirected to text files, this is not a viable solution for long term data storage, nor did the application creators intend it as such. As stated earlier, the text files are meant to be imported into database software. An ordinary user, however, may have no interest in creating a MySQL or SQLite database, or even know where to begin learning SQL.

Creating an entire database of results is a substantial roadblock for casual users who may only need limited access to the data. Requesting specific fields to answer pointed questions will probably be the most efficient approach for the curious fan or fantasy league enthusiast. There is also the matter of managing the considerable amount of required supplemental information, such as roster files, field names, and field descriptions.

**Project Implementation**

I did not find the prospect of maintaining a directory of text files appealing. Accordingly, this module file came into being. This module treats retrosheet's applications as a black box, using python's run method to capture output in python objects rather than text files. In turn, these objects can be used with the Pandas data analysis library to create descriptive statistics, bypassing the setup required in the legacy implementation.

"retro\_object.py" creates a python class titled "RetroObject," which contains two types of methods: formatting methods and emulation methods. The former formats user input into an appropriate query string, while the latter uses the query string to call the executables for retrosheet's software applications.

The bevent method returns information about every individual play in a baseball game, including pitch type, runners on base, batter and hitter handedness, etc. Bgame returns general information about a game, such as weather, starting lineups, and stadium attendance. Both methods return a list of lists that can be passed into the Pandas DataFrame method. Box returns a string of box scores that can either be displayed with python's print function or stored in a text file.

All captured output is in the form of a string. As a result, RetroObject's methods are fairly involved, with a considerable amount of conditional logic. To the extent that I have deviated from python's PEP8 style guide, it is to accommodate the reader in processing these conditional code blocks.

I have attempted to keep in-file comments to a minimum. In lieu of in-file comments, I have used docstrings to convey class and method functionality to the reader. For more descriptive comments about code structure and syntax, please read the "code reference" companion file, which elucidates these matters.

**Why Use this Project?**

Retro\_object.py eliminates the query and storage steps inherent in the legacy application implementation, allowing users to immediately begin the process of creating descriptive statistics. Importantly, python and its associated libraries are open source. They are one-hundred percent, all caps FREE. While Microsoft Excel is excellent spreadsheet software, it does carry a hefty price-tag, with Microsoft incentivizing subscriptions over direct software ownership. Additionally, working entirely in Excel carries limitations, especially with larger data sets. As a general-purpose programming language, python and its associated libraries provide functionality that simply does not exist within excel. There is nothing that can be done in excel that cannot be done more effectively in python.

But that is enough praise for python and open-source projects – for now, anyway. Some examples are in order.

**July Strikeouts at Boston Home Games: A Test Case**

To demonstrate the module’s capabilities, we will obtain information about every July strikeout at Fenway during the 2021 season. We will use the python standard library and a third-party extension, “Pandas,” to conduct our research. The source code for this example can be found in the following html files, which are located in the “project\_exports” directory: “first\_notebook,” “second\_notebook,” and “third\_notebook.” The products of these exports, three similarly named “july\_strikeouts” csv files, can be found in the same directory.

As a matter of convention, import statements – statements that provide access to code written in another module – are written at the top level of a python file. We begin first\_notebook with two imports: “import pandas as pd” and “import retro\_object as ro.” The former imports the Pandas data analysis library, providing us access to an impressive set of numeric and data manipulation methods. The latter provides us access to the retro\_object.py module, which contains the RetroObject class and retrosheet peripherals. Methods and attributes associated with imports may be called through dot notation. For example, pd.DataFrame() or ro.bgame().

Now for the fun part. We have to consider which RetroObject method to call. Since we want information about events within a game, a call to bevent is in order. Our ultimate goal is to construct a Pandas DataFrame, the predominant Pandas data structure that enables the bulk of the library’s functionality. Since bevent returns a list of lists, constructing our desired DataFrame is as simple as passing a call to bevent into DataFrame’s data parameter, as captured by figure 11.1. Note that the call to bevent does not use padded zeros, and it accommodates both string and numeric input. After passing bevent as an argument into the data parameter, we pass a list comprehension as an argument into the columns parameter. This merits an explanation.

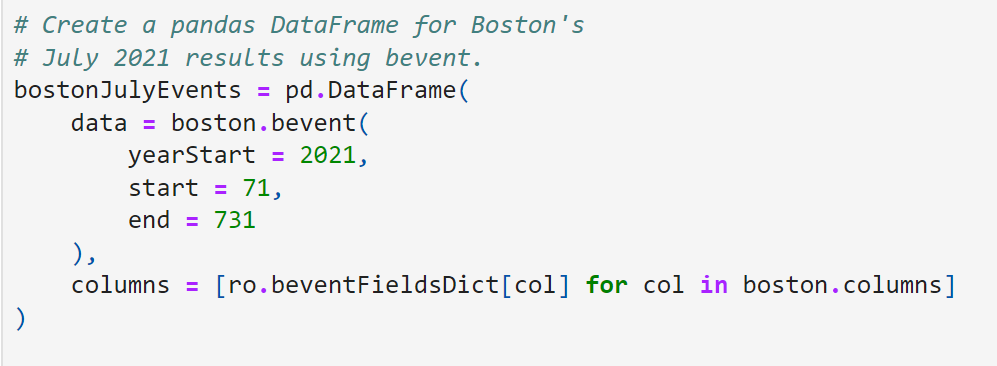


Figure 11.1 - bevent DataFrame.

Our Boston RetroObject retains state, allowing us to access the columns attribute we passed into bevent. Simply put, the list comprehension iterates over the column numbers that compose the columns attribute, passing each number into “ro.beventFieldsDict,” a dictionary. A dictionary is a collection of key-value pairs. By passing a key – column number – into beventFieldsDict, we receive a value – column name – in return.

*Voila!* Using a Boston RetroObject, we have successfully created a Pandas DataFrame. This DataFrame contains the default bevent fields for Boston’s July 2021 results. A cursory review of the DataFrame raises an important question, however. How are we going to filter this information? Printing the DataFrame reveals that it is nine-hundred-thirty rows by thirty-six columns.

As it turns out, we will retrieve our desired information with ease. The Pandas method query – not to be confused with the colloquial use of the term deployed throughout this synopsis – is a Pandas method that allows us to filter a DataFrame’s records on the basis of a Boolean expression. We can express a simple yet powerful request with query: if the event text column contains a “K,” then include the record in our result set. Although we can execute this query against the original DataFrame by setting “in place” to True, we will create a new DataFrame with our query results, titled “strikeThree.”



Figure 12.1 - Pandas query method.

After executing this query, we filter for the columns we would like to view, exporting the results to a csv file titled “july\_strikeouts\_first\_export.csv.” Let’s open this file with excel and view the results.

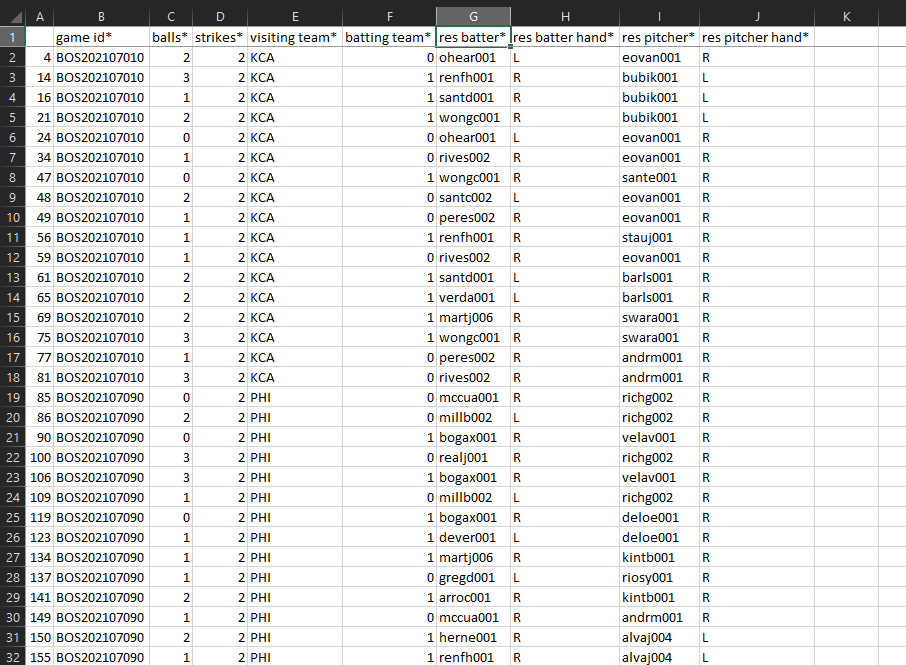


Figure 12.2 – july\_strikeouts\_first\_export.csv.

A nice result for minimal effort, but we can do better. Examining our spreadsheet, we realize that our data must be refined. There are no dates associated with these records. Rather, we have received a game id, a difficult to decipher, alphanumeric representation of the date and venue. Then there is the matter of the “res batter” and “res pitcher” columns, which have returned a primary key representation of the players’ names. This normalization of the data is actually an incredible feat on retrosheet’s part, but it can bemuse those hoping to glean a quick insight from the tools.

Python’s standard library provides us with all of the tools we need to extract the date from a game id. As outlined in figure 13.1, we write four additional import statements at the top of the file. Collectively, they import the datetime class, in addition to a trio of methods from the regex library. Subsequently, we define a function titled “idToDate.” The function uses the regex methods compile, findall, and sub to parse a game id for its year, month, and day components. Once we have retrieved these components, we pass them into datetime and return a date to the caller.

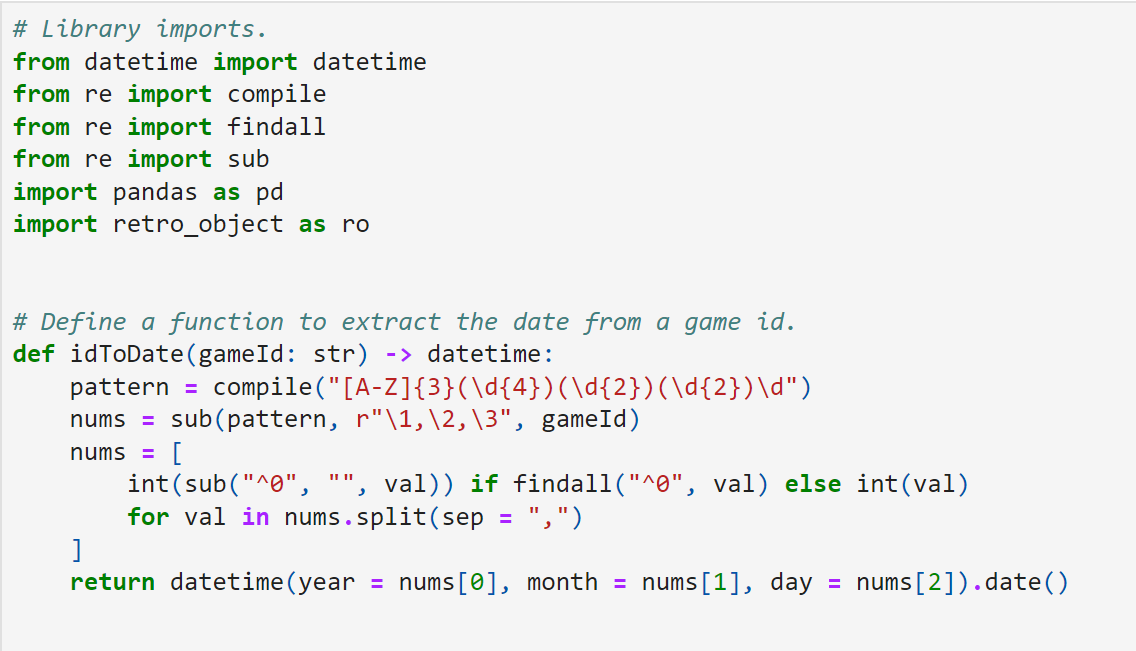


Figure 13.1 - Newly defined function.

After defining idToDate and writing our additional import statements, we chain an additional method to strikeThree – assign, which allows us to add new series to the DataFrame. We add five additional series to the DataFrame: date, batter last name, batter first name, pitcher last name, and pitcher first name. We create the date series by applying the idToDate function to the game id series. Similarly, players’ names are assigned to the DataFrame by applying dictionary calls to the res batter and res pitcher series. We conclude our adjustments by reversing the normalization in the batting team column, replacing zero with “away” and one with “home.” Figure 15.1 captures the results of our second export. Much better, wouldn’t you say?



Figure 14.1 - Revised Pandas query with assign.

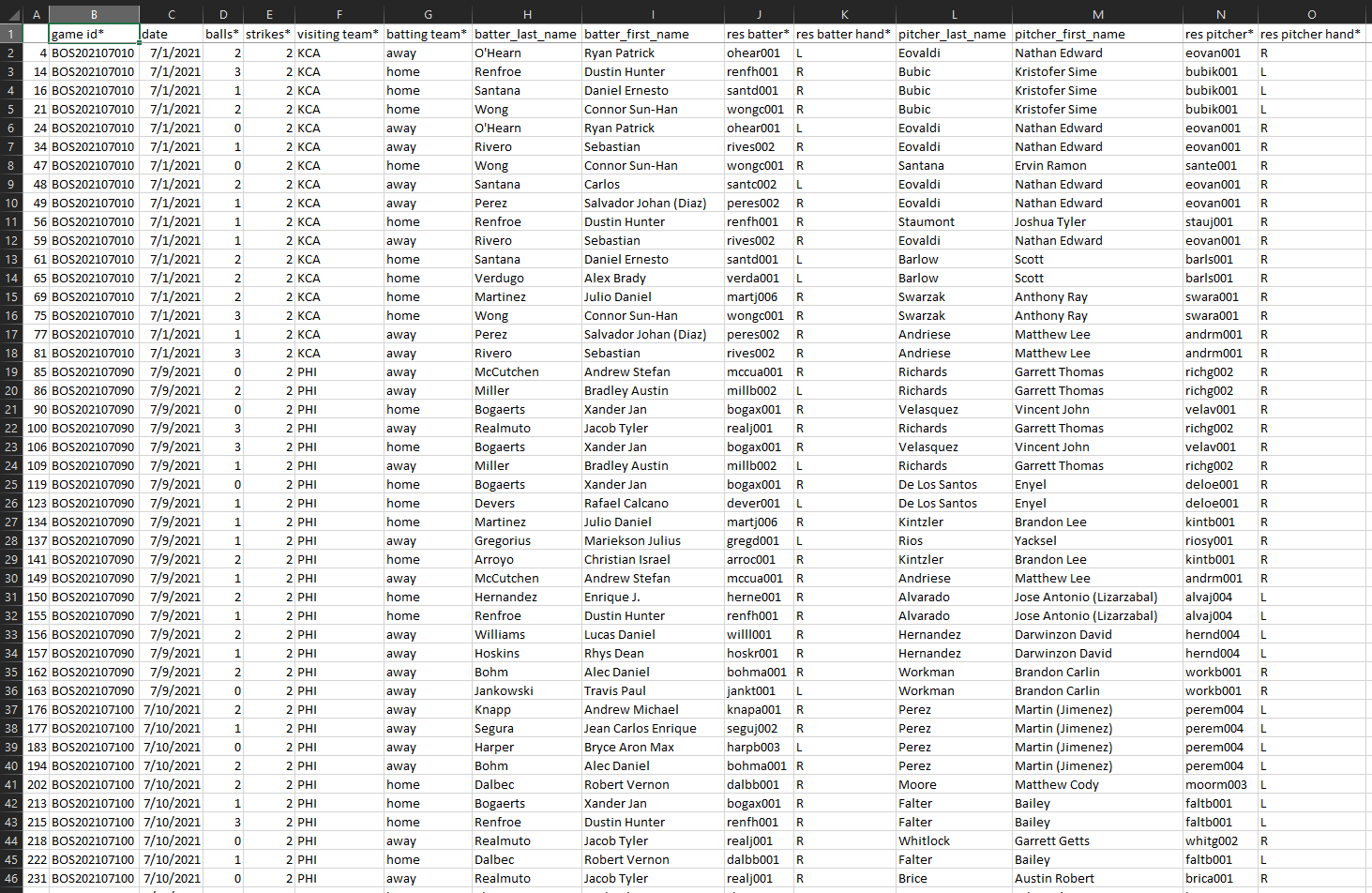


Figure 15.1 – july\_strikeouts\_second\_export.csv.

Let’s up the ante one last time. Instead of retrieving strikeout information for July 2021, we will obtain strikeout information for an entire decade, July 2010 through July 2020. Surely this will require a massive effort on our part…?

The cost of this additional information? A single line. Passing arguments into bevent’s year start and year end parameters executes the desired query for all years in the range, inclusive of the bounds. The final product of our efforts, “july\_strikeouts\_third\_export,” details all two-thousand-one-hundred-eight Fenway strikeouts from July 2010 through July 2020.

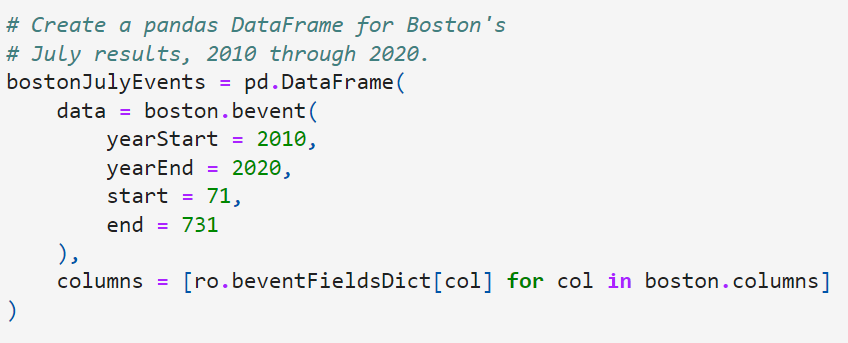


Figure 15.2 – Arguments passed into yearStart and yearEnd.

**The Sky is the Limit**

How should users utilize this module? Our test case demonstrates that it can seamlessly interact with python's data science stack, namely pandas. To that end, I envision a future where a user can package retro\_object.py with pandas and seaborn to produce high quality data visualizations. The module's utility is not limited to data analysis, however. Paired with an object relational mapper like sqlAlchemy, this module could drastically reduce the effort required to build a robust database of MLB statistics, games, and players. A populated database could serve as the springboard to any number of interesting projects, such as expected value calculators and Monte Carlo simulations. Whatever purpose it serves, at the very least, I hope it saves you, dear reader, a little bit of that most precious resource – time.

I would like to conclude this synopsis with a heartfelt thank you to retrosheet. I pursued this project to increase my productivity with their tools, which remain invaluable resources for baseball enthusiasts. That they make such a vast collection of historical information available, free of charge and free of advertisements, is almost anachronistic. I cannot thank them enough for maintaining this beacon of the web.