Authors: Brett Irvin, Stephanie Djajadi and Yanay Rosen

Intro

Tables are a convenient way to store data.

A table can be viewed in two ways:

- a sequence of named columns that each describe a single aspect of all entries in a data set, or
- a sequence of rows that each contain all information about a single entry in a data set.

Strategies for solving table questions

When answering problems involving tables ask yourself these questions:

1. What information do you have, and in what table(s)?
    a. Oftentimes you will be given a table with many extra columns, which columns are relevant?
    b. What kinds of data is stored in those columns? Is it categorical, numerical, something else?
2. What information do you need to produce?
    a. What data type should your final answer be? What variable should it be assigned to?
    b. What does that final answer represent?
3. Where does that information come from?
    a. Is the information you are looking for a particular value connected to a row that is somewhere “hidden” within the data set like a maximum or minimum value?
    b. Is it some type of table/column wide summary like an average?
4. What intermediate information do you need?
    a. Is everything you need contained within one single table? Or do you have multiple tables with a column in common that can be joined to create a more comprehensive table?
    b. Do you need to create any other data from what’s stored in existing columns?
    c. Draw out the initial table, the final table, and then every table in between them
    d. Be wary of information loss! Table operations can cause you to lose information stored in the original table due to aggregation or dropping columns. Be aware of the order in which you combine operations to maintain data through the point where it is needed.
5. What functions do you need to make these intermediate tables?
    a. For each transition between tables identify which function(s) can be used
6. Produce your final answer
    a. Combine the functions and data you identified in the previous steps to produce an answer
    b. Check all the intermediate tables to make sure they are what you expected

Examples from previous semesters' exams

Let's take a look at an example from the Fall 2019 Midterm for how to apply this process.

4. (27 points) Table operations

A table `trees` contains one row for each tree in a small forest. Here are the first few rows:

year	wood density	burned	type	height
1980	0.813	no	oak	23
1829	0.529	yes	redwood	127.47
2000	0.601	no	pine	5.2

The table contains five columns:

- **year**: an int, the year the tree was planted
- **wood density**: a float, the density of the tree's wood
- **burned**: a string, which indicates whether the tree was ever burned in a fire (`yes` or `no`)
- **type**: a string, what kind of tree it is
- **height**: a float, the height of the tree in meters

(g) (4 pt) A new table `type_heights`, which has one row per type. It should have two columns: one labeled `type` that has the name of the type, and one that has the average height of trees for that type. It should not have any other columns.

1. What information do we have in which table?

We only have one table, the `trees` table. It has 5 columns, described above. The `year` column is categorical (although we can extract numerical data from it like ages of trees and/or order it), the `wood density` column is numerical, the `burned` column is categorical, the `type` column is categorical and the `height` column is numerical. The question asks us to produce a new table with a row for type and a row for average height, so we know that the `type` column and `height` column are important.

2. What information do we need to produce?

The question wants us to find the average heights of trees for each type.

3. Where does that information come from?

The information is not directly in the table, but it is a table wide summary value.

4. What intermediate information do we need?

All the information we need is in the original `trees` table as it has the heights and types of all trees so we don't have to create any initial data. **NOTE: we are showing these tables handwritten because you should be writing these steps out on paper!** *"..." indicates values that we don't know.*

Original trees table

Year	wood density	burned	type	height
1980	0.813	no	oak	23
1829	0.529	yes	redwood	127.47
2000	0. 601	no	pine	5.2
...	...	...	...	...

Intermediate table #1

type	height
oak	23
redwood	127.47
pine	5.2
oak	...
Pine	...
...	...

final table

type	height average
birch	...
maple	...
oak	...
pine	...
redwood	...
...	...

5. What functions do we need to accomplish these intermediate tables?

To go from the full table to the first intermediate table with only the relevant columns we can use `.select`. To go from the first intermediate table with two columns to the final table we can `group` on `"type"` and use `np.average`. How did we know that? Check the next sections about which functions are useful for what

6. Producing Final Answer

We know we'll have to do `.select("type", "height")` and then also `.group("type", np.average)`. Let's take a look at the blanks provided in the question on the midterm and see where we could put those:

(g) (4 pt) A new table `type_heights`, which has one row per type. It should have two columns: one labeled `type` that has the name of the type, and one that has the average height of trees for that type. It should not have any other columns.

```
trees_2col = trees._____(_____)

type_heights = trees_2col._____(_____)

type_heights
```

`select` and it's arguments can fill in the first two blanks, and `group` and its arguments can fill the next two blanks!

ANSWER BELOW:

(g) (4 pt) A new table `type_heights`, which has one row per type. It should have two columns: one labeled `type` that has the name of the type, and one that has the average height of trees for that type. It should not have any other columns.

```
trees_2col = trees.select('type', 'height')

type_heights = trees_2col.group('type', np.average)

type_heights
```

Spring 2018 Midterm Example

1. (18 points) Basketball Bonanza

Assume we have a tables for the 2016-2017 NBA Season. Assume that if a column contains numbers, then they are integers, and otherwise, it is a column of strings.

The **nba** table contains 8 columns. The first few rows are shown below.

player	prefix	position	age	salary	games	minutes	points
Al Horford	BOS	C	30	2.65401e+07	68	2193	952
Amir Johnson	BOS	PF	29	1.2e+07	80	1608	520
Avery Bradley	BOS	SG	26	8.26966e+06	55	1835	894
Demetrius Jackson	BOS	PG	22	1.45e+06	5	17	10
Gerald Green	BOS	SF	31	1.4106e+06	47	538	262
Isaiah Thomas	BOS	PG	27	6.58713e+06	76	2569	2199
Jae Crowder	BOS	SF	26	6.28641e+06	72	2335	999
James Young	BOS	SG	21	1.8252e+06	29	220	68
Jaylen Brown	BOS	SF	20	4.743e+06	78	1341	515
Jonas Jerebko	BOS	PF	29	5e+06	78	1232	299

Fill in the blanks of the Python expressions to compute the described values. You must use only the lines provided. The last line of each answer should evaluate to the value described. Assume that the statements `from datascience import *` and `import numpy as np` have been executed. You may add anything you would like to the blanks below, but you may not add code outside of the blanks.

(d) (5 pt) The number of positions for which the total points scored by CLE players in that position was higher than the total points scored by BOS players in that position.

1. What information do we have in which table?

We only have one table, `nba`. `player`, `prefix` and `position` are categorical; `age` can be either categorical or numerical; `salary`, `games`, `minutes` and `points` are numerical.

2. What information do we need to produce?

An integer, the total number of positions from CLE players that scored more total points in each position than BOS.

3. Where does that information come from?

The information can be found in the table by comparing the scores for BOS players and CLE players, using information from the `prefix` and `points` columns.

4. What intermediate information do we need?

We need an intermediate table that has a column that contains each position and a column for the total scores of BOS players at the respective position and a column for the total scores of CLE players at the respective position. Then, we need an array that is `True` where the CLE score for a position was higher and `False` otherwise. **NOTE: we are showing these tables handwritten because you should be writing these steps out on paper!** *"..." indicates values that we don't know by hand.*

Original NBA Table

Player	prefix	position	age	salary	games	minutes	points
Al Horford	BOS	C	30	2.65e07	68	2193	952
Amir Johnson	BOS	PF	29	1.2e07	80	1608	520
...	...	...	...	...	...	...	...

Intermediate table #1

Position	ATL	BOS	BRK	CHI	CHA	CLE	...
C	...	...	...	...	...	...	...
PF	...	...	...	...	...	...	...
PG	...	...	...	...	...	...	...
SF	...	...	...	...	...	...	...
SG	...	...	...	...	...	...	...

Intermediate arrays

BOS total points per position — array([..., ..., ..., ..., ...])

CLE total points per position — array([..., ..., ..., ..., ...])

Boolean array comparison — array([True/False, ..., ..., ..., ...])

5. What functions do we need to accomplish these intermediate values?

Pivot. We want a column for each team prefix and a row for each position possible. To fill in the intersection values of the new table we want to take a sum of the points for each player that fits the team prefix and position prerequisites. To compare the two scores and generate a boolean array we have to first access the summed scores using .column and then use a comparator like > to produce a boolean array.

6. Producing Final Answer

Finally, we want to produce a count of `True` values so we use either `np.count_nonzero` or `sum`. This stands for the total number of CLE positions that outscored BOS positions in summed points.

(d) (5 pt) The number of positions for which the total points scored by CLE players in that position was higher than the total points scored by BOS players in that position.

```
positions = nba.pivot('prefix',_____________________________________________________)


sum(_____________________________________________________________________________)
```

We know we need to pivot using a pivot column of `"prefix"`, a group column of `"position"`, a values column of `"points"` and a collect function of `sum`. Then we need to compare the CLE and BOS arrays using > so that we generate a boolean array that's `True` for where CLE outscored BOS at a position. We can then use either `sum` or `np.count_nonzero`.

Looking at the provided code we can see `pivot` is provided for us as is the first argument of `"prefix"`, so we just need to fill in the three other pivot arguments in that line. Next, we see that the last line has `sum` filled in already to convert the boolean array to an int, so we just need to fill in the first `.column` call, then the appropriate comparator and the other `.column` call.

ANSWER BELOW:

(d) (5 pt) The number of positions for which the total points scored by CLE players in that position was higher than the total points scored by BOS players in that position.

```
positions = nba.pivot('prefix',_____________________________________________________)


sum(positions_____________________________________________________________________)
positions = nba.pivot('prefix', 'position', 'points', sum)
sum(positions.column('CLE')> positions.column('BOS'))
```

Table Functions Quick Overview

There are various manipulations of tables we can do, each one can only be done by specific functions. No table function changes the original table unless you assign the result back to the original table name. In general, you should avoid doing that and instead assign the result to another variable name.

Create a table

- Table.read_table(filename)
    - Creates a new table by reading the file located at the filename
- Table()
    - Creates a new, completely blank table that you can add columns to

Change the order of rows

- tbl.sort('column_name', descending = True/False)
    - Returns a copy of the table with the rows sorted by the values in the specified column with the specified
    - Descending: largest to smallest value
    - Ascending: smallest to largest value. This is the default ordering.

Change the number of rows

- tbl.where(column_label, predicate)
    - Returns a copy of the table that only has rows in which the value in the specific column matched the predicate specified.
    - For a list of predicates, check the python reference
    - The new number of rows is equal to the number of rows in the original table that matched that predicate
- tbl.group(column or array of columns, function)
    - Returns a copy of the table that has one row for each unique value in the specified column. If you passed a list of columns then the new table will have a row for each unique **pairing** of those values
    - The other columns are either a column of the count of how many times that unique value or pair occurred in the original table, or, if you passed a function as the second argument, it contains all the other columns that were not mentioned in the first argument with the function applied to their values.
    - That means that the value at row a and column b will have the value of the function specified called on an array containing all the values from the original column for rows that matched row a.
    - Remember that the column names are changed! They have the name of the function added to the end
    - The new number of rows is equal to the number of unique values (or value pairs) in the original table's specified column(s)

- tbl.pivot(column_label_1, column_label_2, column_label_3, function)
    - Creates a pivot table that has one row for each unique value in the second column argument, and a column for every unique value in the first column argument.
    - If the third and fourth arguments are not specified, the value will be a count of how many times the row+column pair occurred in the original table.
    - If the third and fourth arguments are specified, the value will be the result of calling the fourth argument, a function, on an array of the values in the original table in the third column argument that occurred in rows that matched the value of the row and column.
    - This is the only table function that takes values from inside the table and turns them into **columns**
    - The new number of rows is equal to the number of unique values in the second column argument
- tbl.take(int or array)
    - Returns a copy of the original table that has only has the specified rows
    - Row indexes start at 0
    - The new number of rows is equal to 1 if you passed in an int or equal to the length of the array you passed in

Change the number of columns

- tbl.group
    - If no collect argument is passed, the new table will only have two columns, the grouped column and a count column
- tbl.pivot
    - The new number of columns is 1 + the number of unique values in the pivot column argument (the first argument)
- tbl.drop(column labels)
    - Returns a copy of the original table without the specified columns
    - The new number of columns is equal to original number of columns - number of columns you dropped
- tbl.select(column labels)
    - Returns a copy of the original table with only the specified columns
    - The new number of columns is equal to the number of columns you passed as an argument
- tbl.with_columns(new label, array of values, …)
    - Returns a copy of the original table with new columns added
    - The new number of columns is equal to the original number of columns + the number of columns added

Aggregate data

View http://data8.org/interactive_table_functions/ for interactive demos of group and pivot.

- tbl.group
    - Can aggregate data at the level of one column or more
    - Tells you information about data that falls into categories defined by one or more columns
    - Can be counts or something else like averages
- tbl.pivot
    - Can aggregate data at the level of two columns
    - Tells you information about data that falls into categories defined by two columns
    - Can be counts or something else like averages

Get specific information from a table

- tbl.column(column label).item(i)
    - Get the specific value at column and index i
    - Rows are zero indexed
    - Commonly used after sorting to get the value of some column that corresponds to the row that has the min or max value in some other numerical column
- tbl.num_rows
    - Returns the number of rows
    - Can be used after where for example to tell us how many value match a certain predicate
- tbl.num_columns
    - Returns the number of columns

Call a function on every item in a column

- tbl.apply(function, optional column label/index 1, optional column label/index 2 ...)
    - Calls the provided function on every row in an array.
    - If no columns are specified, each special row object is passed in
    - Given that row object row, we can call row.item(column index or label) to get the individual values from that row. A row has all the same column values as the original table.
    - You can specify which columns should be passed as arguments to the function using the column arguments. i.e. if a function has two arguments then you could call some_table.apply(that_function, column_for_first_argument, column_for_second_argument)
    - Note that you are passing in the name of the function, so don't follow it with parentheses or any arguments!

We hope this guide will help you!