Data 8
Spring 2016

Categories and Grouping

Discussion 4

Group
Data 8 has opened up a marble store, but we only sell our marbles in certain bunches. as follows:

Color	Amount	Price
Red	4	1.30
Green	6	1.20
Blue	12	2.0
Red	7	1.75
Green	9	1.40
Green	2	1.0

Let's use the table for the rest of the discussion, calling it *marbles*. Right now, it has three columns, the color of the marble, the amount of marbles, and the overall price of this set of marbles
The first function we are going to talk about today is group.

The underlying question for now is, how do we get the overall price and amount for each unique color?
****

Group takes in at least 1 argument. That is, when we say `marbles.group()`, we need to put at least one thing inside of the parenthesis. This argument is the category which we are going to be grouping upon. For example, let's say we say `marbles.group('Color')`. Then, we will output a new table which will have only 1 row for each *unique* column. So, essentially, what it does is it *groups* each value in the column we pass in so that we only get one row per unique value. If we only pass in one argument, it will simply count the number of times that value was in the table. For example, the output of `marbles.group('Color')` is:

Color	count
Red	2
Green	3
Blue	1

Notice that in our new table, we never repeat colors in the 'Color' column. They were all grouped together into one single row. Then, in the 'count' column, it simply counted the amount of times that that specific color was in the original *marbles* table. Green was there 3 times, Blue was there once, and Red was there 2 times.
Now, however, let's say we want to find the maximum amount and the maximum price for each color in the row. For example, for Red, the maximum amount is 7 and the maximum price is 1.75, of the two red rows in the original table. *group* can take in a second argument, a second value in the parenthesis. This argument should be a function which takes in many things and outputs one value. For example, the *max* function takes in many things and returns the biggest element. So, what would the call to `marbles.group('Color', max)` look like?

Color	Amount max	Price max
Red	max(array[4,7])	max(array[1.30, 1.75])
Green	max(array[6, 9, 2])	max(array[1.20, 1.40, 1.0])
Blue	max(array[12])	max(array[2.0])

Which evaluates to

Color	Amount max	Price max
Red	7	1.75
Green	9	1.40
Blue	12	2.0

The main interesting thing to note here is that we didn't have to use max, we could've used any function which takes many things, an array, and returns one.
With this in mind, let's go back to the underlying question and answer it. **How do we get the overall price and amount for each unique color?**

Groups

So, we have seen group, and now it's time to talk about Groups. As you might notice, the *groups* function's name bears an awful resemblance to the *group* function which we have already talked about, and this is for a good reason. Now, the first argument of *groups* should take in a list of arguments you would like to group together. Now, instead of each unique element from a category being in its own row, now it's the *unique combinations of elements* from the categories we provide that must have their own row. For this, we will need to update the *marbles* table by adding a column 'Shape', which is either round of rectangular.

Color	Shape	Amount	Price
Red	Round	4	1.30
Green	Rectangular	6	1.20
Blue	Rectangular	12	2.0
Red	Round	7	1.75
Green	Rectangular	9	1.40
Green	Round	2	1.0

Now, our underlying question is: What is the total price and amount of each type of marble, where a specific type depends on both its color and its shape.
****

groups, much like *group*, takes in atleast one argument. Here, however, the argument is a list of categories we would like to group together, instead of one singular item. For example, `marbles.groups(['Color'])` would do the same thing as `marbles.group('Color')`, as there's only one thing in the list. However, `marbles.groups(['Color', 'Shape'])` would return a new table which looks like this.

Color	Shape	count
Red	Round	2
Green	Rectangular	2
Blue	Round	1
Green	Round	1

There were two marbles that were Red and Round, and hence, the count column is 2. Green appears in two rows in this table, but that is because a Green and Rectangular marble is different than a Green and Round marble.
Notice here, we didn't pass in a second argument to *groups*, so the behavior is similar to that of *group*. But, as the two functions are similar, we have the same choice for *groups*. We can pass in a second argument which takes in a list of things and outputs one item. Let's stick with max to see the similarity. So, the call to `marbles.groups(['Color', 'Shape'], max)` would essentially be this:

Color	Shape	Amount max	Price max
Red	Round	max(array[4, 7])	max(array[1.30, 1.75])
Green	Rectangular	max(array[6, 9])	max(array[1.20, 1.40])
Blue	Rectanglular	max(array[12])	max(array[2.0])
Green	Round	max(array[2])	max(array[1.0])

Which evaluates to:

Color	Shape	Amount max	Price max
Red	Round	7	1.75
Green	Rectangular	9	1.40
Blue	Rectangular	12	2.0
Green	Round	2	1.0

Once again, we find the maximum price and the maximum amount for each unique row. Here, the most interesting thing to note is that Green and Round marbles has its own category, and even its own max amount and its own max price. Remember, max isn't the only function we could pass in as the second argument.
Now, going back to the underlying question for this section, **What is the total price and amount of each type of marble, where a specific type depends on both its color and its shape.**

Pivot
At first, *pivot* might seem like the more confusing of the two, but its similarity to *groups* is very interesting. We will continue with the same table as before, copied below for your convenience.

Color	Shape	Amount	Price
Red	Round	4	1.30
Green	Rectangular	6	1.20
Blue	Rectangular	12	2.0
Red	Round	7	1.75
Green	Rectangular	9	1.40
Green	Round	2	1.0

We can think of *pivot* as a variation of *groups*. *pivot* takes in at least three things, the first argument is a category which will be unique on the columns, the second is a category which will be unique on the rows, and the last is the argument on which we want to aggregate on. What it does is, it puts each unique value of the first argument in the columns. From there, it puts each unique value of the second argument into rows. Essentially, now, we have a grid of unique row-value pairs. Now, we fill in the unique row-value pairs by aggregating on our third item using our (optional) fourth function. If we don't pass in the fourth function, then it will just count the occurrences like before. Let's see this in action specifically, with a call to `marbles.pivot("Shape", "Color", "Amount", sum)`. It will look like this:

Color	Round Amount	Rectangular Amount
Red	11	0
Green	2	15
Blue	0	12

Notice, each unique value of Colors is in the row (Red, Blue, and Green), while each unique value of Shape is in the columns (Round and Rectangular). In the cell corresponding to red and rectangular, we applied the sum function to all elements in our marbles table which had color Red and Shape round. The sum of 7 and 4 was 11. Then for the Red and Rectangular cell, we took the sum of all of the values that were Red and Rectangular, which was nothing. Luckily, the sum of an empty sequence is 0. Notice the similarity to *groups*, but now, instead of color and shape both being in the rows, one is in the rows and one is in the columns, making a grid. This grid is called a *pivot table*, but for statisticians, it is a *contingency table*.
Now, try to create a pivot table on the type of marble, finding the average price for each type. Assume the *avg* function is given to you, and takes in many things and finds their average.

Join
The last function we will talk about is join. Join takes in three arguments. The first one is a specific column in the initial table, the second argument is another table (could be the same one we are actually on), and the third argument is a specific column in the table in the second argument. What join does is it goes through each row of the first table and checks if there's a matching row in the second table, based on the column labels passed in. If some row in the second table exists, we take the information from the second table and move on, not looking to see if there's any more rows in the second table. If no such row in the second table exists, we do nothing and move on to the next row. Let's try an example call in order to get ourselves ready, more specifically, `marbles.join("Color" ,marbles, "Color")`. What this means is join the marbles table with the marble tables, matching rows that have the same color. The result of this call is:

Color	Shape	Amount	Price	Shape_2	Amount_2	Price_2
Blue	Rectangle	12	2	Rectangle	12	2
Green	Rectangle	6	1.2	Rectangle	6	1.2
Green	Rectangle	9	1.4	Rectangle	6	1.2
Green	Round	2	1	Rectangle	6	1.2
Red	Round	4	1.3	Round	4	1.3
Red	Round	7	1.75	Round	4	1.3

Let's try and walk through this interesting output. So, first we see that the blue row has the information for the only blue row twice, which makes sense as this is the only time the color blue matches. Now, however, we see three rows for Green. What happened here? Let's remember the syntax for join, table.join(first label, other table, second label). As described, what join did was find the first row in the second table, in this case, marbles, in which the two tables matched up in the 'Color' category, and after it found it, it moved on to the next row in the first table. Hence, we will only see each each row in our first table at most once. Remember, it is possible for a row not to appear if nothing equivalent was found in the second table.

So, why is this useful? Assume that there was a sale, and we get different amounts of discount based on how many marbles we are trying to buy from this store. The *sales* table is as follows:

Count	Discount
2	5%
4	10%
6	20%
7	30%
9	35%
12	40%

Now, our goal is to get our computer to match up how much discount is applied to each bunch. Join will do just the trick for us. We will be comparing the amount columns for both tables, so our call will look like `marbles.join('Amount', sales, 'Count')`, and our resulting table will be:

Amount	Color	Shape	Price	Discount
4	Red	Round	1.30	10%
6	Green	Rectangular	1.20	2%
12	Blue	Rectangular	2.0	40%
7	Red	Round	1.75	30%
9	Green	Rectangular	1.40	35%
2	Green	Round	1.0	5%

Perfect!
Discussion Question: Create the table for `marbles.join('Shape', marbles, 'Shape')`