## CONTENTS

1 **Start Here: data science Tutorial**  
   1.1 Getting Started .......................................................... 3  
   1.2 Creating a Table ...................................................... 4  
   1.3 Accessing Values .................................................... 5  
   1.4 Manipulating Data .................................................. 7  
   1.5 Visualizing Data .................................................... 10  
   1.6 Exporting ............................................................. 14  
   1.7 An Example ........................................................... 15  
   1.8 Drawing Maps ......................................................... 18  

2 **Data 8 data science Reference**  
   2.1 Table Functions and Methods .................................... 19  
   2.2 Table Visualizations ................................................ 28  
   2.3 Advanced Table Functions ......................................... 34  
   2.4 String Methods ....................................................... 39  
   2.5 Array Functions and Methods .................................... 41  
   2.6 Table.where Predicates ............................................ 46  
   2.7 Miscellaneous Functions .......................................... 55  

3 **Reference**  
   3.1 Tables (data science.tables) ...................................... 57  
   3.2 Maps (data science.maps) .......................................... 118  
   3.3 Predicates (data science.predicates) ............................ 122  
   3.4 Formats (data science.formats) .................................. 125  
   3.5 Utility Functions (data science.util) ............................ 126  

**Python Module Index** .................................................. 131  

**Index** ...................................................................... 133
The `datascience` package was written for use in Berkeley’s DS 8 course and contains useful functionality for investigating and graphically displaying data.
This is a brief introduction to the functionality in datascience. For a complete reference guide, please see *Tables* ([datascience.tables](#)).

For other useful tutorials and examples, see:

- The textbook introduction to Tables
- Example notebooks

### Table of Contents

- Getting Started
- Creating a Table
- Accessing Values
- Manipulating Data
- Visualizing Data
- Exporting
- An Example
- Drawing Maps

#### 1.1 Getting Started

The most important functionality in the package is is the `Table` class, which is the structure used to represent columns of data. First, load the class:

```
In [1]: from datascience import Table
```

In the IPython notebook, type `Table` followed by the TAB-key to see a list of members.

Note that for the Data Science 8 class we also import additional packages and settings for all assignments and labs. This is so that plots and other available packages mirror the ones in the textbook more closely. The exact code we use is:

```
# HIDDEN
import matplotlib
```

(continues on next page)
In particular, the lines involving `matplotlib` allow for plotting within the IPython notebook.

### 1.2 Creating a Table

A Table is a sequence of labeled columns of data.

A Table can be constructed from scratch by extending an empty table with columns.

```python
In [2]: t = Table().with_columns(
    "letter", ['a', 'b', 'c', 'z'],
    "count", [9, 3, 3, 1],
    "points", [1, 2, 2, 10],
)

In [3]: print(t)
letter | count | points
a | 9 | 1
b | 3 | 2
c | 3 | 2
z | 1 | 10
```

More often, a table is read from a CSV file (or an Excel spreadsheet). Here's the content of an example file:

```python
In [4]: cat sample.csv
x,y,z
1,10,100
2,11,101
3,12,102
```

And this is how we load it in as a Table using `read_table()`:

```python
In [5]: Table.read_table('sample.csv')
```

```python
Out[5]:
x | y | z
1 | 10 | 100
2 | 11 | 101
3 | 12 | 102
```

CSVs from URLs are also valid inputs to `read_table()`:

```python
In [6]: Table.read_table('https://www.inferentialthinking.com/data/sat2014.csv')
```

```python
Out[6]:
x | y | z
1 | 10 | 100
2 | 11 | 101
3 | 12 | 102
```
State | Participation Rate | Critical Reading | Math | Writing | Combined
--- | --- | --- | --- | --- | ---
North Dakota | 2.3 | 612 | 620 | 584 | 1816
Illinois | 4.6 | 599 | 616 | 587 | 1802
Iowa | 3.1 | 605 | 611 | 578 | 1794
South Dakota | 2.9 | 604 | 609 | 579 | 1792
Minnesota | 5.9 | 598 | 610 | 578 | 1786
Michigan | 3.8 | 593 | 610 | 581 | 1784
Wisconsin | 3.9 | 596 | 608 | 578 | 1782
Missouri | 4.2 | 595 | 597 | 579 | 1771
Wyoming | 3.3 | 590 | 599 | 573 | 1762
Kansas | 5.3 | 591 | 596 | 566 | 1753
... (41 rows omitted)

It's also possible to add columns from a dictionary, but this option is discouraged because dictionaries do not preserve column order.

```
In [7]: t = Table().with_columns(
   ...:   'letter': ['a', 'b', 'c', 'z'],
   ...:   'count': [9, 3, 3, 1],
   ...:   'points': [1, 2, 2, 10],
   ...: )
   ...
```

```
In [8]: print(t)
letter | count | points
----- | ----- | -----
a     | 9     | 1
b     | 3     | 2
c     | 3     | 2
z     | 1     | 10
```

### 1.3 Accessing Values

To access values of columns in the table, use `column()`, which takes a column label or index and returns an array. Alternatively, `columns()` returns a list of columns (arrays).

```
In [9]: t
Out[9]:
letter | count | points
----- | ----- | -----
a     | 9     | 1
b     | 3     | 2
c     | 3     | 2
z     | 1     | 10
```

```
In [10]: t.column('letter')
Out[10]:
array(['a', 'b', 'c', 'z'],
      dtype='<U1')
```
You can use bracket notation as a shorthand for this method:

```
In [12]: t['letter']  # This is a shorthand for t.column('letter')
Out[12]:
array(['a', 'b', 'c', 'z'],
      dtype='<U1')  
```

```
In [13]: t[1]  # This is a shorthand for t.column(1)
Out[13]:
array([9, 3, 3, 1])
```

To access values by row, `row()` returns a row by index. Alternatively, `rows()` returns an list-like `Rows` object that contains tuple-like `Row` objects.

```
In [14]: t.rows
Out[14]:
Rows(letter | count | points
   a | 9 | 1
   b | 3 | 2
   c | 3 | 2
   z | 1 | 10)
```

```
In [15]: t.rows[0]
Out[15]: Row(letter='a', count=9, points=1)
```

```
In [16]: t.row(0)
Out[16]: Row(letter='a', count=9, points=1)
```

```
In [17]: second = t.rows[1]
In [18]: second
Out[18]: Row(letter='b', count=3, points=2)
```

```
In [19]: second[0]
Out[19]: 'b'
```

```
In [20]: second[1]
Out[20]: 3
```

To get the number of rows, use `num_rows`.

```
In [21]: t.num_rows
Out[21]: 4
```
1.4 Manipulating Data

Here are some of the most common operations on data. For the rest, see the reference (Tables (datascience.tables)).

Adding a column with \texttt{with\_column()}:

\begin{verbatim}
In [22]: t
Out[22]:
letter | count | points
a     | 9     | 1
b     | 3     | 2
z     | 1     | 10

In [23]: t.with_column('vowel?', ['yes', 'no', 'no', 'no'])
Out[23]:
letter | count | points | vowel?
       |       |        | yes
a     | 9     | 1      |
       | 3     | 2      |
b     | 3     | 2      | no
       | 1     | 10     | no
z     | 1     | 10     | no

In [24]: t # .with\_column returns a new table without modifying the original
Out[24]:
letter | count | points
a     | 9     | 1
b     | 3     | 2
z     | 1     | 10

In [25]: t.with_column('2 * count', t['count'] * 2) # A simple way to operate on columns
Out[25]:
letter | count | points | 2 * count
       |       |        | 18
a     | 9     | 1      |
       | 3     | 2      |
b     | 3     | 2      |
z     | 1     | 10     |

Selecting columns with \texttt{select()}:

\begin{verbatim}
In [26]: t.select('letter')
Out[26]:
letter
a
b
c
z

In [27]: t.select(['letter', 'points'])
Out[27]:
letter | points
a     | 1
b     | 2

(continues on next page)
Renaming columns with \texttt{relabeled()}:

\begin{verbatim}
In [28]: t
Out[28]:
letter | count | points
a   | 9     | 1
b   | 3     | 2
c   | 3     | 2
z   | 1     | 10

In [29]: t.relabeled('points', 'other name')
Out[29]:
letter | count | other name
a     | 9     | 1
b     | 3     | 2
c     | 3     | 2
z     | 1     | 10

In [30]: t
Out[30]:
letter | count | points
a     | 9     | 1
b     | 3     | 2
c     | 3     | 2
z     | 1     | 10

In [31]: t.relabeled(['letter', 'count', 'points'], ['x', 'y', 'z'])
Out[31]:
x | y | z
a | 9 | 1
b | 3 | 2
c | 3 | 2
z | 1 | 10
\end{verbatim}

Selecting out rows by index with \texttt{take()} and conditionally with \texttt{where()}:

\begin{verbatim}
In [32]: t
Out[32]:
letter | count | points
a     | 9     | 1
b     | 3     | 2
c     | 3     | 2
z     | 1     | 10

In [33]: t.take(2) \# the third row
Out[33]:
letter | count | points
a     | 9     | 1
b     | 3     | 2
c     | 3     | 2
\end{verbatim}
In [34]: t.take[0:2] # the first and second rows
Out[34]:
   letter | count | points
  ------ | ---- | -----
    a    |   9  |    1
    b    |   3  |    2

In [35]: t.where('points', 2) # rows where points == 2
Out[35]:
   letter | count | points
   b      |   3   |    2
   c      |   3   |    2

In [36]: t.where(t['count'] < 8) # rows where count < 8
Out[36]:
   letter | count | points
   b      |   3   |    2
   c      |   3   |    2
   z      |   1   |  10

In [37]: t['count'] < 8 # .where actually takes in an array of booleans
Out[37]: array([False, True, True, True], dtype=bool)

In [38]: t.where([False, True, True, True]) # same as the last line
Out[38]:
   letter | count | points
   b      |   3   |    2
   c      |   3   |    2
   z      |   1   | 10

Operate on table data with sort(), group(), and pivot()

In [39]: t
Out[39]:
   letter | count | points
  ------ | ---- | -----
    a    |   9  |    1
    b    |   3  |    2
    c    |   3  |    2
    z    |   1  |  10

In [40]: t.sort('count')
Out[40]:
   letter | count | points
    z    |   1   | 10
    b    |   3   |    2
    c    |   3   |    2
    a    |   9   |    1

In [41]: t.sort('letter', descending = True)
Out[41]:
   letter | count | points
    z    |   1   | 10
    c    |   3   |    2

1.4. Manipulating Data
# You may pass a reducing function into the collect arg
# Note the renaming of the points column because of the collect arg
In [42]: t.select(['count', 'points']).group('count', collect=sum)
Out[42]:
count | points sum
1     | 10
3     | 4
9     | 1

In [43]: other_table = Table().with_columns(
        ....:   'mar_status', ['married', 'married', 'partner', 'partner', 'married'],
        ....:   'empl_status', ['Working as paid', 'Working as paid', 'Not working',
        ....:                     'Not working', 'Not working'],
        ....:   'count', [1, 1, 1, 1, 1])

In [44]: other_table
Out[44]:
mar_status | empl_status | count
married    | Working as paid | 1
married    | Working as paid | 1
partner    | Not working    | 1
partner    | Not working    | 1
married    | Not working    | 1

In [45]: other_table.pivot('mar_status', 'empl_status', 'count', collect=sum)
Out[45]:
empl_status | married | partner
Not working | 1       | 2
Working as paid | 2       | 0

1.5 Visualizing Data

We’ll start with some data drawn at random from two normal distributions:

In [46]: normal_data = Table().with_columns(
        ....:   'data1', np.random.normal(loc = 1, scale = 2, size = 100),
        ....:   'data2', np.random.normal(loc = 4, scale = 3, size = 100))

In [47]: normal_data
Out[47]:
data1    | data2
1.04138  | -1.29895
3.1199   | 5.46409
-1.82543 | 3.04532
-2.41073 | 5.14941
3.77705 | 9.28869
2.97211 | -0.475766
3.41361 | 9.5635
4.28513 | 5.05362
3.58374 | 1.77316
-1.89773 | -1.13326
... (90 rows omitted)

Draw histograms with `hist()`:

**In [48]:** normal_data.hist()

![Histogram of normal_data](image1)

**In [49]:** normal_data.hist(bins = range(-5, 10))

![Histogram with custom bins](image2)

**In [50]:** normal_data.hist(bins = range(-5, 10), overlay = True)

![Overlay histograms](image3)
If we treat the `normal_data` table as a set of x-y points, we can `plot()` and `scatter()`:

```python
In [51]: normal_data.sort('data1').plot('data1')  # Sort first to make plot nicer
```

```python
In [52]: normal_data.scatter('data1')
```
In [53]: normal_data.scatter('data1', fit_line = True)

Use `barh()` to display categorical data.
In [54]: t
Out[54]:
letter | count | points
a     | 9     | 1
b     | 3     | 2
c     | 3     | 2
z     | 1     | 10

In [55]: t.barh('letter')

1.6 Exporting

Exporting to CSV is the most common operation and can be done by first converting to a pandas dataframe with `to_df()`:

In [56]: normal_data
Out[56]:
data1   | data2
1.04138 | -1.29895
3.1199  | 5.46409
-1.82543| 3.04532
-2.41073| 5.14941
3.77705 | 9.28869
2.97211 | -0.475766
3.41361 | 9.5635
4.28513 | 5.05362
3.58374 | 1.77316
-1.89773| -1.13326
... (90 rows omitted)

# index = False prevents row numbers from appearing in the resulting CSV
In [57]: normal_data.to_df().to_csv('normal_data.csv', index = False)
1.7 An Example

We’ll recreate the steps in Chapter 12 of the textbook to see if there is a significant difference in birth weights between smokers and non-smokers using a bootstrap test.

For more examples, check out the TableDemos repo.

From the text:

The table baby contains data on a random sample of 1,174 mothers and their newborn babies. The column Birth Weight contains the birth weight of the baby, in ounces; Gestational Days is the number of gestational days, that is, the number of days the baby was in the womb. There is also data on maternal age, maternal height, maternal pregnancy weight, and whether or not the mother was a smoker.

```
In [58]: baby = Table.read_table('https://www.inferentialthinking.com/data/baby.csv')
```

```
In [59]: baby
```

```
Out[59]:
Birth Weight | Gestational Days | Maternal Age | Maternal Height | Maternal Pregnancy Weight | Maternal Smoker
120          | 284              | 27           | 62              | 100                      | False
113          | 282              | 33           | 64              | 135                      | False
128          | 279              | 28           | 64              | 115                      | True
108          | 282              | 23           | 67              | 125                      | True
136          | 286              | 25           | 62              | 93                       | False
138          | 244              | 33           | 62              | 178                      | False
132          | 245              | 23           | 65              | 140                      | False
120          | 289              | 25           | 62              | 125                      | False
143          | 299              | 30           | 66              | 136                      | True
140          | 351              | 27           | 68              | 120                      | False
... (1164 rows omitted)
```

# Select out columns we want.
```
In [60]: smoker_and_wt = baby.select(['Maternal Smoker', 'Birth Weight'])
```

```
In [61]: smoker_and_wt
```

```
Out[61]:
Maternal Smoker | Birth Weight
False           | 120
False           | 113
True            | 128
True            | 108
False           | 136
False           | 138
```

(continues on next page)
Let’s compare the number of smokers to non-smokers.

```
In [62]: smoker_and_wt.select('Maternal Smoker').group('Maternal Smoker')
Out[62]:
Maternal Smoker | count
False   | 715
True    | 459
```

We can also compare the distribution of birthweights between smokers and non-smokers.

```
# Non smokers
# We do this by grabbing the rows that correspond to mothers that don't
# smoke, then plotting a histogram of just the birthweights.
In [63]: smoker_and_wt.where('Maternal Smoker', 0).select('Birth Weight').hist()

# Smokers
In [64]: smoker_and_wt.where('Maternal Smoker', 1).select('Birth Weight').hist()
```
What’s the difference in mean birth weight of the two categories?

```
In [65]: nonsmoking_mean = smoker_and_wt.where('Maternal Smoker', 0).column('Birth Weight').mean()
In [66]: smoking_mean = smoker_and_wt.where('Maternal Smoker', 1).column('Birth Weight').mean()
In [67]: observed_diff = nonsmoking_mean - smoking_mean
In [68]: observed_diff
Out[68]: 9.2661425720249184
```

Let’s do the bootstrap test on the two categories.

```
In [69]: num_nonsmokers = smoker_and_wt.where('Maternal Smoker', 0).num_rows
In [70]: def bootstrap_once():
   ....:     """
   ....:     Computes one bootstrapped difference in means.
   ....:     The table.sample method lets us take random samples.
   ....:     We then split according to the number of nonsmokers in the original sample.
   ....:     """
   ....:     resample = smoker_and_wt.sample(with_replacement = True)
   ....:     bootstrap_diff = resample.column('Birth Weight')[num_nonsmokers:].mean() - \
   ....:                     resample.column('Birth Weight')[num_nonsmokers:].mean()
   ....:     return bootstrap_diff
   ....:
In [71]: repetitions = 1000
In [72]: bootstrapped_diff_means = np.array([ bootstrap_once() for _ in range(repetitions) ])
In [73]: bootstrapped_diff_means[:10]
Out[73]:
```

(continues on next page)
array([-1.61099685,  1.33708427,  0.24726602, -0.73273002, -1.07159681,  
       0.10865823,  0.6345872 , -1.67601201,  0.49538522,  1.43694867])

In [74]: num_diffs_greater = (abs(bootstrapped_diff_means) > abs( observed_diff)).sum()

In [75]: p_value = num_diffs_greater / len(bootstrapped_diff_means)

In [76]: p_value
Out[76]: 0.0

1.8 Drawing Maps

To come.
This notebook serves as an interactive, Data 8-friendly reference for the datascience library.

## 2.1 Table Functions and Methods

### 2.1.1 Table()

Create an empty table, usually to extend with data

```
[29]: new_table = Table()
new_table
```

```
[30]: type(new_table)
```

```
[30]: datascience.tables.Table
```

### 2.1.2 Table.read_table()

Table.read_table(filename)

Creates a table by reading the CSV file named filename (a string).

```
[31]: trips = Table.read_table('https://raw.githubusercontent.com/data-8/textbook/gh-pages/˓
data/trip.csv')
trips
```

<table>
<thead>
<tr>
<th>Trip ID</th>
<th>Duration</th>
<th>Start Date</th>
<th>Start Station</th>
<th>Start Terminal</th>
<th>End Date</th>
<th>End Station</th>
<th>End Terminal</th>
</tr>
</thead>
<tbody>
<tr>
<td>876419</td>
<td>413</td>
<td>8/5/2015 8:29</td>
<td>Civic Center BART (7th at Market)</td>
<td>72</td>
<td>8/5/2015 8:36</td>
<td>Townsend at 7th</td>
<td>65</td>
</tr>
<tr>
<td>459672</td>
<td>408</td>
<td>9/18/2014 17:11</td>
<td>Harry Bridges Plaza (Ferry Building)</td>
<td>50</td>
<td>9/18/2014 17:17</td>
<td>Embarcadero at Sansome</td>
<td>60</td>
</tr>
<tr>
<td>903647</td>
<td>723</td>
<td>8/25/2015 7:26</td>
<td>San Francisco Caltrain 2 (330 Townsend)</td>
<td>69</td>
<td>8/25/2015 7:38</td>
<td>Market at 10th</td>
<td>67</td>
</tr>
</tbody>
</table>
### 2.1.3 tbl.with_column

```
tbl = Table()
tbl.with_column(name, values)
tbl.with_columns(n1, v1, n2, v2,...)
```

Creates a new table by adding a column with name `name` and values `values` to another table. `name` should be a string and `values` should have as many entries as there are rows in the original table. If `values` is a single value, then every row of that column has the value.

In the examples below, we start with adding a column to the existing table `trips` with values being an array we construct from existing tables.

```
[32]: trips.with_column(
    "Difference in terminal", abs(trips.column("Start Terminal") - trips.column("End Terminal"))
)
```

```
<table>
<thead>
<tr>
<th>Trip ID</th>
<th>Duration</th>
<th>Start Date</th>
<th>Start Station</th>
<th>Start Terminal</th>
<th>End Date</th>
<th>End Station</th>
<th>End Terminal</th>
</tr>
</thead>
<tbody>
<tr>
<td>876419</td>
<td>413</td>
<td>8/5/2015</td>
<td>Civic Center BART (7th at Market)</td>
<td>72</td>
<td>8/5/2015</td>
<td>Townsend at 7th</td>
<td>65</td>
</tr>
<tr>
<td>269</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td>94518</td>
<td></td>
<td></td>
</tr>
<tr>
<td>459672</td>
<td>408</td>
<td>9/18/2014</td>
<td>Harry Bridges Plaza (Ferry Building)</td>
<td>50</td>
<td>9/18/2014</td>
<td>Embarcadero at Sansome</td>
<td>60</td>
</tr>
<tr>
<td>429</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td>94111</td>
<td></td>
<td>10</td>
</tr>
</tbody>
</table>
```

(continues on next page)
We can also create a new table by adding two new columns with column name followed by the array values.

```python
[33]: cookies = Table()
    cookies = cookies.with_columns(
        "Cookie", make_array("Sugar cookies", "Chocolate chip", "Red velvet", "Oatmeal raisin
        ", "Peanut butter"),
        "Quantity", make_array(10, 15, 15, 10, 5)
    )
cookies
```

<table>
<thead>
<tr>
<th>Cookie</th>
<th>Quantity</th>
</tr>
</thead>
<tbody>
<tr>
<td>Sugar cookies</td>
<td>10</td>
</tr>
<tr>
<td>Chocolate chip</td>
<td>15</td>
</tr>
<tr>
<td>Red velvet</td>
<td>15</td>
</tr>
<tr>
<td>Oatmeal raisin</td>
<td>10</td>
</tr>
<tr>
<td>Peanut butter</td>
<td>5</td>
</tr>
</tbody>
</table>

```python
[34]: prices = make_array(1.00, 1.50, 1.75, 1.25, 1.00)
    cookies = cookies.with_column("Price ($)", prices)
cookies
```

<table>
<thead>
<tr>
<th>Cookie</th>
<th>Quantity</th>
<th>Price ($)</th>
</tr>
</thead>
<tbody>
<tr>
<td>Sugar cookies</td>
<td>10</td>
<td>1</td>
</tr>
<tr>
<td>Chocolate chip</td>
<td>15</td>
<td>1.5</td>
</tr>
<tr>
<td>Red velvet</td>
<td>15</td>
<td>1.75</td>
</tr>
<tr>
<td>Oatmeal raisin</td>
<td>10</td>
<td>1.25</td>
</tr>
<tr>
<td>Peanut butter</td>
<td>5</td>
<td>1</td>
</tr>
</tbody>
</table>
In the last examples, we add a new column Delicious with one value “yes,” and we see every column has the same value.

```
[35]: cookies.with_column("Delicious", "yes")

<table>
<thead>
<tr>
<th>Cookie</th>
<th>Quantity</th>
<th>Price ($)</th>
<th>Delicious</th>
</tr>
</thead>
<tbody>
<tr>
<td>Sugar cookies</td>
<td>10</td>
<td>1</td>
<td>yes</td>
</tr>
<tr>
<td>Chocolate chip</td>
<td>15</td>
<td>1.5</td>
<td>yes</td>
</tr>
<tr>
<td>Red velvet</td>
<td>15</td>
<td>1.75</td>
<td>yes</td>
</tr>
<tr>
<td>Oatmeal raisin</td>
<td>10</td>
<td>1.25</td>
<td>yes</td>
</tr>
<tr>
<td>Peanut butter</td>
<td>5</td>
<td>1</td>
<td>yes</td>
</tr>
</tbody>
</table>
```

### 2.1.4 `tbl.column()`

tbl.column(column_name_or_index)

Outputs an array of values of the column column_name_or_index. column_name_or_index is a string of the column name or number which is the index of the column.

In the examples below, we start with an array of the Cookie column from the table cookies first by the column name then by using the index of the column.

```
[36]: cookies.column("Cookie")
[36]: array(['Sugar cookies', 'Chocolate chip', 'Red velvet', 'Oatmeal raisin', 'Peanut butter'], dtype='<U14')

[37]: cookies.column(0)
[37]: array(['Sugar cookies', 'Chocolate chip', 'Red velvet', 'Oatmeal raisin', 'Peanut butter'], dtype='<U14')
```

### 2.1.5 `tbl.num_rows`

Computes the number of rows in a table.

```
[38]: trips.num_rows
[38]: 100000

[39]: cookies.num_rows
[39]: 5
```
2.1.6 tbl.num_columns

Computes the number of columns in a table.

```plaintext
[40]: trips.num_columns
[40]: 11

[41]: cookies.num_columns
[41]: 3
```

2.1.7 tbl.labels

Outputs the column labels in a table.

```plaintext
[42]: trips.labels
[42]: ('Trip ID', 'Duration', 'Start Date', 'Start Station', 'Start Terminal', 'End Date', 'End Station', 'End Terminal', 'Bike #', 'Subscriber Type', 'Zip Code')

[43]: cookies.labels
[43]: ('Cookie', 'Quantity', 'Price ($)')
```

2.1.8 tbl.select()

```plaintext
tbl.select(col1, col2, ...)
```

Creates a copy of a table with only the selected columns. Each column is the column name as a string or the integer index of the column.

Suppose we want to select the Trip ID, Duration, Bike #, and Zip Code columns from the trips table.

```plaintext
[44]: trips.select("Trip ID", "Duration", "Bike ", "Zip Code")
[44]: | Trip ID | Duration | Bike # | Zip Code |
    |---------|----------|--------|----------|
    | 876419  | 413      | 269    | 94518    |
    | 459672  | 408      | 429    | 94111    |
    | 903647  | 723      | 631    | 94025    |
    | 452829  | 409      | 428    | 94925    |
    | 491023  | 224      | 144    | 94117    |
    | 723352  | 519      | 629    | 94061    |
    | 524499  | 431      | 630    | 94706    |
```

(continues on next page)
518524 | 389 | 458 | 94610
710070 | 11460 | 375 | 94107
793149 | 616 | 289 | 94105
... (99990 rows omitted)

Similarly, we can use indexes to select columns. Remember to start indexing at 0.

```python
[45]: trips.select(0, 1, 8, 10).show(5)
<IPython.core.display.HTML object>
```

### 2.1.9 tbl.drop()

```python
tbl.drop(col1, col2, ...)
```

Creates a copy of a table without the specified columns. Each column is the column name as a string or integer index.

```python
[46]: cookies.drop("Quantity")
```

<table>
<thead>
<tr>
<th>Cookie</th>
<th>Price ($)</th>
</tr>
</thead>
<tbody>
<tr>
<td>Sugar cookies</td>
<td>1</td>
</tr>
<tr>
<td>Chocolate chip</td>
<td>1.5</td>
</tr>
<tr>
<td>Red velvet</td>
<td>1.75</td>
</tr>
<tr>
<td>Oatmeal raisin</td>
<td>1.25</td>
</tr>
<tr>
<td>Peanut butter</td>
<td>1</td>
</tr>
</tbody>
</table>

```python
[47]: trips.drop("End Date", "Subscriber Type")
```

<table>
<thead>
<tr>
<th>Trip ID</th>
<th>Duration</th>
<th>Start Date</th>
<th>Start Station</th>
<th>Start Terminal</th>
<th>End Station</th>
<th>End Terminal</th>
<th>Bike #</th>
<th>Zip Code</th>
</tr>
</thead>
<tbody>
<tr>
<td>876419</td>
<td>413</td>
<td>8/5/2015 8:29</td>
<td>Civic Center BART (7th at Market)</td>
<td>72</td>
<td>Townsend at 7th</td>
<td>65</td>
<td>269</td>
<td>94518</td>
</tr>
<tr>
<td>459672</td>
<td>408</td>
<td>9/18/2014 17:11</td>
<td>Harry Bridges Plaza (Ferry Building)</td>
<td>50</td>
<td>Embarcadero at Sansome</td>
<td>60</td>
<td>429</td>
<td>94111</td>
</tr>
<tr>
<td>903647</td>
<td>723</td>
<td>8/25/2015 7:26</td>
<td>San Francisco Caltrain 2 (330 Townsend)</td>
<td>69</td>
<td>Market at 10th</td>
<td>67</td>
<td>631</td>
<td>94025</td>
</tr>
<tr>
<td>452829</td>
<td>409</td>
<td>9/15/2014 8:29</td>
<td>Steuart at Market</td>
<td>74</td>
<td>Market at 4th</td>
<td>76</td>
<td>428</td>
<td>94925</td>
</tr>
<tr>
<td>491023</td>
<td>224</td>
<td>10/9/2014 16:13</td>
<td>Santa Clara at Almaden</td>
<td>4</td>
<td>San Jose Diridon Caltrain Station</td>
<td>2</td>
<td>144</td>
<td>94117</td>
</tr>
<tr>
<td>723352</td>
<td>519</td>
<td>4/13/2015 17:04</td>
<td>Howard at 2nd</td>
<td>63</td>
<td>San Francisco Caltrain (Townsend at 4th)</td>
<td>70</td>
<td>629</td>
<td>94061</td>
</tr>
<tr>
<td>524499</td>
<td>431</td>
<td>10/31/2014 16:36</td>
<td>Townsend at 7th</td>
<td>65</td>
<td>Civic Center BART (7th at Market)</td>
<td>72</td>
<td>630</td>
<td>94706</td>
</tr>
<tr>
<td>518524</td>
<td>389</td>
<td>10/28/2014 8:48</td>
<td>Market at Sansome</td>
<td>77</td>
<td>2nd at South Park</td>
<td>64</td>
<td>458</td>
<td>94610</td>
</tr>
<tr>
<td>710070</td>
<td>11460</td>
<td>4/2/2015 18:13</td>
<td>Powell Street BART</td>
<td>39</td>
<td>Powell Street BART</td>
<td>39</td>
<td>375</td>
<td>94107</td>
</tr>
<tr>
<td>793149</td>
<td>616</td>
<td>6/4/2015 5:26</td>
<td>Embarcadero at Bryant</td>
<td>54</td>
<td>Embarcadero at Sansome</td>
<td>60</td>
<td>289</td>
<td>94105</td>
</tr>
</tbody>
</table>
... (99990 rows omitted)
2.1.10 tbl.relabel()

`tbl.relabel(old_label, new_label)`

Modifies the table by changing the label of the column named `old_label` to `new_label`. `old_label` can be a string column name or an integer index.

```r
# Example with a data frame
trips.drop(3, 6, 8, 9, 10)
```

<table>
<thead>
<tr>
<th>Trip ID</th>
<th>Duration</th>
<th>Start Date</th>
<th>Start Terminal</th>
<th>End Date</th>
<th>End Terminal</th>
</tr>
</thead>
<tbody>
<tr>
<td>876419</td>
<td>413</td>
<td>8/5/2015 8:29</td>
<td>72</td>
<td>8/5/2015 8:36</td>
<td>65</td>
</tr>
<tr>
<td>459672</td>
<td>408</td>
<td>9/18/2014 17:11</td>
<td>50</td>
<td>9/18/2014 17:17</td>
<td>60</td>
</tr>
<tr>
<td>903647</td>
<td>723</td>
<td>8/25/2015 7:26</td>
<td>69</td>
<td>8/25/2015 7:38</td>
<td>67</td>
</tr>
<tr>
<td>452829</td>
<td>409</td>
<td>9/15/2014 8:29</td>
<td>74</td>
<td>9/15/2014 8:36</td>
<td>76</td>
</tr>
<tr>
<td>723352</td>
<td>519</td>
<td>4/13/2015 17:04</td>
<td>63</td>
<td>4/13/2015 17:12</td>
<td>70</td>
</tr>
<tr>
<td>524499</td>
<td>431</td>
<td>10/31/2014 16:36</td>
<td>65</td>
<td>10/31/2014 16:43</td>
<td>72</td>
</tr>
<tr>
<td>518524</td>
<td>389</td>
<td>10/28/2014 8:48</td>
<td>77</td>
<td>10/28/2014 8:54</td>
<td>64</td>
</tr>
</tbody>
</table>

... (99990 rows omitted)
2.1.11 `tbl.show()`

`tbl.show(n)`

Displays the first $n$ rows of a table. If no $n$ is provided, displays all rows.

```
[53]: trips.show(5)
<IPython.core.display.HTML object>
```

2.1.12 `tbl.sort()`

`tbl.sort(column_name_or_index, descending=False)`

Sorts the rows in the table by the values in the column `column_name_or_index` in ascending order by default. Set `descending=True` to sort in descending order. `column_name_or_index` can be a string column label or an integer index.

```
[54]: cookies

      Type          | Amount remaining | Price ($)
Sugar cookies   | 10             | 1
Chocolate chip | 15             | 1.5
Red velvet     | 15             | 1.75
Oatmeal raisin| 10             | 1.25
Peanut butter  | 5              | 1

[55]: cookies.sort("Price ($)")

      Type          | Amount remaining | Price ($)
Sugar cookies   | 10             | 1
Peanut butter  | 5              | 1
Oatmeal raisin| 10             | 1.25
Chocolate chip | 15             | 1.5
Red velvet     | 15             | 1.75

[56]: # sort in descending order
cookies.sort("Amount remaining", descending = True)

      Type          | Amount remaining | Price ($)
Red velvet     | 15             | 1.75
Chocolate chip | 15             | 1.5
Oatmeal raisin| 10             | 1.25
Sugar cookies  | 10             | 1
Peanut butter  | 5              | 1
`
# alphabetical order
cookies.sort(0)

<table>
<thead>
<tr>
<th>Type</th>
<th>Amount remaining</th>
<th>Price ($)</th>
</tr>
</thead>
<tbody>
<tr>
<td>Chocolate chip</td>
<td>15</td>
<td>1.5</td>
</tr>
<tr>
<td>Oatmeal raisin</td>
<td>10</td>
<td>1.25</td>
</tr>
<tr>
<td>Peanut butter</td>
<td>5</td>
<td>1</td>
</tr>
<tr>
<td>Red velvet</td>
<td>15</td>
<td>1.75</td>
</tr>
<tr>
<td>Sugar cookies</td>
<td>10</td>
<td>1</td>
</tr>
</tbody>
</table>

## 2.1.13 tbl.where()

tbl.where(column, predicate)

Filters the table for rows where the predicate is true. predicate should be one of the provided are.<something> functions. column can be a string column label or an integer index. A list of available predicates can be found below.

cookies.where("Amount remaining", are.above(10))

<table>
<thead>
<tr>
<th>Type</th>
<th>Amount remaining</th>
<th>Price ($)</th>
</tr>
</thead>
<tbody>
<tr>
<td>Chocolate chip</td>
<td>15</td>
<td>1.5</td>
</tr>
<tr>
<td>Red velvet</td>
<td>15</td>
<td>1.75</td>
</tr>
</tbody>
</table>

cookies.where(0, are.equal_to("Chocolate chip"))

<table>
<thead>
<tr>
<th>Type</th>
<th>Amount remaining</th>
<th>Price ($)</th>
</tr>
</thead>
<tbody>
<tr>
<td>Chocolate chip</td>
<td>15</td>
<td>1.5</td>
</tr>
</tbody>
</table>

cookies.where("Price ($)", are.below(1.25))

<table>
<thead>
<tr>
<th>Type</th>
<th>Amount remaining</th>
<th>Price ($)</th>
</tr>
</thead>
<tbody>
<tr>
<td>Sugar cookies</td>
<td>10</td>
<td>1</td>
</tr>
<tr>
<td>Peanut butter</td>
<td>5</td>
<td>1</td>
</tr>
</tbody>
</table>

## 2.1.14 tbl.take()

tbl.take(row_index, ...)

Returns a copy of the table with only the specified rows included. Rows are specified by their integer index, so 0 for the first, 1 for the second, etc.

cookies
### 2.2 Table Visualizations

```python
[68]:
actors = Table().read_table("https://github.com/data-8/textbook/raw/gh-pages/data/actors.csv")
actors
```

<table>
<thead>
<tr>
<th>Actor</th>
<th>Total Gross</th>
<th>Number of Movies</th>
<th>Average per Movie</th>
<th>#1 Movie</th>
</tr>
</thead>
<tbody>
<tr>
<td>Harrison Ford</td>
<td>4871.7</td>
<td>41</td>
<td>118.8</td>
<td>Star Wars: The Force Awakens</td>
</tr>
<tr>
<td>Samuel L. Jackson</td>
<td>4772.8</td>
<td>69</td>
<td>69.2</td>
<td>The Avengers</td>
</tr>
<tr>
<td>Morgan Freeman</td>
<td>4468.3</td>
<td>61</td>
<td>73.3</td>
<td>The Dark Knight</td>
</tr>
<tr>
<td>Tom Hanks</td>
<td>4340.8</td>
<td>44</td>
<td>98.7</td>
<td>Toy Story 3</td>
</tr>
<tr>
<td>Robert Downey, Jr.</td>
<td>3947.3</td>
<td>53</td>
<td>74.5</td>
<td>The Avengers</td>
</tr>
<tr>
<td>Eddie Murphy</td>
<td>3810.4</td>
<td>38</td>
<td>100.3</td>
<td>Shrek 2</td>
</tr>
<tr>
<td>Tom Cruise</td>
<td>3587.2</td>
<td>36</td>
<td>99.6</td>
<td>War of the Worlds</td>
</tr>
<tr>
<td>Johnny Depp</td>
<td>3368.6</td>
<td>45</td>
<td>74.9</td>
<td>Dead Man's Chest</td>
</tr>
<tr>
<td>Michael Caine</td>
<td>3351.5</td>
<td>58</td>
<td>57.8</td>
<td>The Dark Knight</td>
</tr>
<tr>
<td>Scarlett Johansson</td>
<td>3341.2</td>
<td>37</td>
<td>90.3</td>
<td>The Avengers</td>
</tr>
</tbody>
</table>
```

(continues on next page)
2.2.1 tbl.scatter()

tbl.scatter(x_column, y_column, fit_line=False)

Creates a scatter plot with `x_column` on the horizontal axis and `y_column` on the vertical axis. These labels can be column names as strings or integer indices. Set `fit_line=True` to include a line of best fit for the data. You can find more examples in the textbook.

[71]: actors.scatter('Number of Movies', 'Total Gross')

![Scatter Plot]

[73]: actors.scatter(2, 3, fit_line=True)
2.2.2 tbl.plot()

tbl.plot(x_column, y_column)

Plot a line graph with x_column on the horizontal axis and y_column on the vertical axis. Sorts the table in ascending order by values in x_column first. x_column and y_column can be column names as strings or integer indices.

```
[74]: movies_by_year = Table.read_table('https://github.com/data-8/textbook/raw/gh-pages/data/movies_by_year.csv')
movies_by_year.show(3)
<IPython.core.display.HTML object>
```

```
[75]: movies_by_year.plot('Year', 'Number of Movies')
```
2.2.3 tbl.barh()

tbl.barh(categories)
tbl.barh(categories, values)

Plots a horizontal bar chart broken down by categories as the bars. If values is unspecified, one bar for each column of the table (except categories) is plotted. categories and values can be column names as strings or integer indices.

[76]: cookies.barh("Type")

![Horizontal Bar Chart]

[77]: cookies.barh("Type", "Amount remaining")
Plot a histogram of the values in column. Defaults to 10 bins of equal width. If bins is specified, it can be a number of bins to use (e.g. bins=25 will produce a histogram with 25 bins) or an array of values to use as bins (e.g. bins=make_array(1, 3, 4) will produce 2 bins: [1,3) and [3,4)). column can be column names as strings or integer indices.
2.2.5 `Table.interactive_plots()`

`Table.interactive_plots()`

This function will change from static plots like the ones above to interactive plots made with `plotly`. If a plotting method has a `plotly` version, that method will be used instead.

[193]: `Table.interactive_plots()`
`actors.scatter("Total Gross", "Gross")`
2.2.6 Table.static_plots()

Table.static_plots()

This function turns off plotly plots.

[194]: Table.static_plots()
actors.scatter("Total Gross", "Gross")

2.3 Advanced Table Functions

2.3.1 tbl.apply()

tbl.apply(function, column)
tbl.apply(function, col1, col2, ...)

Applies the function function to each element of the column column and returns the values returned as an array. If function takes more than one argument, you can specify multiple columns to use for each argument in order.

[65]: actors.apply(np.average, "Number of Movies")
        34])

(continues on next page)
The example below calculates the average gross for each movie by actor by applying a function that takes in the value of Total Gross and Number of Movies and returns their quotient.

```python
def average_gross(total_gross, num_movies):
    return total_gross / num_movies
```

actors.apply(average_gross, "Total Gross", "Number of Movies")
### 2.3.2 tbl.group()

<table>
<thead>
<tr>
<th>Function Call</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>tbl.group(column_or_columns)</td>
<td>Groups a table by values in column_or_columns.</td>
</tr>
<tr>
<td>tbl.group(column_or_columns, func)</td>
<td>If column_or_columns is an array, groups by each unique combination of elements in those columns. If func is specified, it should be a function that takes in an array of values and returns a single value. If unspecified, this defaults to the count of rows in the set.</td>
</tr>
</tbody>
</table>

#### Examples

**[84]**: trips.group("Start Station")

<table>
<thead>
<tr>
<th>Start Station</th>
<th>count</th>
</tr>
</thead>
<tbody>
<tr>
<td>2nd at Folsom</td>
<td>2302</td>
</tr>
<tr>
<td>2nd at South Park</td>
<td>2610</td>
</tr>
<tr>
<td>2nd at Townsend</td>
<td>3904</td>
</tr>
<tr>
<td>5th at Howard</td>
<td>2190</td>
</tr>
<tr>
<td>Adobe on Almaden</td>
<td>165</td>
</tr>
<tr>
<td>Arena Green / SAP Center</td>
<td>176</td>
</tr>
<tr>
<td>Beale at Market</td>
<td>2377</td>
</tr>
<tr>
<td>Broadway St at Battery St</td>
<td>2157</td>
</tr>
<tr>
<td>California Ave Caltrain Station</td>
<td>127</td>
</tr>
<tr>
<td>Castro Street and El Camino Real</td>
<td>339</td>
</tr>
<tr>
<td>... (60 rows omitted)</td>
<td></td>
</tr>
</tbody>
</table>

**[85]**: trips.group("Start Station", np.mean).select(0,2)

<table>
<thead>
<tr>
<th>Start Station</th>
<th>Duration mean</th>
</tr>
</thead>
<tbody>
<tr>
<td>2nd at Folsom</td>
<td>512.887</td>
</tr>
<tr>
<td>2nd at South Park</td>
<td>654.565</td>
</tr>
<tr>
<td>2nd at Townsend</td>
<td>755.176</td>
</tr>
<tr>
<td>5th at Howard</td>
<td>819.509</td>
</tr>
<tr>
<td>Adobe on Almaden</td>
<td>2522.5</td>
</tr>
<tr>
<td>Arena Green / SAP Center</td>
<td>1999.7</td>
</tr>
<tr>
<td>Beale at Market</td>
<td>679.602</td>
</tr>
<tr>
<td>Broadway St at Battery St</td>
<td>827.753</td>
</tr>
<tr>
<td>California Ave Caltrain Station</td>
<td>4403.29</td>
</tr>
<tr>
<td>Castro Street and El Camino Real</td>
<td>1221.86</td>
</tr>
<tr>
<td>... (60 rows omitted)</td>
<td></td>
</tr>
</tbody>
</table>

**[86]**: trips.group("Start Station").sort("count", descending = True)

<table>
<thead>
<tr>
<th>Start Station</th>
<th>count</th>
</tr>
</thead>
<tbody>
<tr>
<td>San Francisco Caltrain (Townsend at 4th)</td>
<td>7426</td>
</tr>
<tr>
<td>San Francisco Caltrain 2 (330 Townsend)</td>
<td>6114</td>
</tr>
<tr>
<td>Harry Bridges Plaza (Ferry Building)</td>
<td>4795</td>
</tr>
<tr>
<td>Temporary Transbay Terminal (Howard at Beale)</td>
<td>4212</td>
</tr>
<tr>
<td>Townsend at 7th</td>
<td>3925</td>
</tr>
<tr>
<td>2nd at Townsend</td>
<td>3904</td>
</tr>
<tr>
<td>Embarcadero at Sansome</td>
<td>3900</td>
</tr>
<tr>
<td>Steuart at Market</td>
<td>3872</td>
</tr>
<tr>
<td>Market at 10th</td>
<td>3370</td>
</tr>
<tr>
<td>Market at Sansome</td>
<td>3218</td>
</tr>
<tr>
<td>... (60 rows omitted)</td>
<td></td>
</tr>
</tbody>
</table>
2.3.3 tbl.pivot()

```
tbl.pivot(col1, col2)
tbl.pivot(col1, col2, values, collect)
```

Creates a pivot table with values in col1 as columns and values in col2 as rows. If values is unspecified, the values in the cells default to counts. If values is specified, it should be the label of a column whose values to pass as an array to collect, which should return a single value.

```
[88]: more_cones = Table().with_columns(
    'Flavor', make_array('strawberry', 'chocolate', 'chocolate', 'strawberry', 'chocolate', 'bubblegum'),
    'Color', make_array('pink', 'light brown', 'dark brown', 'pink', 'dark brown', 'pink'),
    'Price', make_array(3.55, 4.75, 5.25, 5.25, 5.25, 4.75)
)

more_cones

```

```
[88]: Flavor | Color    | Price
strawberry | pink     | 3.55
chocolate   | light brown | 4.75
chocolate   | dark brown | 5.25
strawberry  | pink      | 5.25
chocolate   | dark brown | 5.25
bubblegum   | pink      | 4.75
```

```
[89]: more_cones.pivot('Flavor', 'Color')

```

```
[89]: Color     | bubblegum | chocolate | strawberry
dark brown   | 0         | 2         | 0
light brown  | 0         | 1         | 0
pink         | 1         | 0         | 2
```

```
[90]: more_cones.pivot('Flavor', 'Color', values='Price', collect=sum)

```

```
[90]: Color        | bubblegum | chocolate | strawberry
dark brown     | 0         | 10.5      | 0
```

(continues on next page)
### 2.3.4 tbl.join()

```python
tbl1.join(col1, tbl2)
tbl1.join(col1, tbl2, col2)
```

Performs a join of tbl1 on tbl2 where rows are only included if the value in col1 is present in both join columns. If col2 is unspecified, it is assumed to be the same label as col1.

```python
[92]: cones = Table().with_columns(
    'Flavor', make_array('strawberry', 'vanilla', 'chocolate', 'strawberry', 'chocolate'),
    'Price', make_array(3.55, 4.75, 6.55, 5.25, 5.75)
)
cones
```

```python
[92]: Flavor   | Price
strawberry | 3.55
vanilla    | 4.75
chocolate  | 6.55
strawberry | 5.25
chocolate  | 5.75
```

```python
[95]: ratings = Table().with_columns(
    'Kind', make_array('strawberry', 'chocolate', 'vanilla', 'mint chip'),
    'Stars', make_array(2.5, 3.5, 4, 3)
)
ratings
```

```python
[95]: Kind   | Stars
strawberry | 2.5
chocolate  | 3.5
vanilla    | 4
mint chip  | 3
```

```python
[97]: # Joins cones on ratings. Note that the mint chip flavor doesn't appear since it's not in cones
rated = cones.join('Flavor', ratings, 'Kind')
```

```python
[97]: Flavor | Price | Stars
chocolate | 6.55 | 3.5
```

(continues on next page)
### 2.3.5 tbl.sample()

**tbl.sample(n, with_replacement=True)**

Returns a new table with `n` rows that were randomly sampled from the original table. If `with_replacement` is true, sampling occurs with replacement. For sampling without replacement, set `with_replacement=False`.

```
[98]: # if you rerun this cell, you should get different results since the sample is random
    : rated.sample(2)
```

<table>
<thead>
<tr>
<th>Flavor</th>
<th>Price</th>
<th>Stars</th>
</tr>
</thead>
<tbody>
<tr>
<td>chocolate</td>
<td>6.55</td>
<td>3.5</td>
</tr>
<tr>
<td>chocolate</td>
<td>6.55</td>
<td>3.5</td>
</tr>
</tbody>
</table>

Notice how the table below has more rows for certain flavors than the original rated table. This is because we are sampling with replacement, so you get theoretically get 5 of the same flavors!

```
[99]: sampled_with_replacement = rated.sample(5)
    : sampled_with_replacement
```

<table>
<thead>
<tr>
<th>Flavor</th>
<th>Price</th>
<th>Stars</th>
</tr>
</thead>
<tbody>
<tr>
<td>strawberry</td>
<td>5.25</td>
<td>2.5</td>
</tr>
<tr>
<td>strawberry</td>
<td>3.55</td>
<td>2.5</td>
</tr>
<tr>
<td>strawberry</td>
<td>3.55</td>
<td>2.5</td>
</tr>
<tr>
<td>chocolate</td>
<td>6.55</td>
<td>3.5</td>
</tr>
<tr>
<td>vanilla</td>
<td>4.75</td>
<td>4</td>
</tr>
</tbody>
</table>

```
[100]: rated.sample(3, with_replacement = False)
    : rated.sample(3, with_replacement = False)
```

<table>
<thead>
<tr>
<th>Flavor</th>
<th>Price</th>
<th>Stars</th>
</tr>
</thead>
<tbody>
<tr>
<td>vanilla</td>
<td>4.75</td>
<td>4</td>
</tr>
<tr>
<td>strawberry</td>
<td>3.55</td>
<td>2.5</td>
</tr>
<tr>
<td>chocolate</td>
<td>6.55</td>
<td>3.5</td>
</tr>
</tbody>
</table>

### 2.4 String Methods

#### 2.4.1 str.split()  

**string.split(separator)**

Splits the string `string` into a list on each occurrence of the substring `separator`. The occurrences of `separator` are removed from the resulting list.

For example, the code below splits the string `Data 8hiishifun` on the substring `hi`.  

```
```
example_string = "Data &hiishifun."
example_string.split("hi")

['Data 8', 'is', 'fun.]

# split on .
another_string = "the.secret.message.is.123"
another_string.split(".")

['the', 'secret', 'message', 'is', '123']

2.4.2 str.join()

string.join(array)

Combines each element of array into one string with string used to connect each element.

fun_array = make_array("high", "great", "best")
"est ".join(fun_array)

'highest greatest best'

# you can join elements on the empty string to just merge the elements
some_strings = make_array("some", "list", "of", "strings")
"".join(some_strings)

'somelistofstrings'

2.4.3 str.replace()

string.replace(old_string, new_string)

Replaces each occurrence of old_string in string with new_string.

berkeley_string = "I saw 5 friends, 10 squirrels, and 20 people flyering on Sproul."
berkeley_string

'I saw 5 friends, 10 squirrels, and 20 people flyering on Sproul.'

berkeley_string.replace("friends", "frisbees")

'I saw 5 frisbees, 10 squirrels, and 20 people flyering on Sproul.'

berkeley_string.replace("friends", "frisbees").replace("flyering on Sproul", "having a picnic on the Glade")

'I saw 5 frisbees, 10 squirrels, and 20 people having a picnic on the Glade.'
### 2.5 Array Functions and Methods

```plaintext
[111]: example_array = make_array(1, 3, 5, 7, 9)
    example_array
[111]: array([1, 3, 5, 7, 9])
```

#### 2.5.1 max()

```plaintext
max(array)
```

Returns the maximum value of an array.

```plaintext
[112]: max(example_array)
[112]: 9
```

#### 2.5.2 min()

```plaintext
min(array)
```

Returns the minimum value of an array.

```plaintext
[113]: min(example_array)
[113]: 1
```

#### 2.5.3 sum()

```plaintext
sum(array)
```

Returns the sum of values in an array.

```plaintext
[114]: sum(example_array)
[114]: 25

[115]: sum(make_array(1, 2, 0, -10))
[115]: -7
```
2.5.4 abs()

```
abs(num)
abs(array)
```

Take the absolute value of number or each number in an array.

```
[118]: abs(-1)
1
[119]: new_arr = make_array(-3, -1, 5.2, 0.25, -4.9)
abs(new_arr)
array([3. , 1. , 5.2 , 0.25, 4.9 ])
```

2.5.5 round(num)

```
round(num)
round(num, d)
np.round(array)
np.round(array, d)
```

Round number or array of numbers to the nearest integer. If d is specified, rounds to d places after the decimal. Use np.round to round arrays.

```
[124]: round(3.14159)
3
[125]: round(3.14159, 3)
3.142
[130]: np.round(new_arr, 1)
array([-3. , -1. , 5.2 , 0.25, 4.9 ])
```

2.5.6 len()

```
len(array)
```

Returns the length of an array.

```
[131]: len(new_arr)
5
```
2.5.7 make_array()

make_array(val1, val2, ...)

Creates a new array with the values passed.

```python
[132]: new_array = make_array(25, 16, 9, 4, 1)
[132]: array([25, 16, 9, 4, 1])
```

2.5.8 np.mean

```python
np.mean(array)
np.average(array)
```

Returns the mean of the values in an array.

```python
[134]: np.mean(new_array)
[134]: 11.0

[133]: np.average(new_array)
[133]: 11.0
```

2.5.9 np.std()

```python
np.std(array)
```

Returns the standard deviation of the values in an array.

```python
[150]: np.std(new_array)
[150]: 8.648699324175862
```

2.5.10 np.diff()

```python
np.diff(array)
```

Returns an array with the pairwise differences between elements in the input array. The output will have length len(array) - 1 and will have elements $x_1 - x_0, x_2 - x_1, x_3 - x_2,$ etc.

```python
[135]: np.diff(new_array)
[135]: array([-9, -7, -5, -3])

[136]: np.diff(make_array(1, 3, 5, 7))
[136]: array([2, 2, 2])
```
2.5.11 np.sqrt()

np.sqrt(num)
np.sqrt(array)

Returns the square root of a number or an array of the square roots of each element in the input array.

```
[137]: np.sqrt(4)
2.0
[138]: np.sqrt(new_array)
array([5., 4., 3., 2., 1.])
```

2.5.12 np.arange()

np.arange(stop)
np.arange(start, stop)
np.arange(start, stop, step)

Returns an array of integers from start to stop incrementing by step. If start is unspecified, it is assumed be 0. If step is unspecified, it is assumed to be 1. The upper bound is exclusive, meaning that max(np.arange(10)) is 9.

```
[139]: np.arange(0, 11)
array([ 0, 1, 2, 3, 4, 5, 6, 7, 8, 9, 10])
[140]: np.arange(5)
array([0, 1, 2, 3, 4])
[142]: np.arange(0, 102, 2.7)
array([ 0. , 2.7, 5.4, 8.1, 10.8, 13.5, 16.2, 18.9, 21.6, 24.3, 27. ,
       29.7, 32.4, 35.1, 37.8, 40.5, 43.2, 45.9, 48.6, 51.3, 54. , 56.7,
       59.4, 62.1, 64.8, 67.5, 70.2, 72.9, 75.6, 78.3, 81. , 83.7, 86.4,
       89.1, 91.8, 94.5, 97.2, 99.9])
```

2.5.13 array.item()

array.item(num)

Returns the item at index num in an array (remember Python indices start at 0!).

```
[143]: np.arange(0, 102, 2).item(1)
2
[146]: new_array.item(2)
9
```
2.5.14 np.random.choice

np.random.choice(array)
np.random.choice(array, n, replace=True)

Picks one or n of items from an array at random. By default, with replacement (set replace=False for without replacement).

```
[149]: np.random.choice(new_array)
[149]: 25
[150]: np.random.choice(new_array, 3)
[150]: array([ 4,  4, 16])
[152]: np.random.choice(np.arange(0, 102, 2), 10, replace=False)
[152]: array([ 98,  22,  12,  24,  54, 100,  52,  28,  88])
```

2.5.15 np.count_nonzero()

Returns the number of nonzero elements in an array. Because False values are considered zeros (as integers), this can also give you the number of True s in an array of boolean values.

```
[153]: another_array = make_array(0, 1, 2, 0, 4, 0, 1, 0, 0)
[153]: 4
[159]: bools = make_array(True, False, True, False, False, True, False)
[159]: 3
```

2.5.16 np.append()

np.append(array, item)

Returns a copy of the input array with item (must be the same type as the other entries in the array) appended to the end.

```
[160]: new_array
[160]: array([25, 16,  9,  4,  1])
```
2.5.17 percentile()

percentile(percent, array)

Returns the value corresponding to the specified percentile of an array. `percent` should be in percentage form (i.e. 50 not 0.5).

```python
[162]: long_array = make_array(1, 1, 1, 2, 2, 2, 3, 3, 3, 4)
long_array
[162]: array([1, 1, 1, 2, 2, 2, 3, 3, 3, 4])
[163]: percentile(50, long_array)
[163]: 2
[164]: percentile(90, long_array)
[164]: 3
```

2.6 Table.where Predicates

All of the predicates described below can be negated by preceding the name with `not_`. For example, we can find values not equal to a specific value using `are.not_equal_to(value)`.

2.6.1 are.equal_to()

tbl.where(column, are.equal_to(value))

Filter leaves rows only where the value in `column` is equal to `value`.

```python
[166]: trips.where("Duration", are.equal_to(519))
```

(continues on next page)
```sql
2.6.2 are.above()

tbl.where(column, are.above(value))

Filter leaves rows only where the value in column is strictly greater than value.

```
```
### 2.6.3 `are.above_or_equal_to()`

Filter leaves rows only where the value in column is greater than or equal to value.

```
[168]: trips.where("Duration", are.above_or_equal_to(1000))
```

<table>
<thead>
<tr>
<th>Trip ID</th>
<th>Duration</th>
<th>Start Date</th>
<th>Start Station</th>
<th>Start Terminal</th>
<th>End Date</th>
<th>End Station</th>
<th>End Terminal</th>
<th>Bike #</th>
</tr>
</thead>
<tbody>
<tr>
<td>710070</td>
<td>11460</td>
<td>4/2/2015</td>
<td>Powell Street BART</td>
<td>39</td>
<td>4/2/</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>831059</td>
<td>1057</td>
<td>7/2/2015</td>
<td>South Van Ness at Market</td>
<td>66</td>
<td>7/2/</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>442750</td>
<td>6084</td>
<td>9/8/2014</td>
<td>Embarcadero at Sansome</td>
<td>60</td>
<td>9/8/</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>608714</td>
<td>19799</td>
<td>1/18/2015</td>
<td>San Francisco Caltrain (Townsend at 4th)</td>
<td>70</td>
<td>1/18/</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>711961</td>
<td>1026</td>
<td>4/4/2015</td>
<td>Davis at Jackson</td>
<td>42</td>
<td>4/4/</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>833071</td>
<td>2314</td>
<td>7/4/2015</td>
<td>Washington at Kearny</td>
<td>46</td>
<td>7/4/</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>570731</td>
<td>1218</td>
<td>12/8/2014</td>
<td>MLK Library</td>
<td>11</td>
<td>12/9/</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>853698</td>
<td>1048</td>
<td>7/20/2015</td>
<td>Broadway St at Battery St</td>
<td>82</td>
<td>7/20/</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>...</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
</tbody>
</table>
2.6.4 are.below()

tbl.where(column, are.below(value))

Filter leaves rows only where the value in column is strictly less than value.

[170]: trips.where("Duration", are.below(100))

<table>
<thead>
<tr>
<th>Trip ID</th>
<th>Duration</th>
<th>Start Date</th>
<th>Start Station</th>
<th>Start Terminal</th>
<th>End Date</th>
<th>End Station</th>
<th>End Terminal</th>
<th>Bike #</th>
<th>Subscriber Type</th>
<th>Zip Code</th>
</tr>
</thead>
<tbody>
<tr>
<td>482797</td>
<td>65</td>
<td>10/4/2014 7:50</td>
<td>San Francisco Caltrain (Townsend at 4th)</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td>430</td>
<td>Customer</td>
<td>95112</td>
</tr>
<tr>
<td>483052</td>
<td>81</td>
<td>10/4/2014 13:52</td>
<td>Harry Bridges Plaza (Ferry Building)</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td>306</td>
<td>Customer</td>
<td>nan</td>
</tr>
<tr>
<td>569620</td>
<td>84</td>
<td>12/8/2014 10:09</td>
<td>Civic Center BART (7th at Market)</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td>326</td>
<td>Subscriber</td>
<td>94111</td>
</tr>
<tr>
<td>502332</td>
<td>79</td>
<td>10/16/2014 17:26</td>
<td>Beale at Market</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td>50</td>
<td>Subscriber</td>
<td>94107</td>
</tr>
<tr>
<td>604012</td>
<td>76</td>
<td>1/14/2015 15:18</td>
<td>Davis at Jackson</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td>613</td>
<td>Subscriber</td>
<td>94602</td>
</tr>
<tr>
<td>704918</td>
<td>70</td>
<td>3/30/2015 22:51</td>
<td>Broadway St at Battery St</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td>394</td>
<td>Subscriber</td>
<td>94107</td>
</tr>
<tr>
<td>513458</td>
<td>83</td>
<td>10/24/2014 8:50</td>
<td>2nd at Folsom</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td>62</td>
<td>Subscriber</td>
<td>94107</td>
</tr>
<tr>
<td>696725</td>
<td>94</td>
<td>3/25/2015 8:47</td>
<td>Post at Kearny</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td>569</td>
<td>Subscriber</td>
<td>94107</td>
</tr>
<tr>
<td>829817</td>
<td>86</td>
<td>7/1/2015 9:27</td>
<td>Market at Sansome</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td>77</td>
<td>Subscriber</td>
<td>94538</td>
</tr>
<tr>
<td>745895</td>
<td>73</td>
<td>4/29/2015 13:05</td>
<td>Yerba Buena Center of the Arts (3rd @ Howard)</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td>380</td>
<td>Subscriber</td>
<td>94947</td>
</tr>
</tbody>
</table>

... (403 rows omitted)
### 2.6.5 are.below_or_equal_to()

```ruby
tbl.where(column, are.below_or_equal_to(value))
```

Filter leaves rows only where the value in `column` is less than or equal to `value`.

```
trip = trips.where("Duration", are.below_or_equal_to(100))
```

```
<table>
<thead>
<tr>
<th>Trip ID</th>
<th>Duration</th>
<th>Start Date</th>
<th>Start Station</th>
<th>Start Terminal</th>
<th>End Date</th>
<th>End Station</th>
<th>End Terminal</th>
<th>Bike #</th>
<th>Subscriber Type</th>
<th>Zip Code</th>
</tr>
</thead>
<tbody>
<tr>
<td>482797</td>
<td>65</td>
<td>10/4/2014</td>
<td>San Francisco Caltrain (Townsend at 4th)</td>
<td>70</td>
<td>10/4/2014</td>
<td>San Francisco Caltrain (Townsend at 4th)</td>
<td>70</td>
<td>430</td>
<td>Subscriber</td>
<td>95112</td>
</tr>
<tr>
<td>569620</td>
<td>84</td>
<td>12/8/2014</td>
<td>Civic Center BART (7th at Market)</td>
<td>72</td>
<td>12/8/2014</td>
<td>10:10</td>
<td>Civic Center BART (7th at Market)</td>
<td>72</td>
<td>326</td>
<td>Subscriber</td>
</tr>
<tr>
<td>502332</td>
<td>79</td>
<td>10/16/2014</td>
<td>Beale at Market</td>
<td>56</td>
<td>10/16/2014</td>
<td>17:27</td>
<td>Temporary Transbay Terminal (Howard at Beale)</td>
<td>55</td>
<td>613</td>
<td>Subscriber</td>
</tr>
<tr>
<td>604012</td>
<td>76</td>
<td>1/14/2015</td>
<td>Davis at Jackson</td>
<td>42</td>
<td>1/14/2015</td>
<td>15:19</td>
<td>Broadway St at Battery St</td>
<td>82</td>
<td>601</td>
<td>Subscriber</td>
</tr>
<tr>
<td>704918</td>
<td>70</td>
<td>3/30/2015</td>
<td>Broadway St at Battery St</td>
<td>82</td>
<td>3/30/2015</td>
<td>22:51</td>
<td>Broadway St at Battery St</td>
<td>82</td>
<td>394</td>
<td>Subscriber</td>
</tr>
<tr>
<td>513458</td>
<td>83</td>
<td>10/24/2014</td>
<td>2nd at Folsom</td>
<td>62</td>
<td>10/24/2014</td>
<td>8:51</td>
<td>Howard at 2nd</td>
<td>63</td>
<td>569</td>
<td>Subscriber</td>
</tr>
<tr>
<td>808199</td>
<td>100</td>
<td>6/15/2015</td>
<td>20:57</td>
<td>Post at Kearny</td>
<td>47</td>
<td>6/15/2015</td>
<td>20:58</td>
<td>2nd at South Park</td>
<td>64</td>
<td>537</td>
</tr>
<tr>
<td>829817</td>
<td>86</td>
<td>7/1/2015</td>
<td>9:28</td>
<td>Market at Sansome</td>
<td>77</td>
<td>7/1/2015</td>
<td>9:28</td>
<td>2nd at South Park</td>
<td>64</td>
<td>292</td>
</tr>
</tbody>
</table>
```

... (430 rows omitted)

### 2.6.6 are.between()

```ruby
tbl.where(column, are.between(x, y))
```

Filter leaves rows only where the value in `column` is greater than or equal to `x` and less than `y` (i.e. in the interval $[x, y]$).

```
trip = trips.where("Duration", are.between(100, 200))
```
### Trip Data

<table>
<thead>
<tr>
<th>Trip ID</th>
<th>Duration</th>
<th>Start Date</th>
<th>Start Station</th>
<th>Start Terminal</th>
<th>End Date</th>
<th>End Station</th>
<th>End Terminal</th>
<th>Bike #</th>
<th>Subscriber Type</th>
<th>Zip Code</th>
</tr>
</thead>
<tbody>
<tr>
<td>585884</td>
<td>151</td>
<td>12/26/2014 13:34</td>
<td>Broadway St at Battery St</td>
<td>82</td>
<td>12/26/2014 13:37</td>
<td>Harry Bridges Plaza (Ferry Building)</td>
<td>50</td>
<td>576</td>
<td>Subscriber</td>
<td>94107</td>
</tr>
<tr>
<td>548322</td>
<td>191</td>
<td>11/17/2014 20:10</td>
<td>Yerba Buena Center of the Arts (3rd @ Howard)</td>
<td>68</td>
<td>11/17/2014 20:13</td>
<td>Market at Sansome</td>
<td>77</td>
<td>29</td>
<td>Subscriber</td>
<td>94705</td>
</tr>
<tr>
<td>594999</td>
<td>185</td>
<td>1/7/2015 17:53</td>
<td>San Antonio Caltrain Station</td>
<td>29</td>
<td>1/7/2015 17:56</td>
<td>San Antonio Shopping Center</td>
<td>31</td>
<td>176</td>
<td>Subscriber</td>
<td>94040</td>
</tr>
<tr>
<td>468534</td>
<td>194</td>
<td>9/24/2014 19:08</td>
<td>Mechanics Plaza (Market at Battery)</td>
<td>75</td>
<td>9/24/2014 19:11</td>
<td>Harry Bridges Plaza (Ferry Building)</td>
<td>50</td>
<td>443</td>
<td>Subscriber</td>
<td>94107</td>
</tr>
<tr>
<td>873710</td>
<td>169</td>
<td>8/3/2015 17:20</td>
<td>Broadway St at Battery St</td>
<td>82</td>
<td>8/3/2015 17:23</td>
<td>Embarcadero at Sansome</td>
<td>60</td>
<td>532</td>
<td>Subscriber</td>
<td>94114</td>
</tr>
<tr>
<td>853087</td>
<td>168</td>
<td>7/20/2015 7:27</td>
<td>Temporary Transbay Terminal (Howard at Beale)</td>
<td>55</td>
<td>7/20/2015 7:30</td>
<td>2nd at Folsom</td>
<td>62</td>
<td>418</td>
<td>Subscriber</td>
<td>94602</td>
</tr>
<tr>
<td>863019</td>
<td>162</td>
<td>7/27/2015 8:31</td>
<td>Temporary Transbay Terminal (Howard at Beale)</td>
<td>55</td>
<td>7/27/2015 8:34</td>
<td>Mechanics Plaza (Market at Battery)</td>
<td>75</td>
<td>504</td>
<td>Subscriber</td>
<td>94111</td>
</tr>
<tr>
<td>883134</td>
<td>173</td>
<td>8/10/2015 15:11</td>
<td>Embarcadero at Folsom</td>
<td>51</td>
<td>8/10/2015 15:14</td>
<td>Beale at Market</td>
<td>56</td>
<td>363</td>
<td>Subscriber</td>
<td>94117</td>
</tr>
</tbody>
</table>
... (5083 rows omitted)

### 2.6.7 are.between_or_equal_to()

```ruby
tbl.where(column, are.between_or_equal_to(x, y))
```

Filter leaves rows only where the value in column is between or equal to x and y (i.e. in the interval \([x, y]\)).
Filter leaves rows only where the value in `column` is a substring of `string_or_array` if it is a string or an element of `string_or_array` if it is an array.

```sql
[176]: trips.where("Start Station", are.contained_in("2nd at Folsom San Antonio Caltrain Station "))
```

(continues on next page)
(continued from previous page)

<table>
<thead>
<tr>
<th>Trip ID</th>
<th>Duration</th>
<th>Start Date</th>
<th>Start Station</th>
<th>Start Terminal</th>
<th>End Date</th>
<th>End Station</th>
<th>End Terminal</th>
<th>Bike #</th>
<th>Subscriber Type</th>
<th>Zip Code</th>
</tr>
</thead>
<tbody>
<tr>
<td>594999</td>
<td>185</td>
<td>1/7/2015</td>
<td>San Antonio Caltrain Station</td>
<td>29</td>
<td>1/7/2015</td>
<td>San Antonio Shopping Center</td>
<td>31</td>
<td></td>
<td>176</td>
<td>94040</td>
</tr>
<tr>
<td>487432</td>
<td>561</td>
<td>10/7/2014</td>
<td>2nd at Folsom</td>
<td>62</td>
<td>10/7/2014</td>
<td>Commercial at Montgomery</td>
<td>45</td>
<td></td>
<td>342</td>
<td>94105</td>
</tr>
<tr>
<td>610970</td>
<td>808</td>
<td>1/20/2015</td>
<td>2nd at Folsom</td>
<td>62</td>
<td>1/20/2015</td>
<td>Harry Bridges Plaza (Ferry Building)</td>
<td>50</td>
<td></td>
<td>310</td>
<td>94025</td>
</tr>
<tr>
<td>753668</td>
<td>196</td>
<td>5/2/2015</td>
<td>2nd at Folsom</td>
<td>62</td>
<td>5/2/2015</td>
<td>Temporary Transbay Terminal (Howard at Beale)</td>
<td>55</td>
<td></td>
<td>533</td>
<td>94973</td>
</tr>
<tr>
<td>768619</td>
<td>840</td>
<td>5/15/2015</td>
<td>2nd at Folsom</td>
<td>62</td>
<td>5/15/2015</td>
<td>Market at 10th</td>
<td>67</td>
<td></td>
<td>604</td>
<td>94903</td>
</tr>
<tr>
<td>594999</td>
<td>185</td>
<td>1/7/2015</td>
<td>San Antonio Caltrain Station</td>
<td>29</td>
<td>1/7/2015</td>
<td>San Antonio Shopping Center</td>
<td>31</td>
<td></td>
<td>176</td>
<td>94040</td>
</tr>
<tr>
<td>487432</td>
<td>561</td>
<td>10/7/2014</td>
<td>2nd at Folsom</td>
<td>62</td>
<td>10/7/2014</td>
<td>Commercial at Montgomery</td>
<td>45</td>
<td></td>
<td>342</td>
<td>94105</td>
</tr>
<tr>
<td>610970</td>
<td>808</td>
<td>1/20/2015</td>
<td>2nd at Folsom</td>
<td>62</td>
<td>1/20/2015</td>
<td>Harry Bridges Plaza (Ferry Building)</td>
<td>50</td>
<td></td>
<td>310</td>
<td>94025</td>
</tr>
<tr>
<td>753668</td>
<td>196</td>
<td>5/5/2015</td>
<td>2nd at Folsom</td>
<td>62</td>
<td>5/5/2015</td>
<td>Temporary Transbay Terminal (Howard at Beale)</td>
<td>55</td>
<td></td>
<td>533</td>
<td>94973</td>
</tr>
</tbody>
</table>

... (2578 rows omitted)
2.6.9 `are.containing()`

```python
tbl.where(column, are.containing(value))
```

Filter leaves rows only where the value in `column` contains the substring `value`.

```sql
[180]: trips.where("End Station", are.containing("at"))
```

```
<table>
<thead>
<tr>
<th>Trip ID</th>
<th>Duration</th>
<th>Start Date</th>
<th>Start Station</th>
<th>Start Terminal</th>
<th>End Date</th>
<th>End Station</th>
<th>End Terminal</th>
</tr>
</thead>
<tbody>
<tr>
<td>876419</td>
<td>413</td>
<td>8/5/2015</td>
<td>Civic Center BART (7th at Market)</td>
<td></td>
<td>8/5/2015</td>
<td>Townsend at 7th</td>
<td>65</td>
</tr>
<tr>
<td>459672</td>
<td></td>
<td>9/18/2014</td>
<td>Harry Bridges Plaza (Ferry Building)</td>
<td></td>
<td>9/18/2014</td>
<td>Embarcadero at Sansome</td>
<td>60</td>
</tr>
<tr>
<td>903649</td>
<td></td>
<td>8/25/2015</td>
<td>San Francisco Caltrain 2 (330 Townsend)</td>
<td>69</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>452829</td>
<td>409</td>
<td>9/15/2014</td>
<td>Steuart at Market</td>
<td>74</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>491023</td>
<td>224</td>
<td>10/9/2014</td>
<td>Santa Clara at Almaden</td>
<td>4</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>723352</td>
<td>519</td>
<td>4/13/2015</td>
<td>Howard at 2nd</td>
<td>63</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>524499</td>
<td>431</td>
<td>10/31/2014</td>
<td>Townsend at 7th</td>
<td>65</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>518524</td>
<td>389</td>
<td>10/28/2014</td>
<td>Market at Sansome</td>
<td>77</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>793149</td>
<td>616</td>
<td>6/4/2015</td>
<td>Embarcadero at Bryant</td>
<td>54</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>681771</td>
<td>895</td>
<td>3/14/2015</td>
<td>Market at 10th</td>
<td>67</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>681771</td>
<td>895</td>
<td>3/14/2015</td>
<td>Market at 10th</td>
<td>67</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>416</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
</tbody>
</table>
```

... (78805 rows omitted)
2.6.10 `are.strictly_between()`

```python
tbl.where(column, are.strictly_between(x, y))
```

Filter leaves rows only where the value in `column` is strictly greater than `x` and less than `y` (i.e. in the interval `(x, y)`).

```sql
[181]: trips.where("Duration", are.strictly_between(100, 200))
```

<table>
<thead>
<tr>
<th>Trip ID</th>
<th>Duration</th>
<th>Start Date</th>
<th>Start Station</th>
<th>Start Terminal</th>
<th>End Date</th>
<th>End Station</th>
<th>End Terminal</th>
<th>Bike #</th>
<th>Subscriber Type</th>
<th>Zip Code</th>
</tr>
</thead>
<tbody>
<tr>
<td>585884</td>
<td>151</td>
<td>12/26/2014 13:34</td>
<td>Broadway St at Battery St</td>
<td>82</td>
<td>12/26/2014 13:37</td>
<td>Harry Bridges Plaza (Ferry Building)</td>
<td>50</td>
<td>576</td>
<td>94107</td>
<td></td>
</tr>
<tr>
<td>548322</td>
<td>191</td>
<td>11/17/2014 20:10</td>
<td>Yerba Buena Center of the Arts (3rd @ Howard)</td>
<td>68</td>
<td>11/17/2014 20:13</td>
<td>Market at Sansome</td>
<td>77</td>
<td>29</td>
<td>94705</td>
<td></td>
</tr>
<tr>
<td>594999</td>
<td>185</td>
<td>1/7/2015 17:53</td>
<td>San Antonio Caltrain Station</td>
<td>29</td>
<td>1/7/2015 17:56</td>
<td>San Antonio Shopping Center</td>
<td>31</td>
<td>176</td>
<td>94040</td>
<td></td>
</tr>
<tr>
<td>468534</td>
<td>194</td>
<td>9/24/2014 19:08</td>
<td>Mechanics Plaza (Market at Battery)</td>
<td>75</td>
<td>9/24/2014 19:11</td>
<td>Harry Bridges Plaza (Ferry Building)</td>
<td>50</td>
<td>443</td>
<td>94107</td>
<td></td>
</tr>
<tr>
<td>873710</td>
<td>169</td>
<td>8/3/2015 17:20</td>
<td>Broadway St at Battery St</td>
<td>82</td>
<td>8/3/2015 17:23</td>
<td>Embarcadero at Sansome</td>
<td>60</td>
<td>532</td>
<td>94705</td>
<td></td>
</tr>
<tr>
<td>853087</td>
<td>168</td>
<td>7/20/2015 7:27</td>
<td>Temporary Transbay Terminal (Howard at Beale)</td>
<td>55</td>
<td>7/20/2015 7:30</td>
<td>2nd at Folsom</td>
<td>62</td>
<td>418</td>
<td>94602</td>
<td></td>
</tr>
<tr>
<td>863019</td>
<td>162</td>
<td>7/27/2015 8:31</td>
<td>Temporary Transbay Terminal (Howard at Beale)</td>
<td>55</td>
<td>7/27/2015 8:34</td>
<td>Mechanics Plaza (Market at Battery)</td>
<td>75</td>
<td>504</td>
<td>94114</td>
<td></td>
</tr>
<tr>
<td>883134</td>
<td>173</td>
<td>8/10/2015 15:11</td>
<td>Embarcadero at Folsom</td>
<td>51</td>
<td>8/10/2015 15:14</td>
<td>Beale at Market</td>
<td>56</td>
<td>363</td>
<td>94117</td>
<td></td>
</tr>
</tbody>
</table>

... (5056 rows omitted)

2.7 Miscellaneous Functions

2.7.1 `sample_proportions()`

```python
sample_proportions(sample_size, model_proportions)
```

Samples `sample_size` objects from the distribution specified by `model_proportions`. `sample_size` should be an integer, `model_proportions` an array of probabilities that sum up to 1. It returns an array with the same size...
as `model_proportions`. Each item in the array corresponds to the proportion of times it was sampled out of the `sample_size` times.

```
[182]: sample_proportions(100, [.5, .3, .2])
array([0.32, 0.32, 0.36])
```

### 2.7.2 minimize()

```
minimize(function)
```

This function returns an array of values that minimize `function`. `function` should be a function that takes in a certain number of arguments and returns a number. The array returned by `minimize` is structured such that if each value in the array was passed into `function` as arguments, it would minimize the output value of `function`.

```
[190]: def f(x, y):
    \[\text{return } 0.47 \times \text{x}^2 + 1.23 \times \text{np.log}(y)\]
minimize(f)
array([ 5.17585792, -0.58835469])
```
3.1 Tables (datascience.tables)

Summary of methods for Table. Click a method to see its documentation.

One note about reading the method signatures for this page: each method is listed with its arguments. However, optional arguments are specified in brackets. That is, a method that’s documented like

Table.foo (first_arg, second_arg[, some_other_arg, fourth_arg])

means that the Table.foo method must be called with first_arg and second_arg and optionally some_other_arg and fourth_arg. That means the following are valid ways to call Table.foo:

```python
some_table.foo(1, 2)
some_table.foo(1, 2, 'hello')
some_table.foo(1, 2, 'hello', 'world')
some_table.foo(1, 2, some_other_arg='hello')
```

But these are not valid:

```python
some_table.foo(1) # Missing arg
some_table.foo(1, 2[, 'hi']) # SyntaxError
some_table.foo(1, 2[, 'hello', 'world']) # SyntaxError
```

If that syntax is confusing, you can click the method name itself to get to the details page for that method. That page will have a more straightforward syntax.

Creation

<table>
<thead>
<tr>
<th>Method</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td><code>Table.__init__</code>([labels, formatter])</td>
<td>Create an empty table with column labels.</td>
</tr>
<tr>
<td><code>Table.from_records</code>(records)</td>
<td>Create a table from a sequence of records (dicts with fixed keys).</td>
</tr>
<tr>
<td><code>Table.from_columns_dict</code>(columns)</td>
<td>Create a table from a mapping of column labels to column values.</td>
</tr>
<tr>
<td><code>Table.read_table</code>(filepath_or_buffer, *args, ...)</td>
<td>Read a table from a file or web address.</td>
</tr>
<tr>
<td><code>Table.from_df</code>(df[, keep_index])</td>
<td>Convert a Pandas DataFrame into a Table.</td>
</tr>
<tr>
<td><code>Table.from_array</code>(arr)</td>
<td>Convert a structured NumPy array into a Table.</td>
</tr>
</tbody>
</table>
3.1.1 datascience.tables.Table.__init__

Table.__init__ (labels=None, formatter=<datascience.formats.Formatter object>)
Create an empty table with column labels.

```python
def tiles = Table(make_array('letter', 'count', 'points'))
```

Args:
- labels (list of strings): The column labels.
- formatter (Formatter): An instance of Formatter that formats the columns’ values.

3.1.2 datascience.tables.Table.from_records

classmethod Table.from_records (records)
Create a table from a sequence of records (dicts with fixed keys).

Args:
- records: A list of dictionaries with same keys.

Returns:
- If the list is empty, it will return an empty table. Otherwise, it will return a table with the dictionary’s keys as the column name, and the corresponding data. If the dictionaries do not have identical keys, the keys of the first dictionary in the list is used.

Example:

```python
def t = Table().from_records([...
    ...    '{column1':'data1','column2':1},
    ...    '{column1':'data2','column2':2},
    ...    '{column1':'data3','column2':3}
    ... ])
```

3.1.3 datascience.tables.Table.from_columns_dict

classmethod Table.from_columns_dict (columns)
Create a table from a mapping of column labels to column values. [Deprecated]
3.1.4 datascience.tables.Table.read_table

classmethod Table.read_table(filepath_or_buffer, *args, **vargs)

Read a table from a file or web address.

Args:

filepath_or_buffer – string or file handle / StringIO; The string
could be a URL. Valid URL schemes include http, ftp, s3, and file.

Returns:

a table read from argument

Example:

```python
>>> Table.read_table('https://www.inferentialthinking.com/data/sat2014.csv')
```

<table>
<thead>
<tr>
<th>State</th>
<th>Participation Rate</th>
<th>Critical Reading</th>
<th>Math</th>
<th>Writing</th>
<th>Combined</th>
</tr>
</thead>
<tbody>
<tr>
<td>North Dakota</td>
<td>2.3</td>
<td>612</td>
<td>620</td>
<td>584</td>
<td>1816</td>
</tr>
<tr>
<td>Illinois</td>
<td>4.6</td>
<td>599</td>
<td>616</td>
<td>587</td>
<td>1802</td>
</tr>
<tr>
<td>Iowa</td>
<td>3.1</td>
<td>605</td>
<td>611</td>
<td>578</td>
<td>1794</td>
</tr>
<tr>
<td>South Dakota</td>
<td>2.9</td>
<td>604</td>
<td>609</td>
<td>579</td>
<td>1792</td>
</tr>
<tr>
<td>Minnesota</td>
<td>5.9</td>
<td>598</td>
<td>610</td>
<td>578</td>
<td>1786</td>
</tr>
<tr>
<td>Michigan</td>
<td>3.8</td>
<td>593</td>
<td>610</td>
<td>581</td>
<td>1784</td>
</tr>
<tr>
<td>Wisconsin</td>
<td>3.9</td>
<td>596</td>
<td>608</td>
<td>578</td>
<td>1782</td>
</tr>
<tr>
<td>Missouri</td>
<td>4.2</td>
<td>595</td>
<td>597</td>
<td>579</td>
<td>1771</td>
</tr>
<tr>
<td>Wyoming</td>
<td>3.3</td>
<td>590</td>
<td>599</td>
<td>573</td>
<td>1762</td>
</tr>
<tr>
<td>Kansas</td>
<td>5.3</td>
<td>591</td>
<td>596</td>
<td>566</td>
<td>1753</td>
</tr>
</tbody>
</table>

... (41 rows omitted)

3.1.5 datascience.tables.Table.from_df

classmethod Table.from_df(df, keep_index=False)

Convert a Pandas DataFrame into a Table.

Args:

df – Pandas DataFrame utilized for creation of Table

keep_index – keeps the index of the DataFrame and turns it into a column called index in the new
Table

Returns:

a table from Pandas Dataframe in argument

Example:

```python
>>> sample_DF = pandas.DataFrame(
...     data = zip([1,2,3],['a','b','c'],[data1,'data2','data3']),
...     columns = ['column1','column2','column3'])
...)
```

```python
>>> sample_DF
          column1  column2  column3
0        1         a     data1
```
1 2 b data2
2 3 c data3

>>> t = Table().from_df(sample_DF)

>>> t
column1 | column2 | column3
1 | a | data1
2 | b | data2
3 | c | data3

3.1.6 datascience.tables.Table.from_array

classmethod Table.from_array(arr)

Convert a structured NumPy array into a Table.

Args:

arr – A structured NumPy array

Returns:

A table with the field names as the column names and the corresponding data.

Example:

```python
>>> arr = np.array([
... ('A',1), ('B',2),
... dtype=[('Name', 'U10'), ('Number', 'i4')]
... )

>>> arr
array([(b'A', 1), (b'B', 2)], dtype=[(b'Name', '<U10'), (b'Number', '<i4')])

>>> t = Table().from_array(arr)

>>> t
Name | Number
A   | 1
B   | 2
```

Extension (does not modify original table)

| Table.with_column(label, values[, formatter]) | Return a new table with an additional or replaced column. |
| Table.with_columns(*labels_and_values,...) | Return a table with additional or replaced columns. |
| Table.with_row(row) | Return a table with an additional row. |
| Table.with_rows(rows) | Return a table with additional rows. |
| Table.relabeled(label, new_label) | Return a new table with label specifying column label(s) replaced by corresponding new_label. |
### 3.1.7 `datascience.tables.Table.with_column`

`Table.with_column(label, values, formatter=None)`

Return a new table with an additional or replaced column.

**Args:**
- `label` (str): The column label. If an existing label is used, the existing column will be replaced in the new table.
- `values` (single value or sequence): If a single value, every value in the new column is `values`. If sequence of values, new column takes on values in `values`.
- `formatter` (single value): Specifies formatter for the new column. Defaults to no formatter.

**Raises:**
- `ValueError`: If
  - `label` is not a valid column name
  - `if label` is not of type (str)
  - `values` is a list/array that does not have the same length as the number of rows in the table.

**Returns:**
- copy of original table with new or replaced column

```python
>>> alphabet = Table().with_column('letter', make_array('c', 'd'))
>>> alphabet = alphabet.with_column('count', make_array(2, 4))
>>> alphabet
<table>
<thead>
<tr>
<th>letter</th>
<th>count</th>
</tr>
</thead>
<tbody>
<tr>
<td>c</td>
<td>2</td>
</tr>
<tr>
<td>d</td>
<td>4</td>
</tr>
</tbody>
</table>
>>> alphabet.with_column('permutes', make_array('a', 'g'))
<table>
<thead>
<tr>
<th>letter</th>
<th>count</th>
<th>permutes</th>
</tr>
</thead>
<tbody>
<tr>
<td>c</td>
<td>2</td>
<td>a</td>
</tr>
<tr>
<td>d</td>
<td>4</td>
<td>g</td>
</tr>
</tbody>
</table>
>>> alphabet
<table>
<thead>
<tr>
<th>letter</th>
<th>count</th>
</tr>
</thead>
<tbody>
<tr>
<td>c</td>
<td>2</td>
</tr>
<tr>
<td>d</td>
<td>4</td>
</tr>
</tbody>
</table>
>>> alphabet.with_column('count', 1)
<table>
<thead>
<tr>
<th>letter</th>
<th>count</th>
</tr>
</thead>
<tbody>
<tr>
<td>c</td>
<td>1</td>
</tr>
<tr>
<td>d</td>
<td>1</td>
</tr>
</tbody>
</table>
>>> alphabet.with_column(1, make_array(1, 2))
Traceback (most recent call last):
...
ValueError: The column label must be a string, but a int was given
>>> alphabet.with_column('bad_col', make_array(1))
Traceback (most recent call last):
...
ValueError: Column length mismatch. New column does not have the same number of rows as table.
```
3.1.8 `datascience.tables.Table.with_columns`

`Table.with_columns(*labels_and_values, **formatter)`

Return a table with additional or replaced columns.

**Args:**

- `labels_and_values`: An alternating list of labels and values
  or a list of label-value pairs. If one of the labels is in existing table, then every value in the corresponding column is set to that value. If label has only a single value (int), every row of corresponding column takes on that value.

- “formatter” (single Formatter value): A single formatter value
  that will be applied to all columns being added using this function call.

**Raises:**

ValueError: If

- any label in `labels_and_values` is not a valid column name, i.e if label is not of type (str).
- if any value in `labels_and_values` is a list/array and does not have the same length as the number of rows in the table.

AssertionError:

- ‘incorrect columns format’, if passed more than one sequence (iterables) for `labels_and_values`.
- ‘even length sequence required’ if missing a pair in label-value pairs.

**Returns:**

Copy of original table with new or replaced columns. Columns added in order of labels. Equivalent to `with_column(label, value)` when passed only one label-value pair.

```python
>>> players = Table().with_columns('player_id',
...     make_array(110234, 110235), 'wOBA', make_array(.354, .236))
>>> players
player_id | wOBA
110234 | 0.354
110235 | 0.236
>>> players = players.with_columns('salaries', 'N/A', 'season', 2016)
>>> players
player_id | wOBA | salaries | season
110234 | 0.354 | N/A | 2016
110235 | 0.236 | N/A | 2016
>>> salaries = Table().with_column('salary',
...     make_array(500000, 15500000))
>>> players.with_columns('salaries', salaries.column('salary'),
...     'bonus', make_array(6, 1), formatter=_formats.CurrencyFormatter)
player_id | wOBA | salaries | season | bonus
110234 | 0.354 | $500,000 | 2016 | $6
110235 | 0.236 | $15,500,000 | 2016 | $1
>>> players.with_columns(2, make_array('600,000', '20,000,000'))
Traceback (most recent call last):
  ...
ValueError: The column label must be a string, but a int was given

>>> players.with_columns('salaries', make_array('$600,000'))
Traceback (most recent call last):
...
ValueError: Column length mismatch. New column does not have the same number of rows as table.

3.1.9 datascience.tables.Table.with_row

Table.with_row(row)

Return a table with an additional row.

Args:
row (sequence): A value for each column.

Raises:
ValueError: If the row length differs from the column count.

>>> tiles = Table(make_array('letter', 'count', 'points'))
>>> tiles.with_row(['c', 2, 3]).with_row(['d', 4, 2])
letter | count | points
--- | --- | ---
c   | 2     | 3
--- | --- | ---
d   | 4     | 2

3.1.10 datascience.tables.Table.with_rows

Table.with_rows(rows)

Return a table with additional rows.

Args:
rows (sequence of sequences): Each row has a value per column.

If rows is a 2-d array, its shape must be (_, n) for n columns.

Raises:
ValueError: If a row length differs from the column count.

>>> tiles = Table(make_array('letter', 'count', 'points'))
>>> tiles.with_rows(make_array(make_array('c', 2, 3),
... make_array('d', 4, 2)))
letter | count | points
--- | --- | ---
c   | 2     | 3
--- | --- | ---
d   | 4     | 2
3.1.11 datascience.tables.Table.relabeled

```
Table.relabeled(label, new_label)
```

Return a new table with `label` specifying column label(s) replaced by corresponding `new_label`.

**Args:**
- `label` – (str or array of str) The label(s) of columns to be changed.
- `new_label` – (str or array of str): The new label(s) of columns to be changed. Same number of elements as label.

**Raises:**
- ValueError – if `label` does not exist in table, or if the `label` and `new_label` are not not of equal length. Also, raised if `label` and/or `new_label` are not str.

**Returns:**
New table with `new_label` in place of `label`.

```python
>>> tiles = Table().with_columns('letter', make_array('c', 'd'),
                    ...   'count', make_array(2, 4))
>>> tiles
letter | count
  c   | 2
  d   | 4
>>> tiles.relabeled('count', 'number')
nletter | number
  c   | 2
  d   | 4
>>> tiles  # original table unmodified
letter | count
  c   | 2
  d   | 4
>>> tiles.relabeled(make_array('letter', 'count'),
                    ...   make_array('column1', 'column2'))
column1 | column2
  c   | 2
  d   | 4
>>> tiles.relabeled(make_array('letter', 'number'),
                    ...   make_array('column1', 'column2'))
Traceback (most recent call last):
  ...
ValueError: Invalid labels. Column labels must already exist in table in order to be replaced.
```

Accessing values
### 3.1.12 `datascience.tables.Table.num_columns`

**property** `Table.num_columns`

Number of columns.

### 3.1.13 `datascience.tables.Table.columns`

**property** `Table.columns`

Return a tuple of columns, each with the values in that column.

**Returns:**

tuple of columns

**Example:**

```python
>>> t = Table().with_columns({
...   'letter': ['a', 'b', 'c', 'z'],
...   'count': [9, 3, 3, 1],
...   'points': [1, 2, 2, 10],
... })
>>> t.columns
(array(['a', 'b', 'c', 'z'], dtype='<U1'),
array([9, 3, 3, 1]),
array([1, 2, 2, 10]))
```

### 3.1.14 `datascience.tables.Table.column`

**Table.column(index_or_label)**

Return the values of a column as an array.

table.column(label) is equivalent to table[label].

```python
>>> tiles = Table().with_columns(
...   'letter', make_array('c', 'd'),
...)
``` (continues on next page)
... 'count', make_array(2, 4),
...)

```python
>>> list(tiles.column('letter'))
['c', 'd']
>>> tiles.column(1)
array([2, 4])
```

**Args:**
- label (int or str): The index or label of a column

**Returns:**
- An instance of `numpy.array`

**Raises:**
- `ValueError`: When the `index_or_label` is not in the table.

### 3.1.15 `datascience.tables.Table.num_rows`

**property** `Table.num_rows`

Computes the number of rows in a table

**Returns:**
- integer value stating number of rows

**Example:**

```python
>>> t = Table().with_columns({
...    'letter': [ 'a', 'b', 'c', 'z' ],
...    'count': [ 9, 3, 3, 1 ],
...    'points': [ 1, 2, 2, 10 ],
...})
>>> t.num_rows
4
```

### 3.1.16 `datascience.tables.Table.rows`

**property** `Table.rows`

Return a view of all rows.

**Returns:**
- list-like `Rows` object that contains tuple-like `Row` objects

**Example:**

```python
>>> t = Table().with_columns({
...    'letter': [ 'a', 'b', 'c', 'z' ],
...    'count': [ 9, 3, 3, 1 ],
...    'points': [ 1, 2, 2, 10 ],
...})
>>> t.rows
Rows(letter | count | points
```

(continues on next page)
### 3.1.17 datascience.tables.Table.row

**Table.row(index)**

Return a row.

Please see extended docstring at https://github.com/data-8/datascience/blob/614db00e7d22e52683860d2beaa4037bec26cf87/datascience/tables.py#L5673-L5765 for how to interact with Rows.

### 3.1.18 datascience.tables.Table.labels

**property Table.labels**

Return a tuple of column labels.

**Returns:**

tuple of labels

**Example:**

```python
>>> t = Table().with_columns({
...    'letter': ['a', 'b', 'c', 'z'],
...    'count': [9, 3, 3, 1],
...    'points': [1, 2, 2, 10],
... })
>>> t.labels
('letter', 'count', 'points')
```

### 3.1.19 datascience.tables.Table.first

**Table.first(label)**

Return the zeroth item in a column.

**Args:**

- **label** (str) – value of column label

**Returns:**

zeroth item of column

**Example:**

```python
>>> t = Table().with_columns({
...    'letter': ['a', 'b', 'c', 'z'],
...    'count': [9, 3, 3, 1],
...    'points': [1, 2, 2, 10],
... })
```
>>> t.first('letter')
'a'

3.1.20 datascience.tables.Table.last

Table.last(label)
Return the last item in a column.

Args:
  label (str) – value of column label

Returns:
  last item of column

Example:

```python
t = Table().with_columns({
  'letter': ['a', 'b', 'c', 'z'],
  'count': [ 9, 3, 3, 1],
  'points': [ 1, 2, 2, 10],
})
>>> t.last('letter')
'z'
```

3.1.21 datascience.tables.Table.values

property Table.values
Return data in self as a numpy array.

If all columns are the same dtype, the resulting array will have this dtype. If there are >1 dtypes in columns, then the resulting array will have dtype object.

Example:

```python
tiles = Table().with_columns(
  'letter', make_array('c', 'd'),
  'count', make_array(2, 4),
)
>>> tiles.values
array([[c', 2],
       ['d', 4]], dtype=object)
t = Table().with_columns(
  'col1', make_array(1, 2),
  'col2', make_array(3, 4),
)
>>> t.values
array([[1, 3],
       [2, 4]])
```
### 3.1.22 datascience.tables.Table.column_index

**Table.column_index(label)**

Return the index of a column by looking up its label.

**Args:**
- `label` (str) – label value of a column

**Returns:**
- integer value specifying the index of the column label

**Example:**

```python
t = Table().with_columns({
...     'letter': ['a', 'b', 'c', 'z'],
...     'count': [9, 3, 3, 1],
...     'points': [1, 2, 2, 10],
... })
t.column_index('letter')
```

### 3.1.23 datascience.tables.Table.apply

**Table.apply(fn, *column_or_columns)**

Apply `fn` to each element or elements of `column_or_columns`. If no `column_or_columns` provided, `fn` is applied to each row.

**Args:**
- `fn` (function) – The function to apply to each element of `column_or_columns`.
- `column_or_columns` – Columns containing the arguments to `fn` as either column labels (str) or column indices (int). The number of columns must match the number of arguments that `fn` expects.

**Raises:**
- `ValueError` – if `column_label` is not an existing column in the table.
- `TypeError` – if insufficient number of `column_label` passed to `fn`.

**Returns:**
- An array consisting of results of applying `fn` to elements specified by `column_label` in each row.

```python
t = Table().with_columns(
...     'letter', make_array('a', 'b', 'c', 'z'),
...     'count', make_array(9, 3, 3, 1),
...     'points', make_array(1, 2, 2, 10))
t
```

```
>>> t.apply(lambda x: x - 1, 'points')
array([ 0,  1, 10])
```

```
>>> t.apply(lambda x, y: x * y, 'count', 'points')
array([ 9,  6,  6, 10])
```

```
>>> t.apply(lambda x: x - 1, 'count', 'points')
Traceback (most recent call last):
  ...  
TypeError: <lambda>() takes 1 positional argument but 2 were given
```

```
>>> t.apply(lambda row: row[1] * 2)
array([18,  6,  6,  2])
```

Whole rows are passed to the function if no columns are specified.

### 3.1.24 datascience.tables.Table.set_format

```Python
Table.set_format(column_or_columns, formatter)
```

Set the pretty print format of a column(s) and/or convert its values.

**Args:**
- `column_or_columns`: values to group (column label or index, or array)
- `formatter`: a function applied to a single value within the column_or_columns at a time or a formatter class, or formatter class instance.

**Returns:**
- A Table with `formatter` applied to each column_or_columns.

The following example formats the column “balance” by passing in a formatter class instance. The formatter is run each time `__repr__` is called. So, while the table is formatted upon being printed to the console, the underlying values within the table remain untouched. It's worth noting that while `set_format` returns the table with the new formatters applied, this change is applied directly to the original table and then the original table is returned. This means `set_format` is what's known as an inplace operation.
The following example formats the column “balance” by passing in a formatter function.

```python
>>> account_info = Table().with_columns(
    ...    "user", make_array("gfoo", "bbar", "tbaz", "hbat"),
    ...    "balance", make_array(200, 555, 125, 430))
>>> account_info
user | balance
--- | ---
gfoo | 200
bbar | 555
tbaz | 125
hbat | 430

>>> from datascience.formats import CurrencyFormatter
>>> account_info.set_format("balance", CurrencyFormatter("BZ$")) # Belize Dollar
>>> account_info
user | balance
--- | ---
gfoo | BZ$200
bbar | BZ$555
tbaz | BZ$125
hbat | BZ$430
```

The following, formats the column “balance” by passing in a formatter class. Note the formatter class must have a Boolean `converts_values` attribute set and a `format_column` method that returns a function that formats a single value at a time. The `format_column` method accepts the name of the column and the value of the column as arguments and returns a formatter function that accepts a value and Boolean indicating whether that value is the column name. In the following example, if the `if label: return value` was removed, the column name “balance” would be formatted and printed as “balance kr”. The `converts_values` attribute should be set to False unless a `convert_values` method is also defined on the formatter class.

```python
>>> def iceland_krona_formatter(value):
...    return f"{value} kr"

>>> account_info.set_format("balance", iceland_krona_formatter)
>>> account_info
user | balance
--- | ---
gfoo | 200 kr
bbar | 555 kr
tbaz | 125 kr
hbat | 430 kr
```

The following formats the column “balance” by passing in a formatter class. Note the formatter class must have a Boolean `converts_values` attribute set and a `format_column` method that returns a function that formats a single value at a time. The `format_column` method accepts the name of the column and the value of the column as arguments and returns a formatter function that accepts a value and Boolean indicating whether that value is the column name. In the following example, if the `if label: return value` was removed, the column name “balance” would be formatted and printed as “balance kr”. The `converts_values` attribute should be set to False unless a `convert_values` method is also defined on the formatter class.
```python
>>> account_info = Table().with_columns(
...     "user", make_array("gfoo", "bbar", "tbaz", "hbat"),
...     "balance", make_array(200, 555, 125, 430))
>>> account_info
user | balance
--- | ---
gfoo | 200
bbar | 555
tbaz | 125
hbat | 430

```  

```python
class IcelandKronaFormatter():
...
    def __init__(self):
        self.converts_values = False
...
    def format_column(self, label, column):
        def format_krona(value, label):
            if label:
                return value
            return f"{value} kr"
...
    return format_krona
```  

```python
account_info.set_format("balance", IcelandKronaFormatter)
user | balance
--- | ---
gfoo | 200 kr
bbar | 555 kr
tbaz | 125 kr
hbat | 430 kr
```  

`account_info["balance"]`  
array([200, 555, 125, 430])

set_format can also be used to convert values. If you set the converts_values attribute to True and define a convert_column method that accepts the column values and returns the converted column values on the formatter class, the column values will be permanently converted to the new column values in a one time operation.

```python
>>> account_info = Table().with_columns(
...     "user", make_array("gfoo", "bbar", "tbaz", "hbat"),
...     "balance", make_array(200.01, 555.55, 125.65, 430.18))
>>> account_info
user | balance
--- | ---
gfoo | 200.01
bbar | 555.55
tbaz | 125.65
hbat | 430.18
```  

```python
class IcelandKronaFormatter():
...
    def __init__(self):
        self.converts_values = True
...
    def format_column(self, label, column):
        def format_krona(value, label):
            if label:
                return value
            return f"{value} kr"
...
    return format_krona
```  

(continues on next page)
In the following example, multiple columns are configured to use the same formatter. Note the following formatter takes into account the length of all values in the column and formats them to be the same character length. This is something that the default table formatter does for you but, if you write a custom formatter, you must do yourself.

```python
>>> class IcelandKronaFormatter:
...     def __init__(self):
...         self.converts_values = False
...     def format_column(self, label, column):
...         val_width = max([len(str(v)) + len(" kr") for v in column])
...         val_width = max(len(str(label)), val_width)
...     def format_krona(self, value, label):
...         if label:
...             return value
...         return f"{value} kr".ljust(val_width)
...     return format_krona

>>> account_info.set_format(["checking", "savings"], IcelandKronaFormatter)
user | checking | savings
-----|----------|---------
gfoo | 200 kr   | 1000 kr
bbar | 555 kr   | 500     
tbaz | 125 kr   | 1175    
hbat | 430 kr   | 6700
```
3.1.25 datascience.tables.Table.move_to_start

Table.move_to_start(column_label)
Move a column to be the first column.

The following example moves column C to be the first column. Note, move_to_start not only returns the original table with the column moved but, it also moves the column in the original. This is what’s known as an inplace operation.

```python
>>> table = Table().with_columns(
    ... "A", make_array(1, 2, 3, 4),
    ... "B", make_array("foo", "bar", "baz", "bat"),
    ... "C", make_array('a', 'b', 'c', 'd'))
>>> table.move_to_start("C")
>>> table
A | B | C
1 | foo | a
2 | bar | b
3 | baz | c
4 | bat | d
>>> table
C | A | B
a | 1 | foo
b | 2 | bar
c | 3 | baz
d | 4 | bat
```

3.1.26 datascience.tables.Table.move_to_end

Table.move_to_end(column_label)
Move a column to be the last column.

The following example moves column A to be the last column. Note, move_to_end not only returns the original table with the column moved but, it also moves the column in the original. This is what’s known as an inplace operation.

```python
>>> table = Table().with_columns(
    ... "A", make_array(1, 2, 3, 4),
    ... "B", make_array("foo", "bar", "baz", "bat"),
    ... "C", make_array('a', 'b', 'c', 'd'))
>>> table
A | B | C
```

### 3.1.27 `datascience.tables.Table.append`

`Table.append(row_or_table)`

Append a row or all rows of a table in place. An appended table must have all columns of self.

The following example appends a row record to the table, followed by appending a table having all columns of self.

```python
gg> table = Table().with_columns(  
...     "A", make_array(1),  
...     "B", make_array("foo"),  
...     "C", make_array('a'))
```  
```python
gg> table  
A | B | C  
1 | foo | a
```  
```python
gg> table.append([2, "bar", 'b'])  
A | B | C  
1 | foo | a  
2 | bar | b
```  
```python
gg> table  
A | B | C  
1 | foo | a  
2 | bar | b
```  
```python
gg> table.append(Table().with_columns(  
...     "A", make_array(3, 4),  
...     "B", make_array("baz", "bat"),  
...     "C", make_array('c', 'd')))  
A | B | C  
1 | foo | a  
2 | bar | b  
3 | baz | c  
4 | bat | d
```  
```python
gg> table  
A | B | C  
1 | foo | a  
2 | bar | b  
3 | baz | c  
4 | bat | d
```  
(continues on next page)
1 | foo | a
2 | bar | b
3 | baz | c
4 | bat | d

3.1.28 datascience.tables.Table.append_column

Table.append_column(label, values, formatter=None)

Appends a column to the table or replaces a column.

__setitem__ is aliased to this method:

table.append_column('new_col', make_array(1, 2, 3)) is equivalent to table['new_col']
= make_array(1, 2, 3).

Args:

  label (str): The label of the new column.

  values (single value or list/array): If a single value, every
  value in the new column is values.

  If a list or array, the new column contains the values in values, which must be the same length as the
  table.

  formatter (single formatter): Adds a formatter to the column being
  appended. No formatter added by default.

Returns:

  Original table with new or replaced column

Raises:

  ValueError: If
  - label is not a string.
  - values is a list/array and does not have the same length as the number of rows in the table.

```python
>>> table = Table().with_columns(
...   'letter', make_array('a', 'b', 'c', 'z'),
...   'count', make_array(9, 3, 3, 1),
...   'points', make_array(1, 2, 2, 10))

>>> table
letter | count | points
-- | -- | --
a | 9 | 1
b | 3 | 2
c | 3 | 2
z | 1 | 10

>>> table.append_column('new_col1', make_array(10, 20, 30, 40))
letter | count | points | new_col1
-- | -- | -- | --
a | 9 | 1 | 10
b | 3 | 2 | 20
c | 3 | 2 | 30
z | 1 | 10 | 40

>>> table.append_column('new_col2', 'hello')
letter | count | points | new_col1 | new_col2
-- | -- | -- | -- | --
a | 9 | 1 | 10 | hello
b | 3 | 2 | 20 | hello
c | 3 | 2 | 30 | hello
z | 1 | 10 | 40 | hello
```

(continues on next page)
>>> table.append_column(123, make_array(1, 2, 3, 4))
Traceback (most recent call last):
...
ValueError: The column label must be a string, but a int was given

>>> table.append_column('bad_col', [1, 2])
Traceback (most recent call last):
...
ValueError: Column length mismatch. New column does not have the same number of rows as table.

3.1.29 datascience.tables.Table.relabel

Table.relabel(column_label, new_label)
Changes the label(s) of column(s) specified by column_label to labels in new_label.

Args:

column_label – (single str or array of str) The label(s) of columns to be changed to new_label.

new_label – (single str or array of str): The label name(s) of columns to replace column_label.

Raises:

ValueError – if column_label is not in table, or if column_label and new_label are not of equal length.

TypeError – if column_label and/or new_label is not str.

Returns:

Original table with new_label in place of column_label.

>>> table = Table().with_columns(
...   'points', make_array(1, 2, 3),
...   'id', make_array(12345, 123, 5123))

>>> table.relabel('id', 'yolo')
points | yolo
1 | 12345
2 | 123
3 | 5123

>>> table.relabel(make_array('points', 'yolo'),
... make_array('red', 'blue'))
red | blue
1 | 12345
2 | 123
3 | 5123

>>> table.relabel(make_array('red', 'green', 'blue'),
... make_array('cyan', 'magenta', 'yellow', 'key'))
3.1.30 datascience.tables.Table.remove

**Table.remove(row_or_row_indices)**

Removes a row or multiple rows of a table in place (row number is 0 indexed). If row_or_row_indices is not int or list, no changes will be made to the table.

The following example removes 2nd row (row_or_row_indices = 1), followed by removing 2nd and 3rd rows (row_or_row_indices = [1, 2]).

```python
generate code
```
<table>
<thead>
<tr>
<th>Method</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td><code>Table.copy(*[, shallow])</code></td>
<td>Return a copy of a table.</td>
</tr>
<tr>
<td><code>Table.select(*column_or_columns)</code></td>
<td>Return a table with only the columns in <code>column_or_columns</code>.</td>
</tr>
<tr>
<td><code>Table.drop(*column_or_columns)</code></td>
<td>Return a Table with only columns other than selected label or labels.</td>
</tr>
<tr>
<td><code>Table.take()</code></td>
<td>Return a new Table with selected rows taken by index.</td>
</tr>
<tr>
<td><code>Table.exclude()</code></td>
<td>Return a new Table without a sequence of rows excluded by number.</td>
</tr>
<tr>
<td><code>Table.move_column(label, index)</code></td>
<td>Returns a new table with specified column moved to the specified column index.</td>
</tr>
<tr>
<td><code>Table.where(column_or_label[, ...])</code></td>
<td>Return a new Table containing rows where <code>value_or_predicate</code> returns True for values in <code>column_or_label</code>.</td>
</tr>
<tr>
<td><code>Table.sort(column_or_label[, descending, ...])</code></td>
<td>Return a Table of rows sorted according to the values in a column.</td>
</tr>
<tr>
<td><code>Table.group(column_or_label[, collect])</code></td>
<td>Group rows by unique values in a column; count or aggregate others.</td>
</tr>
<tr>
<td><code>Table.groups(labels[, collect])</code></td>
<td>Group rows by multiple columns, count or aggregate others.</td>
</tr>
<tr>
<td><code>Table.pivot(columns, rows[, values, ...])</code></td>
<td>Generate a table with a column for each unique value in <code>columns</code>, with rows for each unique value in <code>rows</code>.</td>
</tr>
<tr>
<td><code>Table.stack(key[, labels])</code></td>
<td>Takes k original columns and returns two columns, with col.</td>
</tr>
<tr>
<td><code>Table.join(column_label, other[, other_label])</code></td>
<td>Creates a new table with the columns of self and other, containing rows for all values of a column that appear in both tables.</td>
</tr>
<tr>
<td><code>Table.stats([ops])</code></td>
<td>Compute statistics for each column and place them in a table.</td>
</tr>
<tr>
<td><code>Table.percentile(p)</code></td>
<td>Return a new table with one row containing the pth percentile for each column.</td>
</tr>
<tr>
<td><code>Table.sample([k, with_replacement, weights])</code></td>
<td>Return a new table where k rows are randomly sampled from the original table.</td>
</tr>
<tr>
<td><code>Table.shuffle()</code></td>
<td>Return a new table where all the rows are randomly shuffled from the original table.</td>
</tr>
<tr>
<td><code>Table.sample_from_distribution(distribution, k)</code></td>
<td>Return a new table with the same number of rows and a new column.</td>
</tr>
<tr>
<td><code>Table.split(k)</code></td>
<td>Return a tuple of two tables where the first table contains k rows randomly sampled and the second contains the remaining rows.</td>
</tr>
<tr>
<td><code>Table.bin(*columns, **vargs)</code></td>
<td>Group values by bin and compute counts per bin by column.</td>
</tr>
<tr>
<td><code>Table.pivot_bin(pivot_columns, value_column)</code></td>
<td>Form a table with columns formed by the unique tuples in pivot_columns containing counts per bin of the values associated with each tuple in the value_column.</td>
</tr>
<tr>
<td><code>Table.relabeled(label, new_label)</code></td>
<td>Return a new table with label specifying column label(s) replaced by corresponding new_label.</td>
</tr>
<tr>
<td><code>Table.with_row(row)</code></td>
<td>Return a table with an additional row.</td>
</tr>
<tr>
<td><code>Table.with_rows(rows)</code></td>
<td>Return a table with additional rows.</td>
</tr>
<tr>
<td><code>Table.with_column(label, values[, formatter])</code></td>
<td>Return a new table with an additional or replaced column.</td>
</tr>
<tr>
<td><code>Table.with_columns(*labels_and_values, ...)</code></td>
<td>Return a table with additional or replaced columns.</td>
</tr>
</tbody>
</table>

3.1. Tables (datascience.tables)
### 3.1.31 datascience.tables.Table.copy

**Table.copy(*, shallow=False)**

Return a copy of a table.

**Args:**
- shallow: perform a shallow copy

**Returns:**
- A copy of the table.

By default, copy performs a deep copy of the original table. This means that it constructs a new object and recursively inserts copies into it of the objects found in the original. Note in the following example, table_copy is a deep copy of original_table so when original_table is updated it does not change table_copy as it does not contain reference's to original_table objects due to the deep copy.

```python
>>> value = ['foo']
>>> original_table = Table().with_columns(
...     'A', make_array(1, 2, 3),
...     'B', make_array(value, ['foo', 'bar'], ['foo']),
... )
>>> original_table
A | B
1 | ['foo']
2 | ['foo', 'bar']
3 | ['foo']
>>> table_copy = original_table.copy()
>>> table_copy
A | B
1 | ['foo']
2 | ['foo', 'bar']
3 | ['foo']
>>> value.append('bar')
>>> original_table
A | B
1 | ['foo', 'bar']
2 | ['foo', 'bar']
3 | ['foo']
>>> table_copy
A | B
1 | ['foo']
2 | ['foo', 'bar']
3 | ['foo']
```

By contrast, when a shallow copy is performed, a new object is constructed and references are inserted into it to the objects found in the original. Note in the following example how the update to original_table occurs in both table_shallow_copy and original_table because table_shallow_copy contains references to the original_table.

```python
>>> value = ['foo']
>>> original_table = Table().with_columns(
...     'A', make_array(1, 2, 3),
...     'B', make_array(value, ['foo', 'bar'], ['foo']),
... )
>>> original_table
A | B
1 | ['foo']
2 | ['foo', 'bar']
3 | ['foo']
>>> original_table
A | B
1 | ['foo']
2 | ['foo', 'bar']
3 | ['foo']
```
3.1.32 datascience.tables.Table.select

Table.select(*column_or_columns)

Return a table with only the columns in column_or_columns.

Args:

column_or_columns: Columns to select from the Table as either column labels (str) or column indices (int).

Returns:

A new instance of Table containing only selected columns. The columns of the new Table are in the order given in column_or_columns.

Raises:

KeyError if any of column_or_columns are not in the table.

```python
>>> flowers = Table().with_columns(
...     'Number of petals', make_array(8, 34, 5),
...     'Name', make_array('lotus', 'sunflower', 'rose'),
...     'Weight', make_array(10, 5, 6)
... )

>>> flowers
Number of petals | Name | Weight
8 | lotus | 10
34 | sunflower | 5
5 | rose | 6

>>> flowers.select('Number of petals', 'Weight')
Number of petals | Weight
8 | 10
34 | 5
5 | 6
```
3.1.33  datascience.tables.Table.drop

Table.drop(*column_or_columns)

Return a Table with only columns other than selected label or labels.

Args:

- column_or_columns (string or list of strings): The header names or indices of the columns to be dropped.
  - column_or_columns must be an existing header name, or a valid column index.

Returns:

An instance of Table with given columns removed.
3.1.34 datascience.tables.Table.take

Table.take()

Return a new Table with selected rows taken by index.

**Args:**
- row_indices_or_slice (integer or array of integers): The row index, list of row indices or a slice of row indices to be selected.

**Returns:**
A new instance of Table with selected rows in order corresponding to row_indices_or_slice.

**Raises:**
- IndexError, if any row_indices_or_slice is out of bounds with respect to column length.

```python
>>> grades = Table().with_columns('letter grade', ...
...     make_array('A+', 'A', 'A-', 'B+', 'B', 'B-'), ...
...     'gpa', make_array(4, 4, 3.7, 3.3, 3, 2.7))

>>> grades
letter grade | gpa
A+  | 4
A   | 4
A-  | 3.7
B+  | 3.3
B   | 3
B-  | 2.7

>>> grades.take(0)
letter grade | gpa
A+  | 4

>>> grades.take(-1)
letter grade | gpa
B-  | 2.7

>>> grades.take(make_array(2, 1, 0))
letter grade | gpa
```
A− | 3.7
A  | 4
A+ | 4

>>> grades.take[:3]
letter grade | gpa
A+ | 4
A  | 4
A− | 3.7

>>> grades.take(np.arange(0,3))
letter grade | gpa
A+ | 4
A  | 4
A− | 3.7

>>> grades.take(0, 2)
letter grade | gpa
A+ | 4
A− | 3.7

>>> grades.take(10)
Traceback (most recent call last):
...  IndexError: index 10 is out of bounds for axis 0 with size 6

3.1.35 datascience.tables.Table.exclude

datascience.tables.Table.exclude()

Return a new Table without a sequence of rows excluded by number.

Args:

row_indices_or_slice (integer or list of integers or slice):
The row index, list of row indices or a slice of row indices to be excluded.

Returns:

A new instance of Table.

```python
>>> t = Table().with_columns(
...   'letter grade', make_array('A+', 'A', 'A-', 'B+', 'B', 'B-'),
...   'gpa', make_array(4, 4, 3.7, 3, 3, 2.7))

>>> t
letter grade | gpa
A+ | 4
A  | 4
A− | 3.7
B+ | 3.3
B  | 3
B− | 2.7

>>> t.exclude(4)
letter grade | gpa
A+ | 4
A  | 4
A− | 3.7
B+ | 3.3
```
Note that `exclude` also supports NumPy-like indexing and slicing:

```python
>>> t.exclude[:3]
letter grade | gpa
B+ | 3.3
B  | 3
B- | 2.7
```

```python
>>> t.exclude[1, 3, 4]
letter grade | gpa
A+ | 4
A- | 3.7
B- | 2.7
```

### 3.1.36 `datascience.tables.Table.move_column`

`Table.move_column(label, index)`

Returns a new table with specified column moved to the specified column index.

**Args:**

- `label` (str) A single label of column to be moved.
- `index` (int) A single index of column to move to.

```python
>>> titanic = Table().with_columns('age', make_array(21, 44, 56, 89, 95, ...
   40, 80, 45), 'survival', make_array(0, 0, 0, 1, 1, 0, 1),
```
... 'gender', make_array('M', 'M', 'M', 'F', 'F', 'F'),
... 'prediction', make_array(0, 0, 1, 1, 0, 1, 0, 1))

```python
>>> titanic
age | survival | gender | prediction
21 | 0 | M | 0
44 | 0 | M | 0
56 | 0 | M | 1
89 | 1 | M | 1
95 | 1 | F | 0
40 | 1 | F | 1
80 | 0 | F | 0
45 | 1 | F | 1
```

```python
>>> titanic.move_column('survival', 3)
age | gender | prediction | survival
21 | M | 0 | 0
44 | M | 0 | 0
56 | M | 1 | 0
89 | M | 1 | 1
95 | F | 0 | 1
40 | F | 1 | 1
80 | F | 0 | 0
45 | F | 1 | 1
```

### 3.1.37 datascience.tables.Table.where

Table.where(column_or_label, value_or_predicate=None, other=None)

Return a new Table containing rows where value_or Predicate returns True for values in column_or_label.

**Args:**
- column_or_label: A column of the Table either as a label (str) or an index (int). Can also be an array of booleans; only the rows where the array value is True are kept.
- value_or_predicate: If a function, it is applied to every value in column_or_label. Only the rows where value_or_predicate returns True are kept. If a single value, only the rows where the values in column_or_label are equal to value_or_predicate are kept.
- other: Optional additional column label for value_or_predicate to make pairwise comparisons. See the examples below for usage. When other is supplied, value_or_predicate must be a callable function.

**Returns:**
- If value_or_predicate is a function, returns a new Table containing only the rows where value_or_predicate(val) is True for the val's in column_or_label.
- If value_or_predicate is a value, returns a new Table containing only the rows where the values in column_or_label are equal to value_or_predicate.
- If column_or_label is an array of booleans, returns a new Table containing only the rows where column_or_label is True.

```python
>>> marbles = Table().with_columns(
...   "Color", make_array("Red", "Green", "Blue"),
...```

(continues on next page)
... "Red", "Green", "Green"),
... "Shape", make_array("Round", "Rectangular", "Rectangular",
... "Round", "Rectangular", "Round"),
... "Amount", make_array(4, 6, 12, 7, 9, 2),
... "Price", make_array(1.30, 1.20, 2.00, 1.75, 0, 3.00))

>>> marbles
Color | Shape | Amount | Price
Red | Round | 4 | 1.3
Green | Rectangular | 6 | 1.2
Blue | Rectangular | 12 | 2
Red | Round | 7 | 1.75
Green | Rectangular | 9 | 0
Green | Round | 2 | 3

Use a value to select matching rows

>>> marbles.where("Price", 1.3)
Color | Shape | Amount | Price
Red | Round | 4 | 1.3

In general, a higher order predicate function such as the functions in datascience.predicates.are can be used.

>>> from datascience.predicates import are
>>> # equivalent to previous example
>>> marbles.where("Price", are.equal_to(1.3))
Color | Shape | Amount | Price
Red | Round | 4 | 1.3

>>> marbles.where("Price", are.above(1.5))
Color | Shape | Amount | Price
Blue | Rectangular | 12 | 2
Red | Round | 7 | 1.75
Green | Round | 2 | 3

Use the optional argument other to apply predicates to compare columns.

>>> marbles.where("Price", are.above, "Amount")
Color | Shape | Amount | Price
Green | Round | 2 | 3

>>> marbles.where("Price", are.equal_to, "Amount") # empty table
Color | Shape | Amount | Price

3.1.38 datascience.tables.Table.sort

Table.sort(column_or_label, descending=False, distinct=False)

Return a Table of rows sorted according to the values in a column.

Args:
- column_or_label: the column whose values are used for sorting.
- descending: if True, sorting will be in descending, rather than ascending order.
- distinct: if True, repeated values in column_or_label will be omitted.

Returns:
An instance of Table containing rows sorted based on the values in column_or_label.

```python
>>> marbles = Table().with_columns(
    ... "Color", make_array("Red", "Green", "Blue", "Red", "Green", "Green"),
    ... "Shape", make_array("Round", "Rectangular", "Rectangular", "Round",
    ... "Amount", make_array(4, 6, 12, 7, 9, 2),
    ... "Price", make_array(1.30, 1.30, 2.00, 1.75, 1.40, 1.00))
>>> marbles
<table>
<thead>
<tr>
<th>Color</th>
<th>Shape</th>
<th>Amount</th>
<th>Price</th>
</tr>
</thead>
<tbody>
<tr>
<td>Red</td>
<td>Round</td>
<td>4</td>
<td>1.3</td>
</tr>
<tr>
<td>Green</td>
<td>Rectangular</td>
<td>6</td>
<td>1.3</td>
</tr>
<tr>
<td>Blue</td>
<td>Rectangular</td>
<td>12</td>
<td>2</td>
</tr>
<tr>
<td>Red</td>
<td>Round</td>
<td>7</td>
<td>1.75</td>
</tr>
<tr>
<td>Green</td>
<td>Rectangular</td>
<td>9</td>
<td>1.4</td>
</tr>
<tr>
<td>Green</td>
<td>Round</td>
<td>2</td>
<td>1</td>
</tr>
</tbody>
</table>

>>> marbles.sort("Amount")
<table>
<thead>
<tr>
<th>Color</th>
<th>Shape</th>
<th>Amount</th>
<th>Price</th>
</tr>
</thead>
<tbody>
<tr>
<td>Green</td>
<td>Round</td>
<td>2</td>
<td>1</td>
</tr>
<tr>
<td>Red</td>
<td>Round</td>
<td>4</td>
<td>1.3</td>
</tr>
<tr>
<td>Green</td>
<td>Rectangular</td>
<td>6</td>
<td>1.3</td>
</tr>
<tr>
<td>Red</td>
<td>Round</td>
<td>7</td>
<td>1.75</td>
</tr>
<tr>
<td>Green</td>
<td>Rectangular</td>
<td>9</td>
<td>1.4</td>
</tr>
<tr>
<td>Blue</td>
<td>Rectangular</td>
<td>12</td>
<td>2</td>
</tr>
</tbody>
</table>

>>> marbles.sort("Amount", descending = True)
<table>
<thead>
<tr>
<th>Color</th>
<th>Shape</th>
<th>Amount</th>
<th>Price</th>
</tr>
</thead>
<tbody>
<tr>
<td>Blue</td>
<td>Rectangular</td>
<td>12</td>
<td>2</td>
</tr>
<tr>
<td>Green</td>
<td>Rectangular</td>
<td>9</td>
<td>1.4</td>
</tr>
<tr>
<td>Red</td>
<td>Round</td>
<td>7</td>
<td>1.75</td>
</tr>
<tr>
<td>Green</td>
<td>Rectangular</td>
<td>6</td>
<td>1.3</td>
</tr>
<tr>
<td>Red</td>
<td>Round</td>
<td>4</td>
<td>1.3</td>
</tr>
<tr>
<td>Green</td>
<td>Round</td>
<td>2</td>
<td>1</td>
</tr>
</tbody>
</table>

>>> marbles.sort(3) # the Price column
<table>
<thead>
<tr>
<th>Color</th>
<th>Shape</th>
<th>Amount</th>
<th>Price</th>
</tr>
</thead>
<tbody>
<tr>
<td>Green</td>
<td>Round</td>
<td>2</td>
<td>1</td>
</tr>
<tr>
<td>Red</td>
<td>Round</td>
<td>4</td>
<td>1.3</td>
</tr>
<tr>
<td>Green</td>
<td>Rectangular</td>
<td>6</td>
<td>1.3</td>
</tr>
<tr>
<td>Green</td>
<td>Rectangular</td>
<td>9</td>
<td>1.4</td>
</tr>
<tr>
<td>Red</td>
<td>Round</td>
<td>7</td>
<td>1.75</td>
</tr>
<tr>
<td>Blue</td>
<td>Rectangular</td>
<td>12</td>
<td>2</td>
</tr>
</tbody>
</table>
```
```python
>>> marbles.sort(3, distinct = True)
Color | Shape | Amount | Price
Green | Round | 2      | 1
Red    | Round | 4      | 1.3
Green  | Rectangular | 9  | 1.4
Red    | Round | 7      | 1.75
Blue   | Rectangular | 12 | 2

3.1.39 `datascience.tables.Table.group`

Table\texttt{.group}(*column\_or\_label*, *collect=None*)

Group rows by unique values in a column; count or aggregate others.

**Args:**
- \texttt{column\_or\_label}: values to group (column label or index, or array)
- \texttt{collect}: a function applied to values in other columns for each group

**Returns:**
A Table with each row corresponding to a unique value in \texttt{column\_or\_label}, where the first column contains the unique values from \texttt{column\_or\_label}, and the second contains counts for each of the unique values. If \texttt{collect} is provided, a Table is returned with all original columns, each containing values calculated by first grouping rows according to \texttt{column\_or\_label}, then applying \texttt{collect} to each set of grouped values in the other columns.

**Note:**
The grouped column will appear first in the result table. If \texttt{collect} does not accept arguments with one of the column types, that column will be empty in the resulting table.

```
3.1.40 datascience.tables.Table.groups

Table.groups(labels, collect=None)

Group rows by multiple columns, count or aggregate others.

Args:
  
  labels: list of column names (or indices) to group on

  collect: a function applied to values in other columns for each group

Returns: A Table with each row corresponding to a unique combination of values in
the columns specified in labels, where the first columns are those specified in labels, followed by a
column of counts for each of the unique values. If collect is provided, a Table is returned with all
original columns, each containing values calculated by first grouping rows according to to values in the
labels column, then applying collect to each set of grouped values in the other columns.

Note:
The grouped columns will appear first in the result table. If collect does not accept arguments with one
of the column types, that column will be empty in the resulting table.

```python
>>> marbles = Table().with_columns(
    ...   "Color", make_array("Red", "Green", "Blue", "Red", "Green", "Green"),
    ...   "Shape", make_array("Round", "Rectangular", "Rectangular", "Round",
    ...   →"Rectangular", "Round"),
    ...   "Amount", make_array(4, 6, 12, 7, 9, 2),
    ...   "Price", make_array(1.30, 1.30, 2.00, 1.75, 1.40, 1.00))
>>> marbles
Color  |  Shape  |  Amount  |  Price
Red  |  Round  |  4  |  1.3
Green |  Rectangular  |  6  |  1.3
Blue  |  Rectangular  |  12  |  2
Red  |  Round  |  7  |  1.75
Green |  Rectangular  |  9  |  1.4
Green |  Round  |  2  |  1
>>> marbles.group(["Color", "Shape")
Color  |  Shape  |  count
Blue  |  Rectangular  |  1
Green |  Rectangular  |  2
Green |  Round  |  1
Red  |  Round  |  2
>>> marbles.group(["Color", "Shape"], sum)
Color  |  Shape  |  Amount sum  |  Price sum
Blue  |  Rectangular  |  12  |  2
Green |  Rectangular  |  15  |  2.7
Green |  Round  |  2  |  1
Red  |  Round  |  11  |  3.05
```
3.1.41 datascience.tables.Table.pivot

Table.pivot(columns, rows, values=None, collect=None, zero=None)

Generate a table with a column for each unique value in columns, with rows for each unique value in rows. Each row counts/aggregates the values that match both row and column based on collect.

Args:

columns – a single column label or index, (str or int),
    used to create new columns, based on its unique values.
rows – row labels or indices, (str or int or list),
    used to create new rows based on it’s unique values.
values – column label in table for use in aggregation.
    Default None.
collect – aggregation function, used to group
    values over row-column combinations. Default None.
zero – zero value to use for non-existent row-column
    combinations.

Raises:

TypeError – if collect is passed in and values is not,
    vice versa.

Returns:

New pivot table, with row-column combinations, as specified, with aggregated values by collect across the intersection of columns and rows. Simple counts provided if values and collect are None, as default.

```python
>>> titanic = Table().with_columns('age', make_array(21, 44, 56, 89, 95,
... 40, 80, 45), 'survival', make_array(0, 0, 1, 1, 0, 1, 0, 1),
... 'gender', make_array('M', 'M', 'M', 'M', 'F', 'F', 'F', 'F'),
... 'prediction', make_array(0, 0, 1, 1, 0, 1, 0, 1))

>>> titanic
age | survival | gender | prediction
21 | 0 | M | 0
44 | 0 | M | 0
56 | 0 | M | 1
89 | 1 | M | 1
95 | 1 | F | 0
40 | 1 | F | 1
80 | 0 | F | 0
45 | 1 | F | 1

>>> titanic.pivot('survival', 'gender')
gender | 0 | 1
F | 1 | 3
M | 3 | 1

>>> titanic.pivot('prediction', 'gender')
gender | 0 | 1
F | 2 | 2
M | 2 | 2

>>> titanic.pivot('survival', 'gender', values='age', collect = np.mean)
gender | 0 | 1
F | 80 | 60
```

```python
M | 40.3333 | 89

>>> titanic.pivot('survival', make_array('prediction', 'gender'))
prediction | gender | 0 | 1
0 | F | 1 | 1
0 | M | 2 | 0
1 | F | 0 | 2
1 | M | 1 | 1

>>> titanic.pivot('survival', 'gender', values = 'age')
Traceback (most recent call last):
  ...
TypeError: values requires collect to be specified

>>> titanic.pivot('survival', 'gender', collect = np.mean)
Traceback (most recent call last):
  ...
TypeError: collect requires values to be specified
```

### 3.1.42 `datascience.tables.Table.stack`

`Table.stack(key, labels=None)`

Takes k original columns and returns two columns, with col. 1 of all column names and col. 2 of all associated data.

**Args:**

- **key:** Name of a column from table which is the basis for stacking values from the table.
- **labels:** List of column names which must be included in the stacked representation of the table. If no value is supplied for this argument, then the function considers all columns from the original table.

**Returns:**

A table whose first column consists of stacked values from column passed in `key`. The second column of this returned table consists of the column names passed in `labels`, whereas the final column consists of the data values corresponding to the respective values in the first and second columns of the new table.

**Examples:**

```python
>>> t = Table.from_records([
...     {  
...         'column1':'data1',
...         'column2':86,
...         'column3':'b',
...         'column4':5,
...     },
...     {  
...         'column1':'data2',
...         'column2':51,
...         'column3':'c',
...         'column4':3,
...     },
...     {  
...         'column1':'data3',
...     }
...])
```

(continues on next page)
... 'column2': 32,
... 'column3': 'a',
... 'column4': 6,
... }
... })

>>> t
  column1 | column2 | column3 | column4
data1     | 86      | b       | 5
data2     | 51      | c       | 3
data3     | 32      | a       | 6

>>> t.stack('column2')
  column2 | column | value
  86      | column1| data1
  86      | column3| b
  51      | column4| 5
  51      | column1| data2
  51      | column3| c
  32      | column4| 3
  32      | column1| data3
  32      | column3| a
  32      | column4| 6

>>> t.stack('column2',labels=['column4','column1'])
  column2 | column | value
  86      | column1| data1
  86      | column4| 5
  51      | column1| data2
  51      | column4| 3
  32      | column1| data3
  32      | column4| 6

### 3.1.43 datascience.tables.Table.join

Table.join(*column_label, other, other_label=None*)

Creates a new table with the columns of self and other, containing rows for all values of a column that appear in both tables.

**Args:**

- **column_label**: label of column or array of labels in self that is used to join rows of other.
- **other**: Table object to join with self on matching values of column_label.

**Kwargs:**

- **other_label**: default None, assumes column_label. Otherwise in other used to join rows.
Returns:
New table self joined with other by matching values in column_label and other_label. If the resulting join is empty, returns None.

```python
>>> table = Table().with_columns('a', make_array(9, 3, 3, 1),
...     'b', make_array(1, 2, 2, 10),
...     'c', make_array(3, 4, 5, 6))
>>> table
a  b  c
9  1  3
3  2  4
3  2  5
1  10 6

>>> table2 = Table().with_columns('a', make_array(9, 1, 1, 1),
...     'd', make_array(1, 2, 2, 10),
...     'e', make_array(3, 4, 5, 6))
>>> table2
a  d  e
9  1  3
1  2  4
1  2  5
1  10 6

>>> table.join('a', table2)
a  b  c  d  e
1  10 6  2  4
1  10 6  2  5
1  10 6 10  6
9  1  3  1  3

>>> table.join('a', table2, 'a')  # Equivalent to previous join
a  b  c  d  e
1  10 6  2  4
1  10 6  2  5
1  10 6 10  6
9  1  3  1  3

>>> table.join('a', table2, 'd')  # Repeat column labels relabeled
a  b  c  a_2  e
1  10 6  9  3

>>> table2  # table2 has three rows with a = 1
a  d  e
9  1  3
1  2  4
1  2  5
1  10 6

>>> table  # table has only one row with a = 1
a  b  c
9  1  3
3  2  4
3  2  5
1  10 6

>>> table.join(['a', 'b'], table2, ['a', 'd'])  # joining on multiple columns
a  b  c  e
1  10 6  6
9  1  3  3
```
### 3.1.44 datascience.tables.Table.stats

`Table.stats(ops=(<built-in function min>, <built-in function max>, <function median>, <built-in function sum>))`

Compute statistics for each column and place them in a table.

**Args:**
- `ops` – A tuple of stat functions to use to compute stats.

**Returns:**
- A `Table` with a prepended statistic column with the name of the function’s as the values and the calculated stats values per column.

By default, stats calculates the minimum, maximum, np.median, and sum of each column.

```python
>>> table = Table().with_columns(
    ...    'A', make_array(4, 0, 6, 5),
    ...    'B', make_array(10, 20, 17, 17),
    ...    'C', make_array(18, 13, 2, 9))
>>> table.stats()

<table>
<thead>
<tr>
<th>statistic</th>
<th>A</th>
<th>B</th>
<th>C</th>
</tr>
</thead>
<tbody>
<tr>
<td>min</td>
<td>0</td>
<td>10</td>
<td>2</td>
</tr>
<tr>
<td>max</td>
<td>6</td>
<td>20</td>
<td>18</td>
</tr>
<tr>
<td>median</td>
<td>4.5</td>
<td>17</td>
<td>11</td>
</tr>
<tr>
<td>sum</td>
<td>15</td>
<td>64</td>
<td>42</td>
</tr>
</tbody>
</table>
```

Note, stats are calculated even on non-numeric columns which may lead to unexpected behavior or in more severe cases errors. This is why it may be best to eliminate non-numeric columns from the table before running stats.

```python
>>> table = Table().with_columns(
    ...    'B', make_array(10, 20, 17, 17),
    ...    'C', make_array("foo", "bar", "baz", "baz"))
>>> table.stats()

<table>
<thead>
<tr>
<th>statistic</th>
<th>B</th>
<th>C</th>
</tr>
</thead>
<tbody>
<tr>
<td>min</td>
<td>10</td>
<td>bar</td>
</tr>
<tr>
<td>max</td>
<td>20</td>
<td>foo</td>
</tr>
<tr>
<td>median</td>
<td>17</td>
<td></td>
</tr>
<tr>
<td>sum</td>
<td>64</td>
<td></td>
</tr>
</tbody>
</table>
>>> table.select('B').stats()

<table>
<thead>
<tr>
<th>statistic</th>
<th>B</th>
</tr>
</thead>
<tbody>
<tr>
<td>min</td>
<td>10</td>
</tr>
<tr>
<td>max</td>
<td>20</td>
</tr>
<tr>
<td>median</td>
<td>17</td>
</tr>
<tr>
<td>sum</td>
<td>64</td>
</tr>
</tbody>
</table>
```

`ops` can also be overridden to calculate custom stats.

```python
>>> table = Table().with_columns(
    ...    'A', make_array(4, 0, 6, 5),
    ...    'B', make_array(10, 20, 17, 17),
    ...    'C', make_array(18, 13, 2, 9))
>>> def weighted_average(x):
    ...    return np.average(x, weights=[1, 0, 1.5, 1.25])
>>> table.stats(ops=(weighted_average, np.mean, np.median, np.std))
```

(continues on next page)
### 3.1.45 datascience.tables.Table.percentile

**Table.percentile(p)**

Return a new table with one row containing the pth percentile for each column.

Assumes that each column only contains one type of value.

Returns a new table with one row and the same column labels. The row contains the pth percentile of the original column, where the pth percentile of a column is the smallest value that at least as large as the p% of numbers in the column.

```python
>>> table = Table().with_columns(
...   'count', make_array(9, 3, 3, 1),
...   'points', make_array(1, 2, 2, 10))
>>> table

<table>
<thead>
<tr>
<th>count</th>
<th>points</th>
</tr>
</thead>
<tbody>
<tr>
<td>9</td>
<td>1</td>
</tr>
<tr>
<td>3</td>
<td>2</td>
</tr>
<tr>
<td>3</td>
<td>2</td>
</tr>
<tr>
<td>1</td>
<td>10</td>
</tr>
</tbody>
</table>

>>> table.percentile(80)

<table>
<thead>
<tr>
<th>count</th>
<th>points</th>
</tr>
</thead>
<tbody>
<tr>
<td>9</td>
<td>10</td>
</tr>
</tbody>
</table>
```

### 3.1.46 datascience.tables.Table.sample

**Table.sample(k=None, with_replacement=True, weights=None)**

Return a new table where k rows are randomly sampled from the original table.

**Args:**

- **k** – specifies the number of rows (int) to be sampled from the table. Default is k equal to number of rows in the table.

- **with_replacement** – (bool) By default True;
  Samples k rows with replacement from table, else samples k rows without replacement.

- **weights** – Array specifying probability the ith row of the table is sampled. Defaults to None, which samples each row with equal probability. weights must be a valid probability distribution – i.e. an array the length of the number of rows, summing to 1.

**Raises:**

- **ValueError** – if weights is not length equal to number of rows in the table; or, if weights does not sum to 1.

**Returns:**

A new instance of Table with k rows resampled.
```python
>>> jobs = Table().with_columns(  # jobs
...   'job', make_array('a', 'b', 'c', 'd'),
...   'wage', make_array(10, 20, 15, 8))
```

```
job | wage
a  | 10
b  | 20
c  | 15
d  |  8
```

```
>>> jobs.sample()
job | wage
b  | 20
b  | 20
a  | 10
d  |  8
```

```
>>> jobs.sample(with_replacement=True)
job | wage
d  |  8
b  | 20
c  | 15
a  | 10
```

```
>>> jobs.sample(k = 2)
job | wage
b  | 20
c  | 15
```

```
>>> ws = make_array(0.5, 0.5, 0, 0)
```

```
>>> jobs.sample(k=2, with_replacement=True, weights=ws)
job | wage
a  | 10
a  | 10
```

```
>>> jobs.sample(k=2, weights=make_array(1, 0, 1, 0))
Traceback (most recent call last):
...
ValueError: probabilities do not sum to 1
```

```
>>> jobs.sample(k=2, weights=make_array(1, 0, 0))  # Weights must be length of table.
Traceback (most recent call last):
...
ValueError: 'a' and 'p' must have same size
```

### 3.1.47 datascience.tables.Table.shuffle

The `Table.shuffle()` method returns a new table where all the rows are randomly shuffled from the original table.

**Returns:**

A new instance of `Table` with all k rows shuffled.
3.1.48 datascience.tables.Table.sample_from_distribution

Table.sample_from_distribution(distribution, k, proportions=False)

Return a new table with the same number of rows and a new column. The values in the distribution column are define a multinomial. They are replaced by sample counts/proportions in the output.

```python
>>> sizes = Table([('size', 'count')]).with_rows([
...   ['small', 50],
...   ['medium', 100],
...   ['big', 50],
...])
>>> np.random.seed(99)
>>> sizes.sample_from_distribution('count', 1000)
size | count | count sample
small | 50 | 228
medium | 100 | 508
big | 50 | 264
```

3.1.49 datascience.tables.Table.split

Table.split(k)

Return a tuple of two tables where the first table contains k rows randomly sampled and the second contains the remaining rows.

Args:

k (int): The number of rows randomly sampled into the first table. k must be between 1 and num_rows - 1.

Raises:

ValueError: k is not between 1 and num_rows - 1.

Returns:

A tuple containing two instances of Table.

```python
>>> jobs = Table().with_columns(
...   'job', make_array('a', 'b', 'c', 'd'),
...   'wage', make_array(10, 20, 15, 8))
>>> jobs
job | wage
a | 10
b | 20
c | 15
d | 8
>>> sample, rest = jobs.split(3)
>>> sample
job | wage
c | 15
```
3.1.50  datascience.tables.Table.bin

Table.bin(*columns, **vargs)
Group values by bin and compute counts per bin by column.

By default, bins are chosen to contain all values in all columns. The following named arguments from
numpy.histogram can be applied to specialize bin widths:

If the original table has n columns, the resulting binned table has n+1 columns, where column 0 contains the
lower bound of each bin.

Args:

columns (str or int): Labels or indices of columns to be
binned. If empty, all columns are binned.

bins (int or sequence of scalars): If bins is an int,
it defines the number of equal-width bins in the given range (10, by default). If bins is a sequence, it
defines the bin edges, including the rightmost edge, allowing for non-uniform bin widths.

range ((float, float)): The lower and upper range of
the bins. If not provided, range contains all values in the table. Values outside the range are ignored.

density (bool): If False, the result will contain the number of
samples in each bin. If True, the result is normalized such that the integral over the range is 1. Note that the sum of the histogram values will
not be equal to 1 unless bins of unity width are chosen; it is not a probability mass function.

3.1.51  datascience.tables.Table.pivot_bin

Table.pivot_bin(pivot_columns, value_column, bins=None, **vargs)
Format a table with columns formed by the unique tuples in pivot_columns containing counts per bin of the values
associated with each tuple in the value_column.

By default, bins are chosen to contain all values in the value_column. The following named arguments from
numpy.histogram can be applied to specialize bin widths:

Args:

bins (int or sequence of scalars): If bins is an int,
it defines the number of equal-width bins in the given range (10, by default). If bins is a sequence, it
defines the bin edges, including the rightmost edge, allowing for non-uniform bin widths.

range ((float, float)): The lower and upper range of
the bins. If not provided, range contains all values in the table. Values outside the range are ignored.

normed (bool): If False, the result will contain the number of
samples in each bin. If True, the result is normalized such that the integral over the range is 1.
**Returns:**

New pivot table with unique rows of specified **pivot_columns**, populated with 0s and 1s with respect to values from **value_column** distributed into specified **bins** and **range**.

**Examples:**

```python
t = Table.from_records([...
  ... {...
  ... 'column1':'data1',
  ... 'column2':86,
  ... 'column3':'b',
  ... 'column4':5,
  ... },
  ... {...
  ... 'column1':'data2',
  ... 'column2':51,
  ... 'column3':'c',
  ... 'column4':3,
  ... },
  ... {...
  ... 'column1':'data3',
  ... 'column2':32,
  ... 'column3':'a',
  ... 'column4':6,
  ... }
  ...
])
```

```python
t
column1 | column2 | column3 | column4
data1 | 86 | b | 5
data2 | 51 | c | 3
data3 | 32 | a | 6
```

```python
t.pivot_bin(pivot_columns='column1', value_column='column2')
bin | data1 | data2 | data3
32 | 0 | 0 | 1
37.4 | 0 | 0 | 0
42.8 | 0 | 0 | 0
48.2 | 0 | 1 | 0
53.6 | 0 | 0 | 0
59 | 0 | 0 | 0
64.4 | 0 | 0 | 0
69.8 | 0 | 0 | 0
75.2 | 0 | 0 | 0
80.6 | 1 | 0 | 0
... (1 rows omitted)
```

```python
t.pivot_bin(pivot_columns=['column1', 'column2'], value_column='column4')
bin | data1-86 | data2-51 | data3-32
3 | 0 | 1 | 0
3.3 | 0 | 0 | 0
3.6 | 0 | 0 | 0
3.9 | 0 | 0 | 0
```

(continues on next page)
4.2 | 0 | 0 | 0
4.5 | 0 | 0 | 0
4.8 | 1 | 0 | 0
5.1 | 0 | 0 | 0
5.4 | 0 | 0 | 0
5.7 | 0 | 0 | 1
... (1 rows omitted)

```python
>>> t.pivot_bin(pivot_columns='column1', value_column='column2', bins=[20, 45, 100])
```

<table>
<thead>
<tr>
<th>bin</th>
<th>data1</th>
<th>data2</th>
<th>data3</th>
</tr>
</thead>
<tbody>
<tr>
<td>20</td>
<td>0</td>
<td>0</td>
<td>1</td>
</tr>
<tr>
<td>45</td>
<td>1</td>
<td>1</td>
<td>0</td>
</tr>
<tr>
<td>100</td>
<td>0</td>
<td>0</td>
<td>0</td>
</tr>
</tbody>
</table>

```python
>>> t.pivot_bin(pivot_columns='column1', value_column='column2', bins=5, range=[30, 60])
```

<table>
<thead>
<tr>
<th>bin</th>
<th>data1</th>
<th>data2</th>
<th>data3</th>
</tr>
</thead>
<tbody>
<tr>
<td>30</td>
<td>0</td>
<td>0</td>
<td>1</td>
</tr>
<tr>
<td>36</td>
<td>0</td>
<td>0</td>
<td>0</td>
</tr>
<tr>
<td>42</td>
<td>0</td>
<td>0</td>
<td>0</td>
</tr>
<tr>
<td>48</td>
<td>0</td>
<td>1</td>
<td>0</td>
</tr>
<tr>
<td>54</td>
<td>0</td>
<td>0</td>
<td>0</td>
</tr>
<tr>
<td>60</td>
<td>0</td>
<td>0</td>
<td>0</td>
</tr>
</tbody>
</table>

Exporting / Displaying

<table>
<thead>
<tr>
<th>Method</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td><code>Table.show(max_rows)</code></td>
<td>Display the table.</td>
</tr>
<tr>
<td><code>Table.as_text([max_rows, sep])</code></td>
<td>Format table as text</td>
</tr>
<tr>
<td><code>Table.as_html([max_rows])</code></td>
<td>Format table as HTML</td>
</tr>
<tr>
<td><code>Table.index_by(column_or_label)</code></td>
<td>Return a dict keyed by values in a column that contains lists of</td>
</tr>
<tr>
<td><code>Table.to_array()</code></td>
<td>Convert the table to a structured NumPy array.</td>
</tr>
<tr>
<td><code>Table.to_df()</code></td>
<td>Convert the table to a Pandas DataFrame.</td>
</tr>
<tr>
<td><code>Table.to_csv(filename)</code></td>
<td>Creates a CSV file with the provided filename.</td>
</tr>
</tbody>
</table>

### 3.1.52 datascience.tables.Table.show

`Table.show(max_rows=0)`

Display the table.

**Args:**

- `max_rows`: Maximum number of rows to be output by the function

**Returns:**

A subset of the Table with number of rows specified in `max_rows`. First `max_rows` number of rows are displayed. If no value is passed for `max_rows`, then the entire Table is returned.

**Examples:**

```python
>>> t = Table().with_columns(
...   "column1", make_array("data1", "data2", "data3"),
...   "column2", make_array(86, 51, 32),
... (continues on next page)```
... "column3", make_array("b", "c", "a"),
... "column4", make_array(5, 3, 6)
... )

>>> t
column1 | column2 | column3 | column4
data1  | 86    | b      | 5
data2  | 51    | c      | 3
data3  | 32    | a      | 6

>>> t.show()
<IPython.core.display.HTML object>

>>> t.show(max_rows=2)
<IPython.core.display.HTML object>

### 3.1.53 datascience.tables.Table.as_text

Table.as_text(max_rows=0, sep='|')

Format table as text

**Args:**

- max_rows(int) The maximum number of rows to be present in the converted string of table. (Optional Argument)
- sep(str) The separator which will appear in converted string between the columns. (Optional Argument)

**Returns:**

String form of the table

The table is just converted to a string with columns seperated by the seperator(argument- default(' | ')) and rows seperated by 'n'

Few examples of the as_text() method are as follows:

1.

```python
>>> table = Table().with_columns({'name': ['abc', 'xyz', 'uvw'], 'age': [12, 14, 20], 'height': [5.5, 6.0, 5.9],})
>>> table
name | age | height
abc  | 12  | 5.5
xyz  | 14  | 6
uvw  | 20  | 5.9

>>> table_as_text = table.as_text()
>>> table_as_text
'name | age | height
abc  | 12  | 5.5
xyz  | 14  | 6
uvw  | 20  | 5.9'
>>> type(table)
<class 'datascience.tables.Table'>
```
```python
>>> type(table_astext)
<class 'str'>

2.

```python
definition
>>> sizes = Table(['size', 'count']).with_rows(['small', 50], ['medium', 100], ['big', 50])
```python

```python
>>> sizes
size | count
small | 50
medium | 100
big | 50
```

```python
>>> sizes_astext = sizes.as_text()
>>> sizes_astext
'size | count
small | 50
medium | 100
big | 50'
```

3.

```python
>>> sizes_astext = sizes.as_text(1)
>>> sizes_astext
'size | count
small | 50
... (2 rows omitted)'
```

4.

```python
>>> sizes_astext = sizes.as_text(2, ' - ')
>>> sizes_astext
'size - count
small - 50
... (1 rows omitted)'
```

### 3.1.54 datascience.tables.Table.as_html

**Table.as_html**(max_rows=0)

Format table as HTML

**Args:**

max_rows(int) The maximum number of rows to be present in the converted string of table. (Optional Argument)

**Returns:**

String representing the HTML form of the table

The table is converted to the html format of the table which can be used on a website to represent the table.

Few examples of the as_html() method are as follows. - These examples seem difficult for us to observe and understand since they are in html format, they are useful when you want to display the table on webpages

1. Simple table being converted to HTML

```python
>>> table = Table().with_columns({'name': ['abc', 'xyz', 'uvw'], 'age': [12, 14, 20], 'height': [5.5, 6.0, 5.9]})
```
>>> table
name | age | height
abc  | 12  | 5.5
xyz  | 14  | 6
uvw  | 20  | 5.9

>>> table_as_html = table.as_html()

```html
<table border="1" class="dataframe">
<thead>
<tr>
<th>name</th> <th>age</th> <th>height</th>
</tr>
</thead>
<tbody>
<tr>
<td>abc </td> <td>12 </td> <td>5.5 </td>
</tr>
<tr>
<td>xyz </td> <td>14 </td> <td>6 </td>
</tr>
<tr>
<td>uvw </td> <td>20 </td> <td>5.9 </td>
</tr>
</tbody>
</table>
```

2. Simple table being converted to HTML with max_rows passed in

```python
>>> table
name | age | height
abc  | 12  | 5.5
xyz  | 14  | 6
uvw  | 20  | 5.9

>>> table_as_html_2 = table.as_html(max_rows = 2)

```html
<table border="1" class="dataframe">
<thead>
<tr>
<th>name</th> <th>age</th> <th>height</th>
</tr>
</thead>
<tbody>
<tr>
<td>abc </td> <td>12 </td> <td>5.5 </td>
</tr>
<tr>
<td>xyz </td> <td>14 </td> <td>6 </td>
</tr>
</tbody>
<p>... (1 rows omitted)</p>
```

3.1.55 datascience.tables.Table.index_by

Table.index_by(column_or_label)

Return a dict keyed by values in a column that contains lists of rows corresponding to each value.

Args:
- columns_or_labels: Name or label of a column of the Table, values of which are keys in the returned dict.

Returns:
A dictionary with values from the column specified in the argument columns_or_labels as keys. The corresponding data is a list of Row of values from the rest of the columns of the Table.
Examples:

```python
t = Table().with_columns(
    "column1", make_array("data1", "data2", "data3", "data4"),
    "column2", make_array(86, 51, 32, 91),
    "column3", make_array("b", "c", "a", "a"),
    "column4", make_array(5, 3, 6, 9)
)
```

```plaintext
<table>
<thead>
<tr>
<th>column1</th>
<th>column2</th>
<th>column3</th>
<th>column4</th>
</tr>
</thead>
<tbody>
<tr>
<td>data1</td>
<td>86</td>
<td>b</td>
<td>5</td>
</tr>
<tr>
<td>data2</td>
<td>51</td>
<td>c</td>
<td>3</td>
</tr>
<tr>
<td>data3</td>
<td>32</td>
<td>a</td>
<td>6</td>
</tr>
<tr>
<td>data4</td>
<td>91</td>
<td>a</td>
<td>9</td>
</tr>
</tbody>
</table>
```

```python
t.index_by('column2')

{86: [Row(column1='data1', column2=86, column3='b', column4=5)],
  51: [Row(column1='data2', column2=51, column3='c', column4=3)],
  32: [Row(column1='data3', column2=32, column3='a', column4=6)],
  91: [Row(column1='data4', column2=91, column3='a', column4=9)]}
```

```python
t.index_by('column3')

{'b': [Row(column1='data1', column2=86, column3='b', column4=5)],
 'c': [Row(column1='data2', column2=51, column3='c', column4=3)],
 'a': [Row(column1='data3', column2=32, column3='a', column4=6),
      Row(column1='data4', column2=91, column3='a',
        column4=9)]}
```

### 3.1.56 `datascience.tables.Table.to_array`

**Table.to_array()**

Convert the table to a structured NumPy array.

The resulting array contains a sequence of rows from the table.

**Args:**

None

**Returns:**

arr: a NumPy array

The following is an example of calling to_array() >>> t = Table().with_columns([ ... ‘letter’, [‘a’,’b’,’c’,’z’], ... ‘count’, [9,3,3,1], ... ‘points’, [1,2,2,10], ... ])

```python
t
```

```plaintext
<table>
<thead>
<tr>
<th>letter</th>
<th>count</th>
<th>points</th>
</tr>
</thead>
<tbody>
<tr>
<td>a</td>
<td>9</td>
<td>1</td>
</tr>
<tr>
<td>b</td>
<td>3</td>
<td>2</td>
</tr>
<tr>
<td>c</td>
<td>3</td>
<td>2</td>
</tr>
<tr>
<td>z</td>
<td>1</td>
<td>10</td>
</tr>
</tbody>
</table>
```

```python
example = t.to_array()
```
3.1.57 datascience.tables.Table.to_df

Table.to_df()
Convert the table to a Pandas DataFrame.

Args:
None

Returns:

The Pandas DataFrame of the table
It just converts the table to Pandas DataFrame so that we can use DataFrame instead of the table at some required places.
Here's an example of using the to_df() method:

```python
>>> table = Table().with_columns({'name': ['abc', 'xyz', 'uvw'], ...
... 'age': [12, 14, 20],
... 'height': [5.5, 6.0, 5.9],
... })
```

```python
>>> table
name | age | height
abc  | 12  | 5.5
xyz  | 14  | 6.0
uvw  | 20  | 5.9
```

```python
>>> table_df = table.to_df()
```

```python
>>> table_df
  name  age  height
  0    abc  12  5.5
  1    xyz  14  6.0
  2    uvw  20  5.9
```

```python
>>> type(table)
<class 'datascience.tables.Table'>
```

```python
>>> type(table_df)
<class 'pandas.core.frame.DataFrame'>
```
3.1.58 datascience.tables.Table.to_csv

Table.to_csv(filename)

Creates a CSV file with the provided filename.

The CSV is created in such a way that if we run table.to_csv('my_table.csv') we can recreate the same table with Table.read_table('my_table.csv').

Args:
    filename (str): The filename of the output CSV file.

Returns:
    None, outputs a file with name filename.

```python
>>> jobs = Table().with_columns(
...    'job', make_array('a', 'b', 'c', 'd'),
...    'wage', make_array(10, 20, 15, 8))
>>> jobs
job | wage
a  | 10
b  | 20
c  | 15
d  | 8
```

```python
>>> jobs.to_csv('my_table.csv')
<outputs a file called my_table.csv in the current directory>
```

Visualizations

| Table.plot([column_for_xticks, select, ...]) | Plot line charts for the table. |
| Table.bar([column_for_categories, select, ...]) | Plot bar charts for the table. |
| Table.group_bar(column_label, **vargs) | Plot a bar chart for the table. |
| Table.barh([column_for_categories, select, ...]) | Plot horizontal bar charts for the table. |
| Table.group_barh(column_label, **vargs) | Plot a horizontal bar chart for the table. |
| Table.pivot_hist(pivot_column_label, ...[, ...]) | Draws histograms of each category in a column. |
| Table.hist(*columns[, overlay, bins, ...]) | Plots one histogram for each column in columns. |
| Table.hist_of_counts(*columns[, overlay, ...]) | Plots one count-based histogram for each column in columns. |
| Table.scatter(column_for_x[, select, ...]) | Creates scatterplots, optionally adding a line of best fit. |
| Table.scatter3d(column_for_x, column_for_y) | Convenience wrapper for Table#iscatter3d |
| Table.boxplot(**vargs) | Plots a boxplot for the table. |
| Table.interactive_plots() | Redirects plot, barh, hist, and scatter to their plotly equivalents |
| Table.static_plots() | Turns off redirection of plot, barh, hist, and scatter to their plotly equivalents |
### 3.1.59 `datascience.tables.Table.plot`

`Table.plot(column_for_xticks=None, select=None, overlay=True, width=None, height=None, **vargs)`

Plot line charts for the table. Redirects to `Table#iplot` for plotly charts if interactive plots are enabled with `Table#interactive_plots`.

**Args:**
- `column_for_xticks` (str/array): A column containing x-axis labels

**Kwargs:**
- `overlay` (bool): create a chart with one color per data column; if False, each plot will be displayed separately.
- `show` (bool): whether to show the figure if using interactive plots; if false, the figure is returned instead.
- `vargs`: Additional arguments that get passed into `plt.plot`. See [http://matplotlib.org/api/pyplot_api.html#matplotlib.pyplot.plot](http://matplotlib.org/api/pyplot_api.html#matplotlib.pyplot.plot) for additional arguments that can be passed into `vargs`.

**Raises:**
- `ValueError` – Every selected column must be numerical.

**Returns:**
- Returns a line plot (connected scatter). Each plot is labeled using the values in `column_for_xticks` and one plot is produced for all other columns in `self` (or for the columns designated by `select`).

```python
>>> table = Table().with_columns(
...    'days', make_array(0, 1, 2, 3, 4, 5),
...    'price', make_array(90.5, 90.00, 83.00, 95.50, 82.00, 82.00),
...    'projection', make_array(90.75, 82.00, 82.50, 82.50, 83.00, 82.50))

>>> table
days | price  | projection
0    | 90.5   | 90.75
1    | 90     | 82
2    | 83     | 82.5
3    | 95.5   | 82.5
4    | 82     | 83
5    | 82     | 82.5

>>> table.plot('days')
<line graph with days as x-axis and lines for price and projection>

>>> table.plot('days', overlay=False)
<line graph with days as x-axis and line for price>
<line graph with days as x-axis and line for projection>

>>> table.plot('days', 'price')
<line graph with days as x-axis and line for price>
```
### 3.1.60 datascience.tables.Table.bar

**Table.bar**

```python
Table.bar(column_for_categories=None, select=None, overlay=True, width=None, height=None, **vargs)
```

Plot bar charts for the table.

- Each plot is labeled using the values in `column_for_categories` and one plot is produced for every other column (or for the columns designated by `select`).
- Every selected column except `column_for_categories` must be numerical.

**Args:**
- `column_for_categories` (str): A column containing x-axis categories

**Kwargs:**
- `overlay` (bool): create a chart with one color per data column; if False, each will be displayed separately.
- `vargs`: Additional arguments that get passed into `plt.bar`. See [http://matplotlib.org/api/pyplot_api.html#matplotlib.pyplot.bar](http://matplotlib.org/api/pyplot_api.html#matplotlib.pyplot.bar) for additional arguments that can be passed into `vargs`.

### 3.1.61 datascience.tables.Table.group_bar

**Table.group_bar**

```python
Table.group_bar(column_label, **vargs)
```

Plot a bar chart for the table.

- The values of the specified column are grouped and counted, and one bar is produced for each group.
- Note: This differs from `bar` in that there is no need to specify bar heights; the height of a category’s bar is the number of copies of that category in the given column. This method behaves more like `hist` in that regard, while `bar` behaves more like `plot` or `scatter` (which require the height of each point to be specified).

**Args:**
- `column_label` (str or int): The name or index of a column

**Kwargs:**
- `overlay` (bool): create a chart with one color per data column; if False, each will be displayed separately.
- `width` (float): The width of the plot, in inches
- `height` (float): The height of the plot, in inches
- `vargs`: Additional arguments that get passed into `plt.bar`. See [http://matplotlib.org/api/pyplot_api.html#matplotlib.pyplot.bar](http://matplotlib.org/api/pyplot_api.html#matplotlib.pyplot.bar) for additional arguments that can be passed into `vargs`.

### 3.1.62 datascience.tables.Table.barh

**Table.barh**

```python
Table.barh(column_for_categories=None, select=None, overlay=True, width=None, height=None, **vargs)
```

Plot horizontal bar charts for the table. Redirects to `Table#ibarh` if interactive plots are enabled with `Table#interactive_plots`

**Args:**
- `column_for_categories` (str): A column containing y-axis categories used to create buckets for bar chart.

**Kwargs:**
overlay (bool): create a chart with one color per data column;
if False, each will be displayed separately.

show (bool): whether to show the figure if using interactive plots; if false, the
figure is returned instead

vargs: Additional arguments that get passed into plt.barh.
See http://matplotlib.org/api/pyplot_api.html#matplotlib.pyplot.barh for additional arguments that can
be passed into vargs.

Raises:
ValueError – Every selected except column for column_for_categories
must be numerical.

Returns:
Horizontal bar graph with buckets specified by column_for_categories. Each plot is labeled using the
values in column_for_categories and one plot is produced for every other column (or for the columns
designated by select).

```python
>>> t = Table().with_columns('Furniture', make_array('chairs', 'tables', 'desks'),
    ... 'Count', make_array(6, 1, 2),
    ... 'Price', make_array(10, 20, 30)
    ... )
>>> t
Furniture | Count | Price
chairs    | 6     | 10
tables    | 1     | 20
desks     | 2     | 30
>>> t.barh('Furniture')
<bar graph with furniture as categories and bars for count and price>
>>> t.barh('Furniture', 'Price')
<bar graph with furniture as categories and bars for price>
>>> t.barh('Furniture', make_array(1, 2))
<bar graph with furniture as categories and bars for count and price>
```

### 3.1.63 datascience.tables.Table.group_barh

Table.group_barh(column_label, **vargs)

Plot a horizontal bar chart for the table.

The values of the specified column are grouped and counted, and one bar is produced for each group.

Note: This differs from barh in that there is no need to specify bar heights; the size of a category’s bar is the
number of copies of that category in the given column. This method behaves more like hist in that regard, while
barh behaves more like plot or scatter (which require the second coordinate of each point to be specified in
another column).

Args:
- column_label (str or int): The name or index of a column

Kwvars:
- overlay (bool): create a chart with one color per data column;
  if False, each will be displayed separately.
- width (float): The width of the plot, in inches height (float): The height of the plot, in inches
vargs: Additional arguments that get passed into `plt.bar`
See [http://matplotlib.org/api/pyplot_api.html#matplotlib.pyplot.bar](http://matplotlib.org/api/pyplot_api.html#matplotlib.pyplot.bar) for additional arguments that can be passed into vargs.

### 3.1.64 datascience.tables.Table.pivot_hist

**Table.pivot_hist**(pivot_column_label, value_column_label, overlay=True, width=6, height=4, **vargs)

Draw histograms of each category in a column. (Deprecated)

Recommended: Use `hist(value_column_label, group=pivot_column_label)`, or with `side_by_side=True` if you really want side-by-side bars.

### 3.1.65 datascience.tables.Table.hist

**Table.hist**(columns, overlay=True, bins=None, bin_column=None, unit=None, counts=None, group=None, rug=False, side_by_side=False, left_end=None, right_end=None, width=None, height=None, **vargs)

Plots one histogram for each column in columns. If no column is specified, plot all columns. If interactive plots are enabled via `Table#interactive_plots`, redirects plotting to plotly with `Table#ihist`.

**Kwargs:**

- **overlay** (bool): If True, plots 1 chart with all the histograms overlaid on top of each other (instead of the default behavior of one histogram for each column in the table). Also adds a legend that matches each bar color to its column. Note that if the histograms are not overlaid, they are not forced to the same scale.

- **bins** (list or int): Lower bound for each bin in the histogram or number of bins. If None, bins will be chosen automatically.

- **bin_column** (column name or index): A column of bin lower bounds. All other columns are treated as counts of these bins. If None, each value in each row is assigned a count of 1.

- **counts** (column name or index): Deprecated name for bin_column.

- **unit** (string): A name for the units of the plotted column (e.g. ‘kg’), to be used in the plot.

- **group** (column name or index): A column of categories. The rows are grouped by the values in this column, and a separate histogram is generated for each group. The histograms are overlaid or plotted separately depending on the overlay argument. If None, no such grouping is done.

- **side_by_side** (bool): Whether histogram bins should be plotted side by side (instead of directly overlaid). Makes sense only when plotting multiple histograms, either by passing several columns or by using the group option.

- **left_end** (int or float) and **right_end** (int or float): (Not supported) for overlaid histograms) The left and right edges of the shading of the histogram. If only one of these is None, then that property will be treated as the extreme edge of the histogram. If both are left None, then no shading will occur.

- **density** (boolean): If True, will plot a density distribution of the data. Otherwise plots the counts.
shade_split (string, {“whole”, “new”, “split”}): If left_end or right_end are specified, shade_split determines how a bin is split that the end falls between two bin endpoints. If shade_split = “whole”, the entire bin will be shaded. If shade_split = “new”, then a new bin will be created and data split appropriately. If shade_split = “split”, the data will first be placed into the original bins, and then separated into two bins with equal height.

show (bool): whether to show the figure for interactive plots; if false, the figure is returned instead

vargs: Additional arguments that get passed into :func:plt.hist.
See http://matplotlib.org/api/pyplot_api.html#matplotlib.pyplot.hist for additional arguments that can be passed into vargs. These include: range, normed/density, cumulative, and orientation, to name a few.

```python
>>> t = Table().with_columns(
    ... 'count', make_array(9, 3, 3, 1),
    ... 'points', make_array(1, 2, 2, 10))
>>> t
count | points
9 | 1
3 | 2
3 | 2
1 | 10
>>> t.hist()
<histogram of values in count>
<histogram of values in points>
```

```python
>>> t = Table().with_columns(
    ... 'value', make_array(101, 102, 103),
    ... 'proportion', make_array(0.25, 0.5, 0.25))
>>> t.hist(bin_column='value')
<histogram of values weighted by corresponding proportions>
```

```python
>>> t = Table().with_columns(
    ... 'value', make_array(1, 2, 3, 2, 5),
    ... 'category', make_array('a', 'a', 'a', 'b', 'b'))
>>> t.hist('value', group='category')
<two overlaid histograms of the data [1, 2, 3] and [2, 5]>
```

3.1.66 datascience.tables.Table.hist_of_counts

Table.hist_of_counts(*columns, overlay=True, bins=None, bin_column=None, group=None, side_by_side=False, width=None, height=None, **vargs)

Plots one count-based histogram for each column in columns. The heights of each bar will represent the counts, and all the bins must be of equal size.

If no column is specified, plot all columns.

Kwargs:

overlay (bool): If True, plots 1 chart with all the histograms overlaid on top of each other (instead of the default behavior of one histogram for each column in the table). Also adds a legend that matches each bar color to its column. Note that if the histograms are not overlaid, they are not forced to the same scale.
bins (array or int): Lower bound for each bin in the histogram or number of bins. If None, bins will be chosen automatically.

bin_column (column name or index): A column of bin lower bounds. All other columns are treated as counts of these bins. If None, each value in each row is assigned a count of 1.

group (column name or index): A column of categories. The rows are grouped by the values in this column, and a separate histogram is generated for each group. The histograms are overlaid or plotted separately depending on the overlay argument. If None, no such grouping is done.

side_by_side (bool): Whether histogram bins should be plotted side by side (instead of directly overlaid). Makes sense only when plotting multiple histograms, either by passing several columns or by using the group option.

vargs: Additional arguments that get passed into :func:plt.hist. See http://matplotlib.org/api/pyplot_api.html#matplotlib.pyplot.hist for additional arguments that can be passed into vargs. These include: range, cumulative, and orientation, to name a few.

```python
>>> t = Table().with_columns(
...   'count', make_array(9, 3, 3, 1),
...   'points', make_array(1, 2, 2, 10))
>>> t
   count | points
      9 | 1
      3 | 2
      3 | 2
      1 | 10
>>> t.hist_of_counts()
<histogram of values in count with counts on y-axis>
<histogram of values in points with counts on y-axis>

>>> t = Table().with_columns(
...   'value', make_array(101, 102, 103),
...   'count', make_array(5, 10, 5))
>>> t.hist_of_counts(bin_column='value')
<histogram of values weighted by corresponding counts>

>>> t = Table().with_columns(
...   'value', make_array(1, 2, 3, 2, 5),
...   'category', make_array('a', 'a', 'a', 'b', 'b'))
>>> t.hist('value', group='category')
<two overlaid histograms of the data [1, 2, 3] and [2, 5]>
```

3.1. Tables (datascience.tables)
3.1.67  datascience.tables.Table.scatter

Table.scatter(column_for_x, select=None, overlay=True, fit_line=False, group=None, labels=None, sizes=None, width=None, height=None, s=20, **vargs)

Creates scatterplots, optionally adding a line of best fit. Redirects to Table#iscatter if interactive plots are enabled with Table#interactive_plots

Args:

column_for_x (str): the column to use for the x-axis values
and label of the scatter plots.

Kwargs:

overlay (bool): if true, creates a chart with one color
per data column; if false, each plot will be displayed separately.

fit_line (bool): draw a line of best fit for each set of points.

vargs: additional arguments that get passed into plt.scatter.
see http://matplotlib.org/api/pyplot_api.html#matplotlib.pyplot.scatter for additional arguments that can be passed into vargs. these include: marker and norm, to name a couple.

group: a column of categories to be used for coloring dots per
each category grouping.

labels: a column of text labels to annotate dots.

sizes: a column of values to set the relative areas of dots.

s: size of dots. if sizes is also provided, then dots will be
in the range 0 to 2 * s.

colors: (deprecated) A synonym for group. Retained
temporarily for backwards compatibility. This argument will be removed in future releases.

show (bool): whether to show the figure if using interactive plots; if false,
the figure is returned instead

Raises:

ValueError – Every column, column_for_x or select, must be numerical

Returns:

Scatter plot of values of column_for_x plotted against values for all other columns in self. Each plot uses the values in column_for_x for horizontal positions. One plot is produced for all other columns in self as y (or for the columns designated by select).

```python
global table = Table().with_columns('x', make_array(9, 3, 3, 1),
...  'y', make_array(1, 2, 2, 10),
...  'z', make_array(3, 4, 5, 6))
```

```
>>> table.scatter('x')
<scatterplot of values in y and z on x>
```
>>> table.scatter('x', overlay=False)
<table scatterplot of values in y on x>
<table scatterplot of values in z on x>

>>> table.scatter('x', fit_line=True)
<table scatterplot of values in y and z on x with lines of best fit>

3.1.68 datascience.tables.Table.scatter3d

Table.scatter3d(column_for_x, column_for_y, select=None, overlay=True, fit_line=False, group=None, labels=None, sizes=None, width=None, height=None, s=5, colors=None, **vargs)

Convenience wrapper for Table#iscatter3d

Creates 3D scatterplots by calling Table#iscatter3d with the same arguments. Cannot be used if interactive plots are not enabled (by calling Table#interactive_plots).

Args:

column_for_x (str): The column to use for the x-axis values and label of the scatter plots.

column_for_y (str): The column to use for the y-axis values and label of the scatter plots.

Kwargs:

overlay (bool): If true, creates a chart with one color per data column; if False, each plot will be displayed separately.

group: A column of categories to be used for coloring dots per each category grouping.

labels: A column of text labels to annotate dots.

sizes: A column of values to set the relative areas of dots.

width (int): the width (in pixels) of the plot area

height (int): the height (in pixels) of the plot area

s: Size of dots. If sizes is also provided, then dots will be in the range 0 to 2 * s.

colors: (deprecated) A synonym for group. Retained temporarily for backwards compatibility. This argument will be removed in future releases.

show (bool): whether to show the figure; if false, the figure is returned instead

vargs (dict): additional kwargs passed to

plotly.graph_objects.Figure.update_layout

Raises:

AssertionError – Interactive plots must be enabled by calling Table#interactive_plots first

ValueError – Every column, column_for_x, column_for_x, or select, must be numerical

Returns:
Scatter plot of values of column_for_x and column_for_y plotted against values for all other columns in self.

```python
>>> table = Table().with_columns(
...   'x', make_array(9, 3, 3, 1),
...   'y', make_array(1, 2, 2, 10),
...   'z1', make_array(3, 4, 5, 6),
...   'z2', make_array(0, 2, 1, 0))
```

```python
>>> table
x  y  z1  z2
9  1  3   0
3  2  4   2
3  2  5   1
1  10 6   0
```

```python
>>> table.iscatter3d('x', 'y')
<plotly 3D scatterplot of values in z1 and z2 on x and y>
```

```python
>>> table.iscatter3d('x', 'y', overlay=False)
<plotly 3D scatterplot of values in z1 on x and y>
<plotly 3D scatterplot of values in z2 on x and y>
```

### 3.1.69 datascience.tables.Table.boxplot

`Table.boxplot(**vargs)`

Plots a boxplot for the table.

Every column must be numerical.

**Kwargs:**

- `vargs`: Additional arguments that get passed into `plt.boxplot`. See [here](http://matplotlib.org/api/pyplot_api.html#matplotlib.pyplot.boxplot) for additional arguments that can be passed into `vargs`. These include `vert` and `showmeans`.

**Returns:**

None

**Raises:**

- `ValueError`: The Table contains columns with non-numerical values.

```python
>>> table = Table().with_columns(
...   'test1', make_array(92.5, 88, 72, 71, 99, 100, 95, 83, 94, 93),
...   'test2', make_array(89, 84, 74, 66, 92, 99, 88, 81, 95, 94))
```

```python
>>> table
test1  test2
92.5   89
88     84
72     74
71     66
99     92
100    99
95     88
83     81
94     95
93     94
```

(continues on next page)
```python
>>> table.boxplot()
<boxplot of test1 and boxplot of test2 side-by-side on the same figure>
>>> table2 = Table().with_columns(
...     'numeric_col', make_array(1, 2, 3, 4),
...     'alpha_col', make_array('a', 'b', 'c', 'd'))
>>> table2.boxplot()
Traceback (most recent call last):
  ...  ValueError: The column 'alpha_col' contains non-numerical values. A boxplot cannot be drawn for this table.
```

### 3.1.70 datascience.tables.Table.interactive_plots

**classmethod** `Table.interactive_plots()`

Redirects `plot`, `barh`, `hist`, and `scatter` to their plotly equivalents.

Sets a global variable that redirects Table.plot to Table.iplot, Table.barh to Table.ibarh, etc. This can be turned off by calling Table.static_plots.

```python
>>> table = Table().with_columns(
...     'days', make_array(0, 1, 2, 3, 4, 5),
...     'price', make_array(90.5, 90.00, 83.00, 95.50, 82.00, 82.00),
...     'projection', make_array(90.75, 82.00, 82.50, 82.50, 83.00, 82.50))
>>> table
days | price | projection
0   | 90.5  | 90.75
1   | 90.0  | 82.0
2   | 83.0  | 82.5
3   | 95.5  | 82.5
4   | 82.0  | 83.0
5   | 82.0  | 82.5
>>> table.plot('days')
<matplotlib line graph with days as x-axis and lines for price and projection>
>>> Table.interactive_plots()
>>> table.plot('days')
<plotly interactive line graph with days as x-axis and lines for price and projection>
```

### 3.1.71 datascience.tables.Table.static_plots

**classmethod** `Table.static_plots()`

Turns off redirection of `plot`, `barh`, `hist`, and `scatter` to their plotly equivalents.

Unsets a global variable that redirects Table.plot to Table.iplot, Table.barh to Table.ibarh, etc. This can be turned on by calling Table.interactive_plots.

```python
>>> table = Table().with_columns(
...     'days', make_array(0, 1, 2, 3, 4, 5),
...     'price', make_array(90.5, 90.00, 83.00, 95.50, 82.00, 82.00),
...     'projection', make_array(90.75, 82.00, 82.50, 82.50, 83.00, 82.50))
```
3.2 Maps (datascience.maps)

Draw maps using folium.

```python
class datascience.maps.Circle(lat, lon, popup='', color='blue', area=314.1592653589793, **kwargs)
```

A marker displayed with either Folium’s circle marker or circle methods.

The circle_marker method draws circles that stay the same size regardless of map zoom, whereas the circle
method draws circles that have a fixed radius in meters. To toggle between them, use the radius_in_meters
flag in the draw_on function.

popup – text that pops up when marker is clicked
color – fill color
area – pixel-squared area of the circle

Defaults from Folium:

fill_opacity: float, default 0.6
Circle fill opacity

More options can be passed into kwargs by following the attributes listed in https://leafletjs.com/reference-1.4.0.html#circlemarker or https://leafletjs.com/reference-1.4.0.html#circle.

For example, to draw three circles with circle_marker:

```python
t = Table().with_columns([  
    'lat', [37.8, 38, 37.9],  
    'lon', [-122, -122.1, -121.9],  
    'label', ['one', 'two', 'three'],  
    'color', ['red', 'green', 'blue'],  
    'area', [3000, 4000, 5000],  
])
Circle.map_table(t)
```

To draw three circles with the circle methods, replace the last line with:

```python
Circle.map_table(t, radius_in_meters=True)
```
**draw_on** *(folium_map, radius_in_meters=False)*

Add feature to Folium map object.

**class** datascience.maps.Map(features=(), ids=(), width=960, height=500, **kwargs)**

A map from IDs to features. Keyword args are forwarded to folium.

**color** *(values, ids=(), key_on='feature.id', palette='YlOrBr', **kwargs)*

Color map features by binning values.

values – a sequence of values or a table of keys and values ids – an ID for each value; if none are provided, indices are used key_on – attribute of each feature to match to ids palette – one of the following color brewer palettes:


Defaults from Folium:

**threshold_scale**: list, default None

Data range for D3 threshold scale. Defaults to the following range of quantiles: [0, 0.5, 0.75, 0.85, 0.9], rounded to the nearest order-of-magnitude integer. Ex: 270 rounds to 200, 5600 to 6000.

**fill_opacity**: float, default 0.6

Area fill opacity, range 0-1.

**line_color**: string, default ‘black’

GeoJSON geopath line color.

**line_weight**: int, default 1

GeoJSON geopath line weight.

**line_opacity**: float, default 1

GeoJSON geopath line opacity, range 0-1.

**legend_name**: string, default None

Title for data legend. If not passed, defaults to columns[1].

**copy()**

Copies the current Map into a new one and returns it. Note: This only copies rendering attributes. The underlying map is NOT deep-copied. This is as a result of no functionality in Folium. Ref: https://github.com/python-visualization/folium/issues/1207

**property features**

**format(**kwargs)**

Apply formatting.

**geojson()**

Render features as a FeatureCollection.

**overlay**(feature, color='Blue', opacity=0.6)

Overlays feature on the map. Returns a new Map.

**Args:**

feature: a Table of map features, a list of map features,
a Map, a Region, or a circle marker map table. The features will be overlayed on the Map with specified color.

**color (str)**: Color of feature. Defaults to ‘Blue’
opacity (float): Opacity of overlain feature. Defaults to 0.6.

Returns:
A new Map with the overlain feature.

classmethod read_geojson(path_or_json_or_string_or_url)
Read a geoJSON string, object, file, or URL. Return a dict of features keyed by ID.
class datascience.maps.Marker(lat, lon, popup='', color='blue', **kwargs)
A marker displayed with Folium’s simple_marker method.

popup – text that pops up when marker is clicked
color – The color of the marker. You can use: ['red', 'blue', 'green', 'purple', 'orange', 'darkred', 'lightred', 'beige', 'darkblue', 'darkgreen', 'cadetblue', 'darkpurple', 'white', 'pink', 'lightblue', 'lightgreen', 'gray', 'black', 'lightgray'] to use standard folium icons. If a hex color code is provided, (color must start with ‘#’), a folium.plugin.BeautifyIcon will be used instead.

Defaults from Folium:

marker_icon: string, default ‘info-sign’
    icon from (http://getbootstrap.com/components/) you want on the marker
classified_marker: boolean, default False
    boolean of whether or not you want the marker clustered with other markers

icon_angle: int, default 0
    angle of icon

popup_width: int, default 300
    width of popup

The icon can be further customized by by passing in attributes into kwargs by using the attributes listed in https://python-visualization.github.io/folium/modules.html#folium.map.Icon.

copy()
Return a deep copy
draw_on(folium_map)
Add feature to Folium map object.
format(**kwargs)
Apply formatting.
geojson(feature_id)
GeoJSON representation of the marker as a point.

property lat_lons
Sequence of lat_lons that describe a map feature (for zooming).

classmethod map(latitudes, longitudes, labels=None, colors=None, areas=None, other_attrs=None, clustered_marker=False, **kwargs)
Return markers from columns of coordinates, labels, & colors.
The areas column is not applicable to markers, but sets circle areas.
Arguments: (TODO) document all options

index_map: list of integers, default None (when not applicable)
    list of indices that maps each marker to a corresponding label at the index in cluster_labels (only applicable when multiple marker marker clusters are being used)
cluster_labels: list of strings, default None (when not applicable)
list of labels used for each cluster of markers (only applicable when multiple marker clusters are being used)

colorbar_scale: list of floats, default None (when not applicable)
list of cutoffs used to indicate where the bins are for each color (only applicable when colorscale gradient is being used)

include_color_scale_outliers: boolean, default None (when not applicable)
boolean of whether or not outliers are included in the colorscale gradient for markers (only applicable when colorscale gradient is being used)

radius_in_meters: boolean, default False
boolean of whether or not Circles should have their radii specified in meters, scales with map zoom

clustered_marker: boolean, default False
boolean of whether or not you want the marker clustered with other markers

other_attrs: dictionary of (key) property names to (value) property values, default None
A dictionary that list any other attributes that the class Marker/Circle should have

classmethod map_table(table, clustered_marker=False, include_color_scale_outliers=True, radius_in_meters=False, **kwargs)
Return markers from the columns of a table.
The first two columns of the table must be the latitudes and longitudes (in that order), followed by ‘labels’, ‘colors’, ‘color_scale’, ‘radius_scale’, ‘cluster_by’, ‘area_scale’, and/or ‘areas’ (if applicable) in any order with columns explicitly stating what property they are representing.

Args:
cls: Type of marker being drawn on the map {Marker, Circle}.
table: Table of data to be made into markers. The first two columns of the table must be the latitudes and longitudes (in that order), followed by ‘labels’, ‘colors’, ‘cluster_by’, ‘color_scale’, ‘radius_scale’, ‘area_scale’, and/or ‘areas’ (if applicable) in any order with columns explicitly stating what property they are representing. Additional columns for marker-specific attributes such as ‘marker_icon’ for the Marker class can be included as well.
clustered_marker: Boolean indicating if markers should be clustered with folium.plugins.MarkerCluster.
include_color_scale_outliers: Boolean indicating if outliers should be included in the colorscale gradient or not.
radius_in_meters: Boolean indicating if circle markers should be drawn to map scale or zoom scale.

class datascience.maps.Region(geojson, **kwargs)
A GeoJSON feature displayed with Folium’s geo_json method.
copy()
Return a deep copy
draw_on(folium_map)
Add feature to Folium map object.
format(**kwargs)
Apply formatting.
geojson(feature_id)
Return GeoJSON with ID substituted.
property lat_lons
   A flat list of (lat, lon) pairs.

property polygons
   Return a list of polygons describing the region.
   • Each polygon is a list of linear rings, where the first describes the exterior and the rest describe interior holes.
   • Each linear ring is a list of positions where the last is a repeat of the first.
   • Each position is a (lat, lon) pair.

property properties

property type
   The GEOJSON type of the regions: Polygon or MultiPolygon.

datascience.maps.get_coordinates(table, replace_columns=False, remove_nans=False)
Add latitude and longitude coordinates to table based on other location identifiers. Must be in the United States.
Takes table with columns “zip code” or “city” and/or “county” and “state” in column names and adds the columns “lat” and “lon”. If a county is not found inside the dataset, that row’s latitude and longitude coordinates are replaced with np.nans. The ‘replace_columns’ flag indicates if the “city”, “county”, “state”, and “zip code” columns should be removed afterwards. The ‘remove_nans’ flag indicates if rows with nan latitudes and longitudes should be removed. Robust to capitalization in city and county names. If a row’s location with multiple zip codes is specified, the latitude and longitude pair assigned to the row will correspond to the smallest zip code.

Dataset was acquired on July 2, 2020 from https://docs.gaslamp.media/download-zip-code-latitude-longitude-city-state-county-csv. Found in geocode_datasets/geocode_states.csv. Modified column names and made city/county columns all in lowercase.

Args:
table: A table with counties that need to be mapped to coordinates
replace_columns: A boolean that indicates if “county”, “city”, “state”, and “zip code” columns should be removed
remove_nans: A boolean that indicates if columns with invalid longitudes and latitudes should be removed

Returns:
Table with latitude and longitude coordinates

3.3 Predicates (datascience.predicates)

Predicate functions.

class datascience.predicates.are
Predicate functions. The class is named “are” for calls to where.

For example, given a table, predicates can be used to pick rows as follows.

```python
>>> from datascience import Table
>>> t = Table().with_columns([
...    'Sizes', ['S', 'M', 'L', 'XL'],
...    'Waists', [30, 34, 38, 42],
...])
>>> t.where('Sizes', are.equal_to('L'))
Sizes | Waists
L     | 38
```
>>> t.where('Waists', are.above(38))
Sizes | Waists
XL   | 42

>>> t.where('Waists', are.above_or_equal_to(38))
Sizes | Waists
L    | 38
XL   | 42

>>> t.where('Waists', are.below(38))
Sizes | Waists
S    | 30
M    | 34

>>> t.where('Waists', are.below_or_equal_to(38))
Sizes | Waists
S    | 30
M    | 34
L    | 38

>>> t.where('Waists', are.strictly_between(30, 38))
Sizes | Waists
M    | 34

>>> t.where('Waists', are.between(30, 38))
Sizes | Waists
S    | 30
M    | 34

>>> t.where('Waists', are.between_or_equal_to(30, 38))
Sizes | Waists
S    | 30
M    | 34
L    | 38

>>> t.where('Sizes', are.equal_to('L'))
Sizes | Waists
L    | 38

>>> t.where('Waists', are.not_above(38))
Sizes | Waists
S    | 30
M    | 34
L    | 38

>>> t.where('Waists', are.not_above_or_equal_to(38))
Sizes | Waists
S    | 30
M    | 34

>>> t.where('Waists', are.not_below(38))
Sizes | Waists
L    | 38
XL   | 42

>>> t.where('Waists', are.not_below_or_equal_to(38))
Sizes | Waists
XL   | 42

>>> t.where('Waists', are.not_strictly_between(30, 38))
Sizes | Waists
S    | 30
L    | 38
XL   | 42

(continues on next page)
>>> t.where('Waists', are.not_between(30, 38))
SIZES | Waists
L     | 38
XL    | 42

>>> t.where('Waists', are.not_between_or_equal_to(30, 38))
SIZES | Waists
XL    | 42

>>> t.where('Sizes', are.containing('L'))
SIZES | Waists
L     | 38
XL    | 42

>>> t.where('Sizes', are.not_containing('L'))
SIZES | Waists
S     | 30
M     | 34

>>> t.where('Sizes', are.contained_in('MXL'))
SIZES | Waists
M     | 34
L     | 38
XL    | 42

>>> t.where('Sizes', are.contained_in('L'))
SIZES | Waists
L     | 38

>>> t.where('Sizes', are.not_contained_in('MXL'))
SIZES | Waists
S     | 30

static above(y)
    Greater than y.

static above_or_equal_to(y)
    Greater than or equal to y.

static below(y)
    Less than y.

static below_or_equal_to(y)
    Less than or equal to y.

static between(y, z)
    Greater than or equal to y and less than z.

static between_or_equal_to(y, z)
    Greater than or equal to y and less than or equal to z.

static contained_in(superstring)
    A string that is part of the given superstring.

static containing(substring)
    A string that contains within it the given substring.

static equal_to(y)
    Equal to y.
```
static not_above(y)
    Is not above y
static not_above_or_equal_to(y)
    Is neither above y nor equal to y
static not_below(y)
    Is not below y
static not_below_or_equal_to(y)
    Is neither below y nor equal to y
static not_between(y, z)
    Is equal to y or less than y or greater than z
static not_between_or_equal_to(y, z)
    Is less than y or greater than z
static not_contained_in(superstring)
    A string that is not contained within the superstring
static not_containing(substring)
    A string that does not contain substring
static not_equal_to(y)
    Is not equal to y
static not_strictly_between(y, z)
    Is equal to y or equal to z or less than y or greater than z
static strictly_between(y, z)
    Greater than y and less than z.
```

### 3.4 Formats (**datascience.formats**)

String formatting for table entries.

**class datascience.formats.CurrencyFormatter**(symbol='$', *args, **vargs)
Format currency and convert to float.

- **convert_value(value)**
  Convert value to float. If value is a string, ensure that the first character is the same as symbol i.e. the value is in the currency this formatter is representing.

- **format_value(value)**
  Format currency.

**class datascience.formats.DateFormatter**(format='%Y-%m-%d %H:%M:%S', *args, **vargs)
Format date & time and convert to UNIX timestamp.

- **convert_value(value)**
  Convert 2015-08-03 to a Unix timestamp int.

- **format_value(value)**
  Format timestamp as a string.
class datascience.formats.DistributionFormatter(decimals=2, *args, **vargs)
    Normalize a column and format as percentages.
    convert_column(values)
        Normalize values.

class datascience.formats.Formatter(min_width=None, max_width=None, etc=None)
    String formatter that truncates long values.
    convert_column(values)
        Convert each value using the convert_value method.
    static convert_value(value)
        Identity conversion (override to convert values).
    property converts_values
        Whether this Formatter also converts values.
    etc = ' ...'
    format_column(label, column)
        Return a formatting function that pads & truncates values.
    static format_value(value)
        Pretty-print an arbitrary value.
    max_width = 60
    min_width = 4

class datascience.formats.NumberFormatter(decimals=2, decimal_point=',', separator=','
                                           int_to_float=False, *args, **vargs)
    Format numbers that may have delimiters.
    convert_value(value)
        Convert string 93,000.00 to float 93000.0.
    format_value(value)
        Pretty-print an arbitrary value.

class datascience.formats.PercentFormatter(decimals=2, *args, **vargs)
    Format a number as a percentage.
    format_value(value)
        Format number as percentage.

3.5 Utility Functions (datascience.util)

Utility functions
datascience.util.is_non_string_iterable(value)
    Returns a boolean value representing whether a value is iterable.
**datascience.util.make_array(*elements)**

Returns an array containing all the arguments passed to this function. A simple way to make an array with a few elements.

As with any array, all arguments should have the same type.

**Args:**
- elements (variadic): elements

**Returns:**
- A NumPy array of same length as the provided varadic argument `elements`

```python
>>> make_array(0)
array([0])
>>> make_array(2, 3, 4)
array([2, 3, 4])
>>> make_array("foo", "bar")
dtype='U3')
array(['foo', 'bar'],
>>> make_array()
array([], dtype=float64)
```

**datascience.util.minimize(f, start=None, smooth=False, log=None, array=False, **vargs)**

Minimize a function f of one or more arguments.

**Args:**
- f: A function that takes numbers and returns a number
- start: A starting value or list of starting values
- smooth: Whether to assume that f is smooth and use first-order info
- log: Logging function called on the result of optimization (e.g. print)
- vargs: Other named arguments passed to scipy.optimize.minimize

**Returns:**
- (a) the minimizing argument of a one-argument function
- (b) an array of minimizing arguments of a multi-argument function

**datascience.util.percentile(p, arr=None)**

Returns the pth percentile of the input array (the value that is at least as great as p% of the values in the array).

If arr is not provided, percentile returns itself curried with p

```python
>>> percentile(74.9, [1, 3, 5, 9])
5
>>> percentile(75, [1, 3, 5, 9])
5
>>> percentile(75.1, [1, 3, 5, 9])
9
>>> f = percentile(75)
>>> f([1, 3, 5, 9])
5
```

**datascience.util.plot_cdf_area(rbound=None, lbound=None, mean=0, sd=1)**

Plots a normal curve with specified parameters and area below curve shaded between lbound and rbound.

3.5. Utility Functions (datascience.util)
_args:
rbound (numeric): right boundary of shaded region
lbound (numeric): left boundary of shaded region; by default is negative infinity
mean (numeric): mean/expectation of normal distribution
sd (numeric): standard deviation of normal distribution

datascience.util.plot_normal_cdf(rbound=None, lbound=None, mean=0, sd=1)
Plots a normal curve with specified parameters and area below curve shaded between lbound and rbound.
_args:
rbound (numeric): right boundary of shaded region
lbound (numeric): left boundary of shaded region; by default is negative infinity
mean (numeric): mean/expectation of normal distribution
sd (numeric): standard deviation of normal distribution

datascience.util.proportions_from_distribution(table, label, sample_size, column_name='Random Sample')
Adds a column named column_name containing the proportions of a random draw using the distribution in label.
This method uses np.random.Generator.multinomial to draw sample_size samples from the distribution in table.column(label), then divides by sample_size to create the resulting column of proportions.
_args:
table: An instance of Table.
label: Label of column in table. This column must contain a distribution (the values must sum to 1).
sample_size: The size of the sample to draw from the distribution.
column_name: The name of the new column that contains the sampled proportions. Defaults to 'Random Sample'.

Returns:
A copy of table with a column column_name containing the sampled proportions. The proportions will sum to 1.

Throws:
ValueError: If the label is not in the table, or if table.column(label) does not sum to 1.

datascience.util.sample_proportions(sample_size: int, probabilities)
Return the proportion of random draws for each outcome in a distribution.
This function is similar to np.random.Generator.multinomial, but returns proportions instead of counts.
_args:
sample_size: The size of the sample to draw from the distribution.
probabilities: An array of probabilities that forms a distribution.

Returns:
An array with the same length as probability that sums to 1.
datascience.util.table_apply(table, func, subset=None)

Applies a function to each column and returns a Table.

Args:
  table: The table to apply your function to.
  func: The function to apply to each column.
  subset: A list of columns to apply the function to; if None,
           the function will be applied to all columns in table.

Returns:
  A table with the given function applied. It will either be the shape == shape(table), or shape (1, table.shape[1])
d

datascience.formats, 125
datascience.maps, 118
datascience.predicates, 122
datascience.util, 126
<table>
<thead>
<tr>
<th>F</th>
<th>L</th>
</tr>
</thead>
<tbody>
<tr>
<td>features ([data.science.maps.Map property], 119)</td>
<td>labels ([data.science.tables.Table property], 67)</td>
</tr>
<tr>
<td>first() ([data.science.tables.Table method], 67)</td>
<td>last() ([data.science.tables.Table method], 68)</td>
</tr>
<tr>
<td>format() ([data.science.maps.Map method], 119)</td>
<td>lat_lons ([data.science.maps.Marker property], 120)</td>
</tr>
<tr>
<td>format() ([data.science.maps.Marker method], 120)</td>
<td>lat_lons ([data.science.maps.Region property], 121)</td>
</tr>
<tr>
<td>format_column() ([data.science.formats.Formatter method], 126)</td>
<td></td>
</tr>
<tr>
<td>format_value() ([data.science.formats.CurrencyFormatter method], 125)</td>
<td></td>
</tr>
<tr>
<td>format_value() ([data.science.formats.DateFormatter method], 125)</td>
<td></td>
</tr>
<tr>
<td>format_value() ([data.science.formats.Formatter static method], 126)</td>
<td></td>
</tr>
<tr>
<td>format_value() ([data.science.formats.NumberFormatter method], 126)</td>
<td></td>
</tr>
<tr>
<td>format_value() ([data.science.formats.PercentFormatter method], 126)</td>
<td></td>
</tr>
<tr>
<td>Formatter ([class in data.science.formats], 126)</td>
<td></td>
</tr>
<tr>
<td>from_array() ([data.science.tables.Table class method], 60)</td>
<td></td>
</tr>
<tr>
<td>from_columns_dict() ([data.science.tables.Table class method], 58)</td>
<td></td>
</tr>
<tr>
<td>from_df() ([data.science.tables.Table class method], 59)</td>
<td></td>
</tr>
<tr>
<td>from_records() ([data.science.tables.Table class method], 58)</td>
<td></td>
</tr>
<tr>
<td>G</td>
<td>M</td>
</tr>
<tr>
<td>geojson() ([data.science.maps.Map method], 119)</td>
<td>make_array() (in module data.science.util), 126</td>
</tr>
<tr>
<td>geojson() ([data.science.maps.Marker method], 120)</td>
<td>Map ([class in data.science.maps], 119)</td>
</tr>
<tr>
<td>geojson() ([data.science.maps.Region method], 121)</td>
<td>map() ([data.science.maps.Marker class method], 120)</td>
</tr>
<tr>
<td>get_coordinates() ([in module data.science.util], 126)</td>
<td>max_width ([data.science.formats.Formatter attribute], 126)</td>
</tr>
<tr>
<td>group() ([data.science.tables.Table method], 89)</td>
<td>min_width ([data.science.formats.Formatter attribute], 126)</td>
</tr>
<tr>
<td>group_bar() ([data.science.tables.Table method], 109)</td>
<td>minimize() ([in module data.science.util], 127)</td>
</tr>
<tr>
<td>group_barh() ([data.science.tables.Table method], 110)</td>
<td>module</td>
</tr>
<tr>
<td>groups() ([data.science.tables.Table method], 90)</td>
<td>data.science.formats, 125</td>
</tr>
<tr>
<td>H</td>
<td></td>
</tr>
<tr>
<td>hist() ([data.science.tables.Table method], 111)</td>
<td>data.science.maps, 118</td>
</tr>
<tr>
<td>hist_of_counts() ([data.science.tables.Table method], 112)</td>
<td>data.science.predicates, 122</td>
</tr>
<tr>
<td>I</td>
<td></td>
</tr>
<tr>
<td>index_by() ([data.science.tables.Table method], 104)</td>
<td></td>
</tr>
<tr>
<td>interactive_plots() ([data.science.tables.Table class method], 117)</td>
<td></td>
</tr>
<tr>
<td>is_non_string_iterable() ([in module data.science.util], 126)</td>
<td></td>
</tr>
<tr>
<td>J</td>
<td>N</td>
</tr>
<tr>
<td>join() ([data.science.tables.Table method], 93)</td>
<td>not_above() ([data.science.predicates.arestaticmethod], 124)</td>
</tr>
<tr>
<td></td>
<td>not_above_or_equal_to() ([data.science.predicates.arestaticmethod], 125)</td>
</tr>
<tr>
<td></td>
<td>not_below() ([data.science.predicates.arestaticmethod], 125)</td>
</tr>
<tr>
<td></td>
<td>not_below_or_equal_to() ([data.science.predicates.arestaticmethod], 125)</td>
</tr>
<tr>
<td></td>
<td>not_between() ([data.science.predicates.arestaticmethod], 125)</td>
</tr>
<tr>
<td></td>
<td>not_between_or_equal_to() ([data.science.predicates.arestaticmethod], 125)</td>
</tr>
<tr>
<td></td>
<td>not_contained_in() ([data.science.predicates.arestaticmethod], 125)</td>
</tr>
<tr>
<td></td>
<td>not_containing() ([data.science.predicates.arestaticmethod], 125)</td>
</tr>
<tr>
<td></td>
<td>not_equal_to() ([data.science.predicates.arestaticmethod], 125)</td>
</tr>
<tr>
<td></td>
<td>not_strictly_between() ([data.science.predicates.arestaticmethod], 125)</td>
</tr>
<tr>
<td></td>
<td>num_columns ([data.science.tables.Table property], 65)</td>
</tr>
<tr>
<td></td>
<td>num_rows ([data.science.tables.Table property], 66)</td>
</tr>
<tr>
<td></td>
<td>NumberFormatter ([class in data.science.formats], 126)</td>
</tr>
</tbody>
</table>
O

overlay() (datascience.maps.Map method), 119

P

PercentFormatter (class in datascience.formats), 126
percentile() (datascience.tables.Table method), 96
percentile() (in module datascience.util), 127
pivot() (datascience.tables.Table method), 91
pivot_bin() (datascience.tables.Table method), 99
pivot_hist() (datascience.tables.Table method), 111
plot() (datascience.tables.Table method), 108
plot_cdf_area() (in module datascience.util), 127
plot_normal_cdf() (in module datascience.util), 128
polygons (datascience.maps.Region property), 122
properties (datascience.maps.Region property), 122
proportions_from_distribution() (in module datascience.util), 128

R

read_geojson() (datascience.maps.Map class method), 120
read_table() (datascience.tables.Table class method), 59
Region (class in datascience.maps), 121
relabel() (datascience.tables.Table method), 77
relabeled() (datascience.tables.Table method), 64
remove() (datascience.tables.Table method), 78
row() (datascience.tables.Table method), 67
rows (datascience.tables.Table property), 66

S

sample() (datascience.tables.Table method), 96
sample_from_distribution() (datascience.tables.Table method), 98
sample_proportions() (in module datascience.util), 128
scatter() (datascience.tables.Table method), 114
scatter3d() (datascience.tables.Table method), 115
select() (datascience.tables.Table method), 81
set_format() (datascience.tables.Table method), 70
show() (datascience.tables.Table method), 101
shuffle() (datascience.tables.Table method), 97
sort() (datascience.tables.Table method), 88
split() (datascience.tables.Table method), 98
stack() (datascience.tables.Table method), 92
static_plots() (datascience.tables.Table class method), 117
stats() (datascience.tables.Table method), 95
strictly_between() (datascience.predicates.are static method), 125

take() (datascience.tables.Table method), 83
to_array() (datascience.tables.Table method), 105
to_csv() (datascience.tables.Table method), 107
to_df() (datascience.tables.Table method), 106
type (datascience.maps.Region property), 122

V

values (datascience.tables.Table property), 68

W

where() (datascience.tables.Table method), 86
with_column() (datascience.tables.Table method), 61
with_columns() (datascience.tables.Table method), 62
with_row() (datascience.tables.Table method), 63
with_rows() (datascience.tables.Table method), 63

T

table_apply() (in module datascience.util), 128