Naming Conventions

In keeping with python convention, I have written project level file names in snake-case, with a single underscore separating distinct, lower-case words. Module level variable names are written in camel-case style: lowercase letters compose the first word in a variable, and the first letter of any subsequent word is capitalized. In accordance with the PEP8 style guide, classes are an exception to this rule, beginning with a capital rather than lowercase letter.

As highlighted in the project synopsis, the majority of RetroObject's methods format user input into a valid query string. These strings are then passed to “bevent,” “bgame,” and “box.” Since users are not meant to call formatting methods directly, they begin with double underscores, e.g., \_\_createColumnString. Note that the first method in the RetroObject class, the double underscore enclosed \_\_init\_\_, is a common method shared by all python classes. Roughly equivalent to a constructor, \_\_init\_\_ is responsible for creating instances of a class.

Line Length, Indentation, and Blank Spaces

PEP8 recommends limiting most lines to a maximum of seventy-nine characters, with docstrings and long blocks of text limited to seventy-two characters. These restrictions allow readers to split their screens while maintaining code readability, enabling the cross referencing of source code and library documentation. With this goal in mind, continuation lines have been written in accordance with PEP8's indenting recommendations, though there are several recurring patterns that may seem awkward to programmers. For example, longer list comprehensions have been split across multiple lines, resulting in the following pattern:

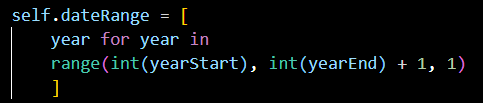


Figure 2.1 – List comprehension split across multiple lines.

While my natural inclination is to write the above comprehension on a single line, I have opted for the current form as a matter of consistency with PEP8. My apologies for any vertigo that may ensue.

With the exception of multi-line expressions, whitespace has been omitted immediately insides parentheses and before commas. However, a single space has been inserted after a comma that separates one value from another. For example, when unpacking a tuple, the code favors the following convention: x, y = 5, 10. Similarly, values passed into method parameters are separated by a single space.

Code Notes

The program begins by importing a number of standard library modules, third party modules, and local modules. Standard library modules are imported first, third party modules second, local modules third. As a secondary sorting convention, modules are ordered alphabetically ascending.

retro\_object imports event\_parser, project\_directories, and project\_variables. event\_parser uses regular expressions to determine the bounds of the event files on record. This will determine default year start and year end arguments for bevent, bgame, and box. project\_directories provides file paths called throughout retro\_object, while project\_variables contains a number of dictionaries that are used throughout retro\_object to determine relevant team information and default column input.

There are several dictionaries located within project\_variables that are not imported by retro\_object. The dictionary of player, manager, and umpire biographical information - titled bioDict - is especially useful to end users. Figure 3.1 demonstrates how these local modules can be imported by a scripts running in a different directory.



Figure 3.1 – Example of local imports.

Next, the module defines the RetroObject class, which contains our program's core logic. Class level attributes are defined before methods, and they may be called by any instance of the class through dot notation. A class object is created by passing a valid "retroString" to a RetroObject call, e.g., RetroObject('WAS'). Please note that ro.validTeams will return a list of acceptable retroStrings. Moreover, capitalization does not matter; the constructor automatically converts user input to upper case.

Like all python classes, the first defined method is \_\_init\_\_, which creates class instances and assigns them the following attributes: "lowerBound," "upperBound," "league," "fullName," and "fileExtension." Generally speaking, the bounds are used to determine if a user requested season is valid, while fileExtension ensures that the proper file is queried - i.e., a query about an American League team is made against a .EVA file, a National League team a .EVN file. fullName and league exist for informational purposes, since team abbreviations are not necessarily intuitive. These attributes are sourced from the json file "team\_extensions.json," which is loaded into the program and stored as the dictionary "teamDict." A sixth attribute, titled "entry," will return the dictionary value associated with a retroString, which serves as a dictionary key.

Please note that two teams - the Houston Astros and Milwaukee Brewers - have played in both the American League and the National League. Consequently, they have two entries for every attribute, each ending in a one or two, e.g., lowerBoundOne versus lowerBoundTwo.

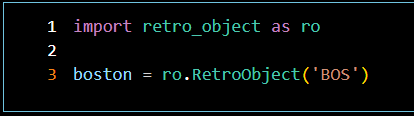


Figure 4.1 - RetroObject call signature.

The first method encountered after \_\_init\_\_ is bevent, which returns play-by-play records as a list of lists. Bevent begins by changing the current working directory to the application directory, and it concludes by returning to the module level directory. This pattern is repeated for bgame and box.

Six arguments may be passed to bevent, dictating the column numbers to be queried, the desired date range from which to pull information, and the seasons requested by the caller. After assigning these variables to self, the method runs a conditional statement to determine whether the caller passed both a game id and start/end parameters to the method. If the caller passed both types of arguments, then the program warns that game id takes precedence over start and end, and only one record - that associated with the game id - will be returned.

Subsequently, three variables are created - dateRange, output, and columnValidation. Bevent iterates over the years in dateRange with a for loop, running its executable for every year in the range and concatenating the results to output. \_\_columnValidation, assigned to \_\_columnPipe's output, helps construct the query string passed into the run method.

Once the loop has concluded, bevent uses the string method, "split," to convert rawOutput into a list, with newline characters serving as the demarcation separating entries. Formatting changes are made to entries within output using a list comprehension and the string method "replace," and the program splits the list by comma, creating a list of lists. Finally, bevent loops over every list in output, calling the \_\_convertToInt method to change strings to integers, where appropriate. Output is returned to the caller.

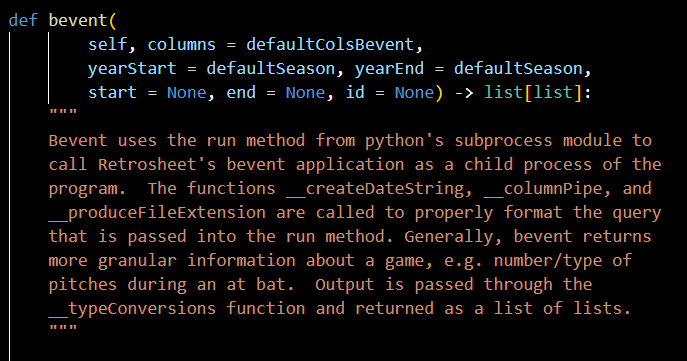


Figure 5.1 - bevent call signature and doc string.

The two methods after bevent - bgame and box - mirror bevent's structure. There is only one meaningful difference between the type of data returned by bevent and bgame - the latter contains three columns that return date information. To accommodate these columns, bgame calls \_\_typeConversions rather than \_\_convertToInt, changing string values to datetime objects where appropriate. Note that \_\_typeConversions calls \_\_mergeDateTime to properly format the time component in any datetime object.

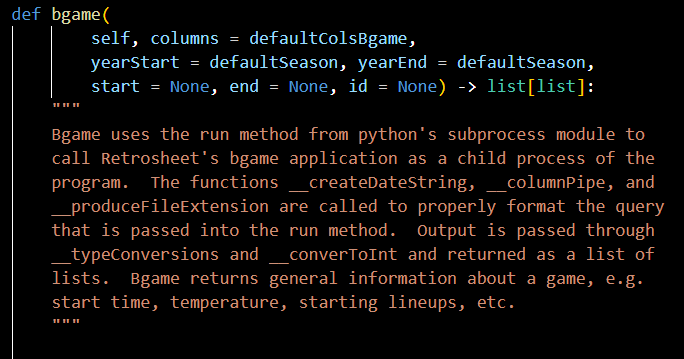


Figure 6.1 - bgame call signature and doc string.

Box is simply a truncated version of bevent and bgame. The method returns output once it has exited the for loop, with no additional formatting. Box's functionality is largely identical to its concomitant command line application, allowing the user to return box scores from multiple seasons with a single call.

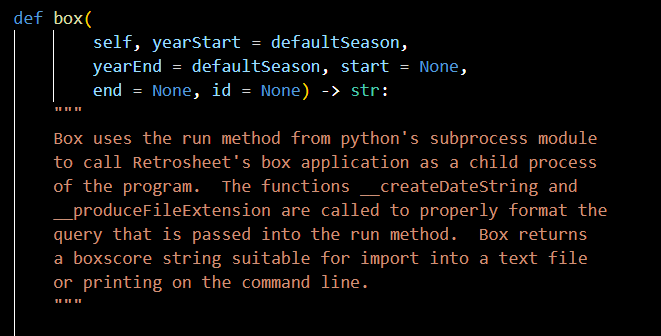


Figure 6.2 - box call signature and doc string.

The remaining methods in RetroObject serve a singular purpose - format the input passed into, and the output returned by, the emulation methods bevent, bgame, and box. They accomplish this in a nested manner. Relatively simple formatting methods are passed into more complex ones, which are in turn called by the emulation methods.

\_\_columnPipe, called by bevent and bgame prior to executing their for loops, converts the user's requested columns into a valid string. While this may seem like a simple task, quirks in the command line applications complicate its execution.

Internally, bevent and bgame columns begin with zero and run through eighty-four and ninety-six, respectively. However, if a user were to pass every column number to an executable, then an error message is returned, indicating that the requested columns exceed the number of columns. This is a semantic error, with the retrosheet application conflating column maximums with the length of the column array requested. To prevent this error, \_\_columnPipe returns a column string in the following form: 0-84. This convention is applied to all consecutive column numbers, such that a request for columns "1, 2, 3, 4, 5, 20, 21, 22" would return "1-5, 20-22." Although \_\_columnPipe returns a column string, it cedes the logic to another method, titled "\_\_createColumnString." \_\_columnPipe also returns a sorted array of the columns selected, and in the case of bgame, the difference between the index positions of columns one and four in the sorted column array.

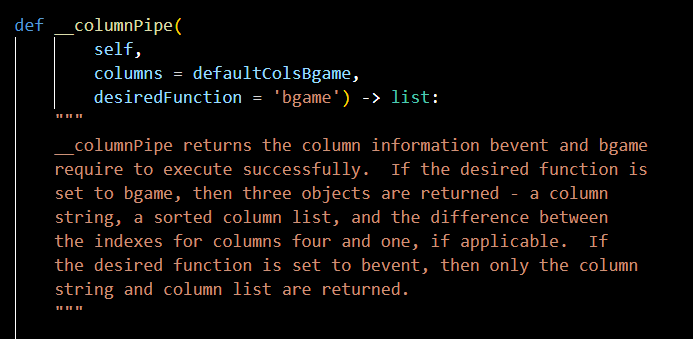


Figure 7.1 - \_\_columnPipe call signature and doc string.

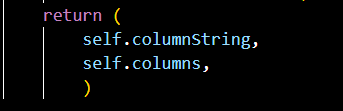


Figure 8.1 - \_\_columnPipe return value.

\_\_createColumnString is straightforward in its approach. An array of column numbers is passed to the method. A for loop iterates over the array, attempting to assign the next element to the variable nextValue. If this assignment does not succeed, then iteration has concluded, and columnString is returned. Otherwise, the method determines whether the next value is one greater than the current value. If it is, then the loop continues to the next value. If it is not, then a string of the following form, minValue-value, is concatenated to columnString, and minValue is assigned to nextValue.

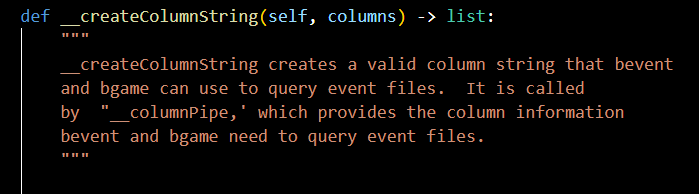


Figure 8.2 - \_\_createColumnString call signature and doc string.

When bgame calls \_\_columnPipe, \_\_columnPipe calls the method \_\_linkDateColumns. If a user requests bgame's fourth column without its first column, then \_\_linkDateColumns appends the first column to the list of requested columns. Because bgame's fourth column is a time string, it is devoid of meaning without an associated date. In recognition of this deficiency, \_\_linkDateColumns prevents the fourth column from being returned without a date component.

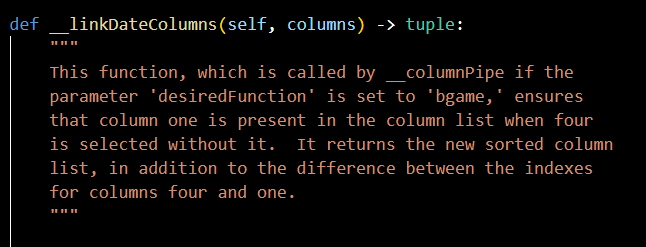


Figure 9.1 - \_\_linkDateColumns call signature and doc string.

The other formatting method directly called by bevent, bgame, and box is \_\_createDateString. It returns the date components used in a valid query string, accomplishing this feat by incorporating two additional methods: \_\_convertToDatetime and \_\_evalGameId.

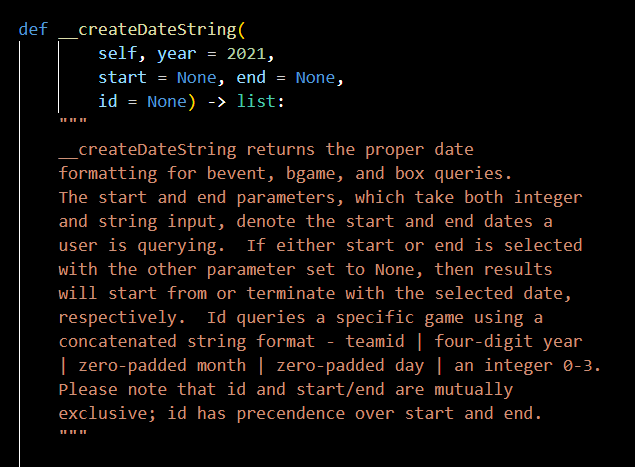


Figure 9.2 - \_\_createDateString call signature and doc string.

\_\_createDateString begins via tuple assignment, with year, start, and end referencing \_\_convertToDatetime's output. \_\_convertToDatetime uses another method, \_\_convertDatetimeComponents, to ensure that year, start, and end are either datetime or NoneType objects.

Once year, start, and end have been properly assigned, \_\_createDateString runs a gauntlet of conditional tests and returns the properly formatted date string. If a game id is passed to the method, then the id is passed to \_\_evalGameId, which determines whether the id is properly constructed. If the formatting is correct, then the game id is returned.

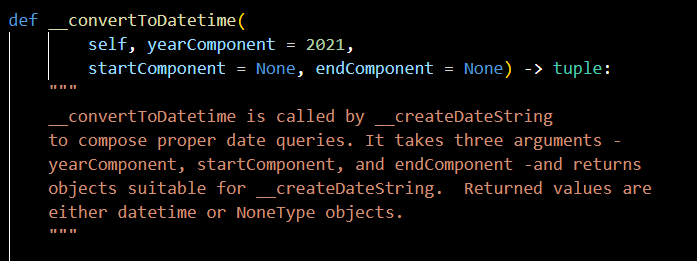


Figure 10.1 - \_\_convertToDatetime call signature and doc string.

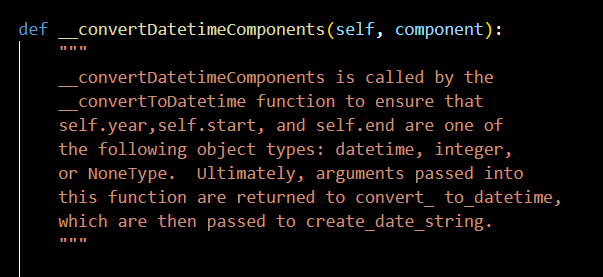


Figure 10.2 - \_\_convertDatetimeComponents call signature and doc string.

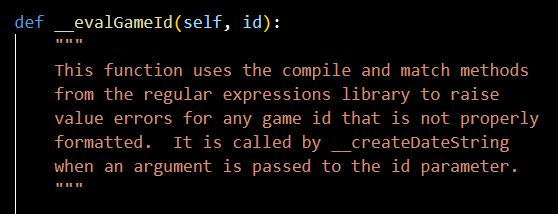


Figure 11.1 - \_\_evalGameId call signature and doc string.

As described, the program's control flow is fairly basic. Emulation functions call \_\_columnPipe and \_\_createDateString to form valid queries. In turn, \_\_columnPipe and \_\_createDateString call methods to accomplish this task. The output returned by python's run method is passed through \_\_convertToInt and \_\_typeConversions and returned to the caller. For more information regarding a method, please reference that method's docstring.

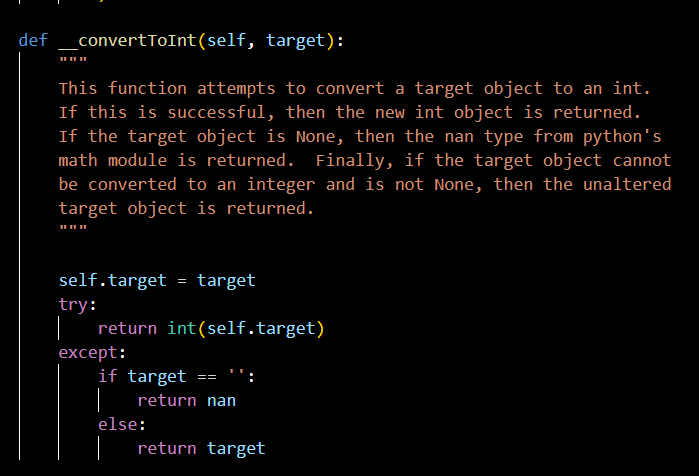


Figure 11.2 - \_\_convertToInt call signature and doc string.

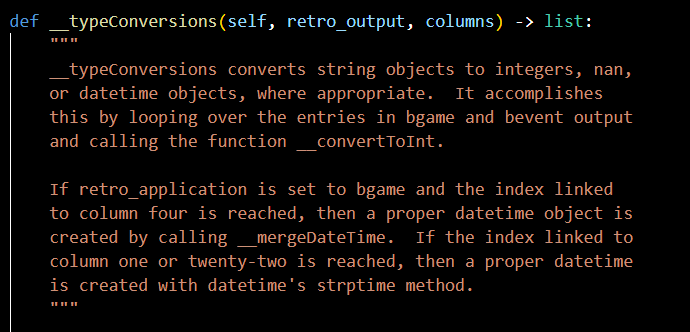


Figure 12.1 - \_\_typeConversions call signature and doc string.