```
In [ ]: # Initialize Otter
        import otter
        grader = otter.Notebook("lab02.ipynb")
```

```
In [1]: # Run this cell to load all required Python libraries
        exec(open("./utils.py").read())
```

1 Lab 2 – Tables and Data Manipulation

1.1 Data 6, Summer 2022

In this lab, we will be talking all about *Tables*. We use tables to store all sorts of data from sports statistics to population information. If there's data you have ever been curious about, it is very likely that the Internet has a table somewhere with that data!

Tables are integral to the foundation of Data Science, and we will go over how to **query** a table. **Querying** a table is, simply put, requesting information about the table. Some examples of common queries (in English, not code):

- How many data points are there?
- Which data points have a specific characteristic?
- What is the attribute of a specific data point?
- And many more!

There are so many ways we can use tables to get information we need, and there are several existing libraries in Python that we can use to do this! In this course, we will be using the `datascience` library. This is the standard library used both in Data 6 and Data 8 at UC Berkeley. If you take Data Science classes beyond those two, you'll learn more!

1.2 Table Creation

Let's take a look at a table in action. Python does not have any tables by default, so we can either *create a new table from scratch* or *import a table from a file*. First, let's see how we can make our own table from scratch.

We start out with an empty `Table` – this is the same idea as having an empty array or string. Note that `Table` is capitalized and there is nothing in the parentheses.

```
In [2]: # Run this cell to load an empty table
```

```
        our_table = Table()
        our_table
```

```
Out[2]:
```

1.3 Adding Data: `with_columns`

Now, let's put some data in our table! To do so, we use the `with_columns` method. This method requires **two arguments**: 1. The name of the column as a string 2. An array of values to put in the column

An example call to `with_columns` looks like: `my_tbl.with_columns("My New Column", my_array)`, where `my_tbl` is our table that we would like to add to and `my_array` is a previously-defined array.

Run the cell below to see how we can add multiple columns into `our_table`.

```
In [3]: our_table = our_table.with_columns(
            "Department", make_array("Data Science", "Economics", "Political Science", "Sociology"),
            "Course Number", make_array(6, 1, 2, 121)
        )
        our_table
```

```
Out[3]: Department        | Course Number
        Data Science      | 6
        Economics         | 1
        Political Science | 2
        Sociology         | 121
```

We need to make sure that the columns we add to the table have the same number of rows (the length of the array we pass in) as the table. Otherwise, we'll get an error.

Watch what happens if we try to add a new column that doesn't have enough data (you'll see an error!)

```
In [4]: # Just run this cell
        our_table.with_columns("Too Few Rows", np.array([1, 2, 3]))
```

```
---------------------------------------------------------------------------
ValueError                                Traceback (most recent call last)
Input In [4], in <cell line: 2>()
      1 # Just run this cell
----> 2 our_table.with_columns("Too Few Rows", np.array([1, 2, 3]))
```

```
File ~/.pyenv/versions/3.9.11/lib/python3.9/site-packages/datascience/tables.py:2593, i Table.with_co
   2591 for i in range(0, len(labels_and_values), 2):
   2592     label, values = labels_and_values[i], labels_and_values[i+1]
-> 2593     self = self.with_column(label, values, formatter)
   2594 return self

File ~/.pyenv/versions/3.9.11/lib/python3.9/site-packages/datascience/tables.py:2510, i Table.with_co
   2508 elif isinstance(formatter, dict):
   2509     formatter = formatter["formatter"]
-> 2510 new_table.append_column(label, values, formatter)
   2511 return new_table

File ~/.pyenv/versions/3.9.11/lib/python3.9/site-packages/datascience/tables.py:1002, i Table.append_
    999     values = np.array(tuple(values))
   1001 if self.num_rows != 0 and len(values) != self.num_rows:
-> 1002     raise ValueError('Column length mismatch. New column does not have '
   1003                      'the same number of rows as table.')
   1004 else:
   1005     self._num_rows = len(values)

ValueError: Column length mismatch. New column does not have the same number of rows as table.
```

Question 1.1: Add a new column to `our_table` called `"Number of Students"` that contains the number of students in each department. James tells you that the *Data Science* department has 240 students, the *Economics* department has 905 students, the *Political Science* Department has 209 students, and the *Sociology* department has 63 students.

Assign this new table to the variable `our_table_new_column`

```
In [5]: # Our table has 4 rows, so our new column needs an array with 4 items, 1 for each row
        students_array = make_array(240, 905, 209, 63) # SOLUTION
        our_table_new_column = our_table.with_columns("Number of Students", students_array) # SOLUTION
        our_table_new_column
```

```
Out[5]: Department        | Course Number | Number of Students
        Data Science      | 6             | 240
        Economics         | 1             | 905
        Political Science | 2             | 209
        Sociology         | 121           | 63
```

```
In [ ]: grader.check("q1_1")
```

```
In [8]: # This is our final table!
        # You may use this cell to explore the table and see what you can do with it so far!
        our_table_new_column
```

```
Out[8]: Department        | Course Number | Number of Students
        Data Science      | 6             | 240
        Economics         | 1             | 905
        Political Science | 2             | 209
        Sociology         | 121           | 63
```

1.4 Table attributes: `num_rows` and `num_columns`

We can ask for all sorts of information about the table itself:

```
In [9]: our_table_new_column.num_rows
```

```
Out[9]: 4
```

```
In [10]: our_table_new_column.num_columns
```

```
Out[10]: 3
```

1.5 Accessing Data: `column`

We can also ask about the data in the table using the `column` method. As mentioned in lecture, we can pass in a `label` or an `index` to this method.

Note: Recall that index into the columns of a table using **zero-indexing** – `0` corresponds to the first column, `1` corresponds to the second, etc….

```
In [11]: # Converts a column in the table to an array
         our_table_new_column.column("Department")
```

```
Out[11]: array(['Data Science', 'Economics', 'Political Science', 'Sociology'],
               dtype='<U17')
```

```
In [12]: # Same thing, but uses column index instead of label
         our_table_new_column.column(0)
```

```
Out[12]: array(['Data Science', 'Economics', 'Political Science', 'Sociology'],
               dtype='<U17')
```

Question 1.2: Find the average number of students in each department by first accessing the `"Number of Students"` column as an array and then taking the average. Assign the average to `avg_num_students`.

Note: You may use any `np` functions here

```
In [13]: avg_num_students = np.mean(our_table_new_column.column("Number of Students")) # SOLUTION
         avg_num_students
```

```
Out[13]: 354.25
```

```
In [ ]: grader.check("q1_2")
```

1.6 Loading a Table

Although creating our own tables by hand can be useful, more often than not the data we want to work with is far too large to be able to type out by hand. More commonly, we load datasets in from other sources using the `Table.read_table()` method. We can pass in a *file path* to this method and it will load that data into a table we can use in Python!

1.6.1 Background on the Data

The dataset that we'll use in this lab comes from the Behavioral Risk Factor Surveillance System (BRFSS), a health survey fielded by the Centers for Disease Control and Prevention (CDC). From the BRFSS website: >The Behavioral Risk Factor Surveillance System (BRFSS) is the nation's premier system of health-related telephone surveys that collect state data about U.S. residents regarding their health-related risk behaviors, chronic health conditions, and use of preventive services.

> By collecting behavioral health risk data at the state and local level, BRFSS has become a powerful tool for targeting and building health promotion activities.

The dataset that you will investigate is a **subset of the 2022 BRFSS Survey**. We've taken all the data points corresponding to fully-completed surveys and, in our opinion, the most interesting columns. Since the entire data set is so large, we've randomly sampled 100,000 rows from the original data. While we've wrangled and cleaning the data set you'll use in your investigation, you're welcome to investigate the original source; you can do so via the Survey Data section of the BRFSS site.

1.7 Seeing a table: `show()`

The use of the `show()` method **displays** the first n rows of a table. Like `print()` this does not return a value, it just displays the value to us at the end of a cell.

```
In [15]: # Run this cell to load in the data
         brfss = Table.read_table("data/brfss.csv")
         brfss.show(5)
```

```
<IPython.core.display.HTML object>
```

Question 2.1: Fill in the `num_rows_brfss` and `num_columns_brfss` with the number of rows and columns in the original `brfss` table, respectively.

```
In [16]: num_rows_brfss = brfss.num_rows # SOLUTION
         num_columns_brfss = brfss.num_columns # SOLUTION
         print(f"Our `brfss` table has {num_rows_brfss} rows and {num_columns_brfss} columns")
```

```
Our `brfss` table has 100000 rows and 23 columns
```

```
In [ ]: grader.check("q2_1")
```

1.7.1 Investigating our Data

Now that we've loaded our data into the `brfss` table, let's take a closer look at its columns. Run the following cell to output the column names.

```
In [19]: brfss.labels
```

```
Out[19]: ('State',
          'Day',
          'Month',
          'Year',
          'Cell Phone',
          'College Housing (Cell)',
          'College Housing (Landline)',
          'General Health',
          'Physical Health',
          'Mental Health',
          'Personal Doctor',
          'Health Plan',
          'Sleep Time',
          'Exercise',
          'Diabetes',
          'Diabetes Age',
          'Binge Drinks',
          'Sex',
          'Children',
          '100 Cigarettes',
          'Days Smoking',
          'Income Lower',
          'Income Upper')
```

Based on these column names, it looks like the data includes questions about **telecommunications**, **housing**, **demographic information**, **mental and physical health**, **alcohol and drug consumption**, and **physical exercise**. Each column in the `brfss` table corresponds to a question asked in the official BRFSS Survey.

1.8 Data Manipulation

Now that we have a solid understanding of the basic table methods from the `datascience` library, let's put our skills to use! Even with a few tools, we are already able to arrive at powerful realizations about real world data.

Question 2.2: Assign `num_alabama_rows` to the the number of times the name **Alabama** appeared in the `brfss` table.

Hint: Use the table methods you've learned above!

```
In [20]: alabama_tbl = brfss.where("State", "Alabama") # SOLUTION
         num_alabama_rows = alabama_tbl.num_rows # SOLUTION
         num_alabama_rows
```

```
Out[20]: 1352
```

```
In [ ]: grader.check("q2_2")
```

Take a closer look at some of the columns in the `brfss` table. For the next two questions, we will be looking at the `"Binge Drinks"` column, which corresponds to this survey question: > Considering all types of alcoholic beverages, how many times during the past 30 days did you have 5 or more drinks for men or 4 or more drinks for women on an occasion?

Notice that this column contains negative values, most notably `-1`. Why might this be the case? Discuss with the people around you and check in with your GSI to confirm.

```
In [22]: # Run this cell
         brfss.column("Binge Drinks")
```

```
Out[22]: array([ 0., -1., -1., …,  0.,  0.,  0.])
```

Question 2.3: Create a new table called `missing_binge_drinks` which only contains rows from the `brfss` table where there is a `-1` in the `"Binge Drinks"` column.

```
In [23]: missing_binge_drinks = brfss.where("Binge Drinks", -1) # SOLUTION
         missing_binge_drinks
```

```
Out[23]: State          | Day  | Month | Year | Cell Phone | College Housing (Cell) | College Housing (
         West Virginia  | 1    | 6     | 2020 | Missing    | Missing                | Missing
         Georgia        | 0    | 6     | 2020 | Missing    | Missing                | Missing
         Florida        | 2    | 8     | 2020 | Missing    | Missing                | Missing
         Maine          | 0    | 10    | 2020 | Missing    | Missing                | Missing
         Arizona        | 0    | 4     | 2020 | Yes        | Missing                | Missing
         Texas          | 1    | 7     | 2020 | Yes        | Missing                | Missing
         South Carolina | 0    | 11    | 2020 | Missing    | Missing                | Missing
         Maine          | 2    | 2     | 2020 | Missing    | Missing                | Missing
         Ohio           | 1    | 9     | 2020 | Yes        | Missing                | Missing
         Massachusetts  | 1    | 2     | 2020 | Yes        | Missing                | Missing
         … (51032 rows omitted)
```

```
In [ ]: grader.check("q2_3")
```

Question 2.4 (*Discussion*): Say we wanted to find the average of one of the columns from our original table. How does the inclusion of `-1` values *affect* this average? If we removed all the negative values, how would the average change?

Then, once you've answered, run the following cell to confirm your understanding.

Type your answer here, replacing this text.

SOLUTION: The average is lower with negative values. If we removed them, the average would be higher.

```
In [25]: brfss_no_negatives_children = brfss.where("Children", are.not_equal_to(-1))
         print(f"With negatives: {np.average(brfss.column('Children'))}")
         print(f"Without negatives: {np.average(brfss_no_negatives_children.column('Children'))}")
```

```
With negatives: 0.49044
Without negatives: 0.5016119932296285
```

Question 2.5: As an ***extra challenge***, see if you can output Alabama's average number of children for all respondents who gave an answer (i.e. do not have a value of -1) **without using the `np.average` function**. You should use the table `brfss_no_negatives_children` that's defined in the cell above.

Note: We've provided some starter code. Feel free to use it or try another approach!

```
In [26]: alabama_tbl = brfss_no_negatives_children.where("State", "Alabama") # SOLUTION
         children_total = sum(alabama_tbl.column("Children")) # SOLUTION
         average_num_children = children_total / alabama_tbl.num_rows # SOLUTION
         average_num_children
```

```
Out[26]: 0.5443698732289336
```

```
In [ ]: grader.check("q2_5")
```

Question 2.6: Using the `no_missing_income` table we've provided for you, determine the **average income** for respondents who: 1. Have a health insurance plan ("Yes" in "Health Plan" column) 2. Do not have a health insurance plan ("No" in "Health Plan" column) 3. Refused to answer this question ("Declined to Answer" in "Health Plan" column)

You may use the starter code provided for you, but are not required to.

```
In [28]: no_missing_income = brfss.where("Income Upper", are.not_equal_to(-1))

         health_plan = no_missing_income.where("Health Plan", "Yes") # SOLUTION
         no_health_plan = no_missing_income.where("Health Plan", "No") # SOLUTION
         declined = no_missing_income.where("Health Plan", "Declined to Answer") # SOLUTION

         average_plan = np.mean(health_plan.column("Income Upper")) # SOLUTION
         average_no_plan = np.mean(no_health_plan.column("Income Upper")) # SOLUTION
         average_declined = np.mean(declined.column("Income Upper")) #SOLUTION

         print(f"Respondents with a Health Insurance Plan made: \t\t${round(average_plan, 2)}")
         print(f"Respondents without a Health Insurance Plan made: \t${round(average_no_plan, 2)}")
         print(f"Respondents who refused to answer made: \t\t${round(average_declined, 2)}")
```

```
Respondents with a Health Insurance Plan made:                 $43832.35
Respondents without a Health Insurance Plan made:         $34940.18
Respondents who refused to answer made:                  $30963.91
```

```
In [ ]: grader.check("q2_6")
```

With just over a week of data science under your belts, you're already able to uncover underlying **patterns and trends** within real world data. In this case, we've found that those with health insurance plans make, on average, almost 9,000 more dollars a year than those without health insurance plans.

What's valuable, too, is the information we gain when a population is *missing* or, in this case, *declines to participate*. Notice that the average salary of respondents who refused to answer was around 4,000 dollars lower than those *without* a Health Insurance Plan.

1.9 Done!

That's it! There's nowhere for you to submit this, as labs are not assignments. However, please ask any questions you have with this notebook in lab or on Ed.

To double-check your work, the cell below will rerun all of the autograder tests.

```
In [ ]: grader.check_all()
```

1.10 Submission

Make sure you have run all cells in your notebook in order before running the cell below, so that all images/graphs appear in the output. The cell below will generate a zip file for you to submit. **Please save before exporting!**

```
In [ ]: # Save your notebook first, then run this cell to export your submission.
        grader.export(pdf=False)
```