Northeastern University
DS2001 – Programming with Data — Social Science Practicum
Fall 2019, Mattsson & McCabe

Practicum 5: Ranking

In this week's practicum we will be asking Python to process a larger amount of data than we have in the past. We will use functions to make this more tractable, and, using baseball data as an example, explore ways of ranking data to isolate interesting individuals.

1 Baseball

Baseball has been of interest to data scientists and statisticians, and vice versa, for some time. The quality of data in baseball is generally quite high; we have play-by-play data for virtually every game since the 1930s and summary box scores going back to the 1890s. The sport also lends itself to statistical analysis better than other sports, because it is easier to divide into discrete events and quantify than basketball or even football. Statistical analysis of baseball—sabermetrics—has been around since the 1980s or so but really took off in the 2000s with the success of the low-budget Oakland A's and the popularization of their story through the book *Moneyball*. We'll use some of that data today, studying batting performances since 1969 using data collected in the well-known Lahman database.

1.1 Download

Please download `pr05.zip` from the practicum website (or Blackboard), unzip it, and move the directory/folder to where you want it in your file system. Use Atom's `File > "Add Project Folder..."` to open this folder and view `baseball.py`.

2 Viewing and filtering data

2.1 Viewing snippets of the data

This dataset is considerably larger than the previous datasets we have worked with. Printing it all out will produce a ton of output that we don't want to scroll through. While looking at the whole data at times can be useful, we want to write a function to print the header plus the first few rows of the dataset.

2.2 Filtering numeric columns

This dataset consists of a number of numeric columns, from year to games played to counts of various outcomes. We are often interested in specific subsets of these data: for example, all player-seasons from the 1990s, or all players with at least 200 hits, and so on.
Using patterns you've learned in previous weeks, like combining `for` and `if` with list `append`s, write a function that takes the data, a column index, and a minimum and maximum value, and returns a subset of the data (in the same form - i.e., a list of lists with the same number of columns) matching these criteria.

Evaluating this filtering function. We can test this filtering function using some commonsense assumptions and by referring to baseball records. Some tests:

- almost all of these variables are counts; that is, they should be either zero or positive. So if the minimum value is zero, and the maximum number is something unreasonably large—say, 10,000—then the function should just return the whole dataset.
- Ichiro Suzuki set the single-season hits record in 2004, with 262 hits.
- In the same year, 2004, Barry Bonds set the major league record for walks (232)
- Rickey Henderson stole 130 bases in 1982.

2.3 Filtering categorical columns

We have a few string-valued columns that we may want to filter on as well. Write a function that takes our data and a list of strings, then uses `in` to filter

Evaluating this filtering function. When filtering by team, you may want to use our `head` function to avoid writing too many rows to the screen.

- The Montreal Expos (`MON`) folded in 2004 and were replaced by the Washington Nationals (`WAS`).
- While in the historical data there are other leagues, in the post-1969 expansion era, there are only two leagues, the AL and the NL, so every row should have a `league` of AL or NL.

2.4 Finding specific cases: Franchise records

Both of these filtering functions take a dataset (in list of lists form) and return a dataset (in list of lists form). This means that the output of one filtering function can be the input of another one; i.e., we can chain filter calls to analyze specific subsets of the data. To that end, find and record the names of the players who hold the following single-season team records:

- the Atlanta Braves (ATL) single-season strikeout leader (look for $>$ 160 SO)
- the Boston Red Sox (BOS) single-season home runs leader (look for $>$ 50 HR)

3 Ordering data

We have used single-season records to validate our filtering functions. Now, let's move on to finding baseball records when we don't already know them. We will use Python's built-in `sorted` function to order our data, and combine this with our own `head` function to print the top single-season records for various baseball stats.

3.1 Using helper functions

Python's `sorted` function will sort any list you give it, returning the sorted list. This function behaves like you would expect it to for simple lists. But we want to use `sorted` to sort a list where each element is itself a list of baseball statistics for one player in one year. The built-in function has no way of knowing what we mean by "sort" in this case, unless we tell it. For this, the `sorted` function takes an optional parameter `key`, which should be a function indicating what to use to do the sort.
Since we wish to sort our dataset by a specific value (say, year, or home runs), we need to define helper functions that extract that value from each row (i.e., a player-season). If we wanted to sort by first names, we would need to write a function that extracts the first name from one row. That would look like:

```
def get_first_name(row):
    return row[0]

sorted(our_list_of_lists, key=get_first_name)
```

Go ahead and write the helper functions that you know you'll need.

3.2 Single-season leaderboards

Combining the `sorted` function with out `head` function, we can now create leaderboards for specific stats. Write a `leaderboard` function that takes four parameters: the header, an input dataset, a Helper function, and the number of rows to print along with the header.

Evaluating the leaderboard function. Without domain knowledge, it's a little trickier to validate these outputs. But we can try some commonsense tests. For example, getting the leaderboard of first names should return a bunch of Zacks (because Z is the "largest" letter):

```
leaderboard(baseball_header, baseball_data, get_first_name, 5) # -> 2 Zoilos and 3 Zacks
```

Does this work? Probably not! By default, the `sorted` function places the *smallest* values first and we want the *largest*. Thankfully, `sorted` can accept another optional parameter `reverse`, which is a boolean value that is `False` by default. Try this instead:

```
sorted(our_list_of_lists, key=get_first_name, reverse=True)
```

3.3 Using leaderboards to find records

Using our `leaderboard` function, we can find the same records as before, plus others. So use the function to replicate and extend your earlier analysis and answer the following questions:

1. We saw before that 2004 Ichiro Suzuki set the record for hits in a season. He also holds second place (2001). Who is next?
2. How many times is Barry Bonds in the top 10 in walks in a season?

(Optional) Using anonymous functions Writing out these helper functions each time can be pretty tedious. An alternate strategy is to use **anonymous functions** with the `lambda` keyword. This keyword returns a function in its place (without a name, thus "anonymous function"); for example

```
lambda x: x + 1
```

is equivalent to writing

```
def <something>(x):
   return x + 1
```

In general, it's better to define functions normally (i.e., with names) so that you can call them later in your code. However, in some cases, like with `sorted`, we want to create a function that will only be used in that spot. So, if we wanted to sort a list of lists by the 0th index, we could write

```
sorted(our_list_of_lists, key=lambda x: x[0])
```

This is saying take `our_list_of_lists`, and for each element (a list), look inside at the 0th element to determine where it should go in the sort; i.e., it's the same thing we did with the `get_first_name` function.

4 (extra) An application: "Three True Outcomes" in baseball

4.1 Combining columns

There are three outcomes of a plate appearance that don't involve the defense at all, only the pitcher and the batter: a walk, a home run, or a strikeout.[1] Three-true-outcome players tend to be patient sluggers who combine a low batting average with lots of home runs and walks. This style of play is sometimes considered boring; after all, it's an outcome where there's no opportunity to see an exciting defensive play. It's supposedly become more common in recent years, to the despair of baseball announcers everywhere. Let's create a `true_outcome` column in the dataset summing a player's walks, home runs, and strikeouts in a season and use it to explore the TTT trend.

4.2 Aggregating data and getting an answer

We're interested now, not in player-seasons, but in trends across seasons. So here we want to aggregate things up to the season level. For each year, we want to add up all the plate appearance and "true outcomes" across-player seasons, then calculate the ratio

$$TT\% = \frac{\text{num.}\, TT}{\text{num.}\, PA}$$

for each year. Has it increased over time? How much?

5 Codebook

Index	Header	Detail
0	first_name	
1	last_name	
2	year	Season
3	stint	The nth team someone played for this season
4	team	Team name (short ID)
5	league	League (AL/NL)
6	AB	Times at bat
7	H	Hits
8	HR	Home runs
9	SB	Stolen bases
10	BB	Walks
11	SO	Strikeouts
12	HBP	Times hit by pitch
13	PA	Plate appearances

This handout was originally created by Carolina Mattsson and Stefan McCabe, Fall 2019.

[1]Being hit by a pitch should also be "true outcome" by this definition, but no one seems to care about that, so let's ignore it.