

Introduction to Data and Data Storage

Data primarily is in one of two forms, **numerical** or **categorical**. They are also known as **quantitative** and **qualitative** data respectively. Naively, by the name, you can infer that numerical data has to do with numbers and categorical data has to do with categories. The distinct difference between the two is that numerical data represents data that can be measured while categorical data represents data that can be divided into “groups” or “bins”.

One may think that the difference between the two is that numerical is numbers and categorical are strings. However, categorical variables can also be numbers. The classic example is zip codes. Depending on where you live, you are put in a bin based on that location, a group, which is named by a number. The way you can tell the difference between these two is by asking one of two questions:

1. Is the difference between any two data points meaningful?
2. Is the average of the data meaningful?

If you find that the difference or average is meaningful, then it is numerical, if not then it is categorical. So for zip codes, the difference between where Google was created (94025) and where most Berkeley students live (94704) is -679, which does nothing for us. It doesn't measure anything, it's just a number floating around. You could do the same for the average. I find that asking the first question is the easiest for me to find out what type of data something is.

Examples:

Numerical	Temperature	Temperature is continuous data, and the difference between two points tells us which data point is hotter/colder.
	Number of heads in 100 flips of a coin	This is discrete data, and the average between many of these tests tells us... well the average number of heads, which will be a key feature later in the semester.
Categorical	Zip Codes	As described above, this is categorical data
	Marital Status	You either married or you are not married and those are the two categories.

Tables

Last time, we talked about numpy arrays. We are going to abstract notion and combine many numpy arrays to form a table, where the numpy arrays correspond to a name, and instead of being horizontal, the arrays are vertical. We can demonstrate this below. Below, I have created an array corresponding to the time my local coffee shop is open and another array corresponding to the number of sales during that hour:

```
In [1]: import numpy as np
        times_open = make_array(7, 8, 9, 10, 11, 12, 1, 2, 3, 4, 5)
        people = make_array(30, 70, 90, 85, 62, 34, 77, 61, 53, 42, 22)
```

Then we can make this into a table using methods from the data science library. First we call

```
Table().with_columns(
    <col1 name>, <col1 array>,
    <col2 name>, <col2 array>,
    <col3 name>, <col3 array>, ...
    <coln name>, <coln array>,
)
```

So what this does is it creates a table with at least one column name, up to “n” column names, with each corresponding array as it’s values. Here’s more code to continue from above:

```
In [2]: coffee_shop = Table().with_columns(
        'Time', times_open,
        'Sales', people
    ) # store this table in the variable coffee_shop
    coffee_shop # display coffee_shop
```

```
Out [2]:
```

Time	Sales
7	30
8	70
9	90
10	85
11	62
12	34
1	77
2	61
3	53
4	42
5	22

So in the output above, you can see that the two vertical columns are the arrays with the titles we gave them in the `Table().with_columns` function call.

Now say we accidentally lost our `people` array, so now we don't have an array to know how many sales we made that day. It's ok! Since we still have access to the table, we actually still have the array! This is because tables just consist of multiple numpy arrays next to each other. So the way we get the array back is by using the `<Table Name>.column(<col name>)` method. This will give us the array corresponding to that column name back. Let's see it in action:

```
In [3]: people_again = coffee_shop.column('Sales')
        people_again
Out [3]: array([30, 70, 90, 85, 62, 34, 77, 61, 53, 42, 22])
```

Now say we wanted to add a column to this table. Say a sales representative said that the average sale cost about \$3 and (s)he wanted you to add that to your beautiful new table. Well luckily, we can combine the two things we just learned to do just that! Let's see how to do it:

```
In [4]: coffee_shop = Table().with_columns(
        'Avg Sale', coffee_shop.column('Sales') * 3
    )
        coffee_shop
```

```
Out [4]:
```

	Time	Sales	Avg Sale
	7	30	90
	8	70	210
	9	90	270
	10	85	255
	11	62	186
	12	34	102
	1	77	231
	2	61	183
	3	53	159
	4	42	126
	5	22	66

Now your boss comes to you and says they want to see the most popular times for the table above. Luckily there's a `<Table Name>.sort(<col name>, descending=False)` method that sorts by a column name. By default (in red) it sorts in ascending order. If you set descending to True, then it will sort in descending order. Now let's get to making your boss happy!

```
In [5]: coffee_shop_sorted = coffee_shop.sort('Sales', descending=True)
        coffee_shop_sorted
```

```
Out [5]:
```

Time	Sales	Avg Sale
9	90	270
10	85	255
1	77	231
8	70	210
11	62	186
2	61	183
3	53	159
4	42	126
12	34	102
7	30	90
5	22	66

Your boss thinks you've done good work, however she thinks the table is a little too big. She wants a table of all the hours the store made over \$200. To do this, we can use the where clause. `where` takes in two arguments, a column name and a condition. What the where clause does is it looks at every row and at the specific entry in that row that corresponds to the column. Then it checks to see whether the condition is **True** or **False**. If it's true, then it adds it to the table it will give back to us. If it's **False**, then it won't and skips the row. It does this for all rows.

```
<Table Name>.where(<col name>, <condition (are.____)>)
```

Let's look at the `coffee_shop` table from before and find the rows that are bigger than 200:

Time	Sales	Avg Sale
7	30	90
8	70	210
9	90	270
10	85	255
11	62	186
12	34	102
1	77	231
2	61	183
3	53	159
4	42	126
5	22	66

Returns a table with these rows

Now let's see how we code this:

```
In [6]: coffee_shop.where('Avg Sale', are.above_or_equal_to(200))
```

```
Out [6]:
```

Time	Sales	Avg Sale
8	70	210
9	90	270
10	85	255
1	77	231

There is a whole list of conditional statements you can use as the second argument in the where clause. They can be found on your lab, but I'll copy and paste it in here for convenience.

Predicate	Example	Result
<code>are.equal_to</code>	<code>are.equal_to(50)</code>	Find rows with values equal to 50
<code>are.not_equal_to</code>	<code>are.not_equal_to(50)</code>	Find rows with values not equal to 50
<code>are.above</code>	<code>are.above(50)</code>	Find rows with values above (and not equal to) 50
<code>are.above_or_equal_to</code>	<code>are.above_or_equal_to(50)</code>	Find rows with values above 50 or equal to 50
<code>are.below</code>	<code>are.below(50)</code>	Find rows with values below 50
<code>are.between</code>	<code>are.between(2, 10)</code>	Find rows with values above or equal to 2 and below 10

Now, your friend comes up to you and asks how many of those hours made over \$200. Well, we already know how to make a table listing out the times that made over \$200. All we need to know is how many rows there are. Now I know this example is contrived and you can easily count the number of rows, but later in projects you'll have tables with 100's of rows, and it's easier just to use the `<Table Name>.num_rows` method! Here's how we can code it:

```
In [7]: coffee_shop.where('Avg Sale', are.above(199)).num_rows
        # the line above identical to the Output[6] above
```

```
Out [7]: 4
```

There are so many more table manipulation methods you can use. `select` is another big one, that works like `column`, except returns a table and can be used to return multiple columns at once. Remember, `.column` returns a numpy array, `.select` returns a table. Take a look at the datascience handbook for more! The ones I mentioned above are some of the most common and the most useful I've found.