datascience Documentation

Release 0.6.0

John DeNero, David Culler, Alvin Wan, and Sam Lau

July 01, 2016

Contents

Release 0.6.0

Date July 01, 2016

The `datascience` package was written for use in Berkeley's DS 8 course and contains useful functionality for investigating and graphically displaying data.

Start Here: `datascience` Tutorial

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

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
matplotlib.use('Agg')
from datascience import Table
%matplotlib inline
import matplotlib.pyplot as plt
```

```
import numpy as np
plt.style.use('fivethirtyeight')
```

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.

```
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:

```
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()*:

```
In [5]: Table.read_table('sample.csv')
Out[5]:
x    | y    | z
1    | 10   | 100
2    | 11   | 101
3    | 12   | 102
```

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

```
In [6]: Table.read_table('http://data8.org/textbook/notebooks/sat2014.csv')
Out[6]:
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)
count | letter | points
9     | a      | 1
3     | b      | 2
3     | c      | 2
1     | z      | 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]:
count | letter | points
9     | a      | 1
3     | b      | 2
3     | c      | 2
1     | z      | 10

In [10]: t.column('letter')
\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\
array(['a', 'b', 'c', 'z'],
      dtype='<U1')

In [11]: t.column(1)
\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\
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(['a', 'b', 'c', 'z'],
      dtype='<U1')
```

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(count | letter | points
9     | a      | 1
3     | b      | 2
3     | c      | 2
1     | z      | 10)

In [15]: t.rows[0]
\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\

In [16]: t.row(0)
\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\

In [17]: second = t.rows[1]

In [18]: second
Out[18]: Row(count=3, letter='b', points=2)

In [19]: second[0]
\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\Out[19]: 3

In [20]: second[1]
\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\Out[20]: 'b'
```

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 `with_column()`:

```
In [22]: t
Out[22]:
count | letter | points
9     | a      | 1
3     | b      | 2
3     | c      | 2
1     | z      | 10

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

In [24]: t # .with_column returns a new table without modifying the original
\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\
```

```
count | letter | points
9     | a      | 1
3     | b      | 2
3     | c      | 2
1     | z      | 10

In [25]: t.with_column('2 * count', t['count'] * 2) # A simple way to operate on columns
\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\
count | letter | points | 2 * count
9     | a      | 1      | 18
3     | b      | 2      | 6
3     | c      | 2      | 6
1     | z      | 10     | 2
```

Selecting columns with *select()*:

```
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
c      | 2
z      | 10
```

Renaming columns with *relabeled()*:

```
In [28]: t
Out[28]:
count | letter | points
9     | a      | 1
3     | b      | 2
3     | c      | 2
1     | z      | 10

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

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

In [31]: t.relabeled(['letter', 'count', 'points'], ['x', 'y', 'z'])
```

```
\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\
y    | x     | z
9    | a     | 1
3    | b     | 2
3    | c     | 2
1    | z     | 10
```

Selecting out rows by index with *take()* and conditionally with *where()*:

```
In [32]: t
Out[32]:
count | letter | points
9     | a      | 1
3     | b      | 2
3     | c      | 2
1     | z      | 10

In [33]: t.take(2) # the third row
\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\
count | letter | points
3     | c      | 2

In [34]: t.take[0:2] # the first and second rows
\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\
count | letter | points
9     | a      | 1
3     | b      | 2
```

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

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

In [37]: t['count'] < 8 # .where actually takes in an array of booleans
\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\

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

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

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

```
1      | z      | 10

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

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

```
# 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)
\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\
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
1.39757  | 7.59503
1.2359   | 3.84659
0.533464 | 5.41543
-2.98546 | 0.463486
1.22306  | 2.1895
-0.82011 | 1.60256
3.19549  | 5.37406
1.24943  | 5.57266
1.81792  | 1.55282
5.40878  | 5.53846
... (90 rows omitted)
```

Draw histograms with `hist()`:

```
In [48]: normal_data.hist()
[array([  7.59503201,   3.84658856,   5.41543401,   0.46348589,
         2.18949788,   1.60256299,   5.3740626 ,   5.57266286,
         1.55281978,   5.53846123,   1.37223108,   4.49033989,
         6.7476888 ,   1.06318089,   8.2509157 ,   7.08825877,
         4.44024461,   0.43123591,   5.19505952,   4.25175755,
         2.10180902,   2.62730855,  -0.97583087,   5.03869248,
         5.16029445,   0.2626408 ,   4.78624151,   3.644004  ,
        -0.06924523,   5.25701937,   2.15445846,  -2.42660685,
         6.89927222,   4.7568616 ,   1.45341363,   2.08479199,
         4.15957331,   3.52858234,   4.77222311,   4.99783835,
        10.78832185,   5.54254342,   5.87893827,   4.75348006,
         7.02795783,   7.05544858,   5.51512336,   0.23611106,
         2.44966325,   7.3345859 ,   7.42623743,   2.75453035,
         3.90293962,  -1.58905586,   7.22956635,   4.58301934,
         7.99574987,   5.72156548,   6.60220327,   6.71129968,
        10.44854963,   6.94753016,   3.95011565,   7.91739098,
        -3.79383561,   2.75927614,   0.60496927,   7.82426615,
         3.01817281,   6.53223143,   4.88197089,  -1.85114153,
         2.78590191,   7.00570408,   2.5409338 ,  12.11848503,
         7.72720877,   4.92562292,   3.72154753,  -2.5089611 ,
         5.28792474,   0.08378674,   2.86487762,   4.28711374,
         1.95786142,   1.95627955,   5.68622143,   2.95045651,
         5.55623487,   4.25304166,   6.64447626,   0.79888228,
         5.73070838,   6.12305509,   0.90249808,   6.03188514,
         4.25189511,   5.66569765,  -0.0311821 ,   4.68138798]), array([ 1.39756527,  1.2358974 ,  0.
        -0.82011006,  3.19548765,  1.24943434,  1.81792415,  5.40878036,
         6.31062484,  2.48345106,  1.00341303,  0.65012056,  1.03322608,
        -2.78207622,  2.10162821, -0.31957088, -2.11594303,  0.5413891 ,
         4.00933402,  1.68050977, -2.37409128,  1.61720495,  2.9580784 ,
         1.01117906,  2.07417038,  1.33476815,  2.40681026,  0.26641137,
         2.07931717, -0.19413454, -0.87434644, -1.50186649,  0.67287369,
         2.67882269,  3.81920793,  1.23817324,  1.77876134, -1.49645794,
        -0.41313526,  1.1968799 , -1.45108038,  0.53017803,  4.72747609,
         2.45330265,  2.16156973,  0.72001738,  2.58236328, -0.87397981,
         1.67503755,  3.87138119,  0.41288538,  3.45116176,  0.29197318,
        -2.44357775,  2.86812668,  0.65109746,  2.61716999, -0.77406891,
         0.12685672,  2.16945914,  4.98544105,  1.02271237,  2.57472569,
         4.33015185,  1.94987058,  3.05993595,  2.46930365,  4.27071542,
         1.16594877,  0.08620929, -2.10857218,  1.11538131,  1.44116439,
```

```
       -1.4933336 ,   1.2570993 ,   0.38612722,  1.95879249,   4.42217918,
         1.39726257,  0.48101735, -1.26091139, -3.17716373,   3.18013124,
         1.8074025 ,   0.2878458 ,   3.21118692,  3.16719713,   0.7036239 ,
        -0.07132284,  1.72723558,   2.84404404, -2.0728383 ,  -0.33569562,
         3.99169634, -1.03458546,   2.62548374,  4.09651375,   1.05543234])]  [(1.0, 0.784, 0.0, 0.7), ((
```

_build/latex/_images/hist.png

```
In [49]: normal_data.hist(bins = range(-5, 10))
[array([  7.59503201,   3.84658856,   5.41543401,   0.46348589,
          2.18949788,   1.60256299,   5.3740626 ,   5.57266286,
          1.55281978,   5.53846123,   1.37223108,   4.49033989,
          6.7476888 ,   1.06318089,   8.2509157 ,   7.08825877,
          4.44024461,   0.43123591,   5.19505952,   4.25175755,
          2.10180902,   2.62730855,  -0.97583087,   5.03869248,
          5.16029445,   0.2626408 ,   4.78624151,   3.644004  ,
         -0.06924523,   5.25701937,   2.15445846,  -2.42660685,
          6.89927222,   4.7568616 ,   1.45341363,   2.08479199,
          4.15957331,   3.52858234,   4.77222311,   4.99783835,
         10.78832185,   5.54254342,   5.87893827,   4.75348006,
          7.02795783,   7.05544858,   5.51512336,   0.23611106,
          2.44966325,   7.3345859 ,   7.42623743,   2.75453035,
          3.90293962,  -1.58905586,   7.22956635,   4.58301934,
          7.99574987,   5.72156548,   6.60220327,   6.71129968,
         10.44854963,   6.94753016,   3.95011565,   7.91739098,
         -3.79383561,   2.75927614,   0.60496927,   7.82426615,
          3.01817281,   6.53223143,   4.88197089,  -1.85114153,
          2.78590191,   7.00570408,   2.5409338 ,  12.11848503,
          7.72720877,   4.92562292,   3.72154753,  -2.5089611 ,
          5.28792474,   0.08378674,   2.86487762,   4.28711374,
          1.95786142,   1.95627955,   5.68622143,   2.95045651,
          5.55623487,   4.25304166,   6.64447626,   0.79888228,
          5.73070838,   6.12305509,   0.90249808,   6.03188514,
```

```
        4.25189511,   5.66569765,  -0.0311821 ,   4.68138798]), array([ 1.39756527,  1.2358974 ,   0.
     -0.82011006,   3.19548765,   1.24943434,   1.81792415,   5.40878036,
      6.31062484,   2.48345106,   1.00341303,   0.65012056,   1.03322608,
     -2.78207622,   2.10162821,  -0.31957088,  -2.11594303,   0.5413891 ,
      4.00933402,   1.68050977,  -2.37409128,   1.61720495,   2.9580784 ,
      1.01117906,   2.07417038,   1.33476815,   2.40681026,   0.26641137,
      2.07931717,  -0.19413454,  -0.87434644,  -1.50186649,   0.67287369,
      2.67882269,   3.81920793,   1.23817324,   1.77876134,  -1.49645794,
     -0.41313526,   1.1968799 ,  -1.45108038,   0.53017803,   4.72747609,
      2.45330265,   2.16156973,   0.72001738,   2.58236328,  -0.87397981,
      1.67503755,   3.87138119,   0.41288538,   3.45116176,   0.29197318,
     -2.44357775,   2.86812668,   0.65109746,   2.61716999,  -0.77406891,
      0.12685672,   2.16945914,   4.98544105,   1.02271237,   2.57472569,
      4.33015185,   1.94987058,   3.05993595,   2.46930365,   4.27071542,
      1.16594877,   0.08620929,  -2.10857218,   1.11538131,   1.44116439,
     -1.4933336 ,   1.2570993 ,   0.38612722,   1.95879249,   4.42217918,
      1.39726257,   0.48101735,  -1.26091139,  -3.17716373,   3.18013124,
      1.8074025 ,   0.2878458 ,   3.21118692,   3.16719713,   0.7036239 ,
     -0.07132284,   1.72723558,   2.84404404,  -2.0728383 ,  -0.33569562,
      3.99169634,  -1.03458546,   2.62548374,   4.09651375,   1.05543234])] [(1.0, 0.784, 0.0, 0.7), ((
```


_build/latex/_images/hist_binned.png

```
In [50]: normal_data.hist(bins = range(-5, 10), overlay = True)
[array([  7.59503201,   3.84658856,   5.41543401,   0.46348589,
          2.18949788,   1.60256299,   5.3740626 ,   5.57266286,
          1.55281978,   5.53846123,   1.37223108,   4.49033989,
          6.7476888 ,   1.06318089,   8.2509157 ,   7.08825877,
          4.44024461,   0.43123591,   5.19505952,   4.25175755,
          2.10180902,   2.62730855,  -0.97583087,   5.03869248,
          5.16029445,   0.2626408 ,   4.78624151,   3.644004  ,
         -0.06924523,   5.25701937,   2.15445846,  -2.42660685,
          6.89927222,   4.7568616 ,   1.45341363,   2.08479199,
```

```
       4.15957331,   3.52858234,   4.77222311,   4.99783835,
      10.78832185,   5.54254342,   5.87893827,   4.75348006,
       7.02795783,   7.05544858,   5.51512336,   0.23611106,
       2.44966325,   7.3345859 ,   7.42623743,   2.75453035,
       3.90293962,  -1.58905586,   7.22956635,   4.58301934,
       7.99574987,   5.72156548,   6.60220327,   6.71129968,
      10.44854963,   6.94753016,   3.95011565,   7.91739098,
      -3.79383561,   2.75927614,   0.60496927,   7.82426615,
       3.01817281,   6.53223143,   4.88197089,  -1.85114153,
       2.78590191,   7.00570408,   2.5409338 ,  12.11848503,
       7.72720877,   4.92562292,   3.72154753,  -2.5089611 ,
       5.28792474,   0.08378674,   2.86487762,   4.28711374,
       1.95786142,   1.95627955,   5.68622143,   2.95045651,
       5.55623487,   4.25304166,   6.64447626,   0.79888228,
       5.73070838,   6.12305509,   0.90249808,   6.03188514,
       4.25189511,   5.66569765,  -0.0311821 ,   4.68138798]), array([ 1.39756527,  1.2358974 ,  0.
      -0.82011006,  3.19548765,  1.24943434,  1.81792415,  5.40878036,
       6.31062484,  2.48345106,  1.00341303,  0.65012056,  1.03322608,
      -2.78207622,  2.10162821, -0.31957088, -2.11594303,  0.5413891 ,
       4.00933402,  1.68050977, -2.37409128,  1.61720495,  2.9580784 ,
       1.01117906,  2.07417038,  1.33476815,  2.40681026,  0.26641137,
       2.07931717, -0.19413454, -0.87434644, -1.50186649,  0.67287369,
       2.67882269,  3.81920793,  1.23817324,  1.77876134, -1.49645794,
      -0.41313526,  1.1968799 , -1.45108038,  0.53017803,  4.72747609,
       2.45330265,  2.16156973,  0.72001738,  2.58236328, -0.87397981,
       1.67503755,  3.87138119,  0.41288538,  3.45116176,  0.29197318,
      -2.44357775,  2.86812668,  0.65109746,  2.61716999, -0.77406891,
       0.12685672,  2.16945914,  4.98544105,  1.02271237,  2.57472569,
       4.33015185,  1.94987058,  3.05993595,  2.46930365,  4.27071542,
       1.16594877,  0.08620929, -2.10857218,  1.11538131,  1.44116439,
      -1.4933336 ,  1.2570993 ,  0.38612722,  1.95879249,  4.42217918,
       1.39726257,  0.48101735, -1.26091139, -3.17716373,  3.18013124,
       1.8074025 ,  0.2878458 ,  3.21118692,  3.16719713,  0.7036239 ,
      -0.07132284,  1.72723558,  2.84404404, -2.0728383 , -0.33569562,
       3.99169634, -1.03458546,  2.62548374,  4.09651375,  1.05543234]])] [(1.0, 0.784, 0.0, 0.7), ((
```

```
_build/latex/_images/hist_overlay.png
```

If we treat the `normal_data` table as a set of x-y points, we can *plot()* and *scatter()*:

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

```
_build/latex/_images/plot.png
```

```
In [52]: normal_data.scatter('data1')
```

_build/latex/_images/scatter.png
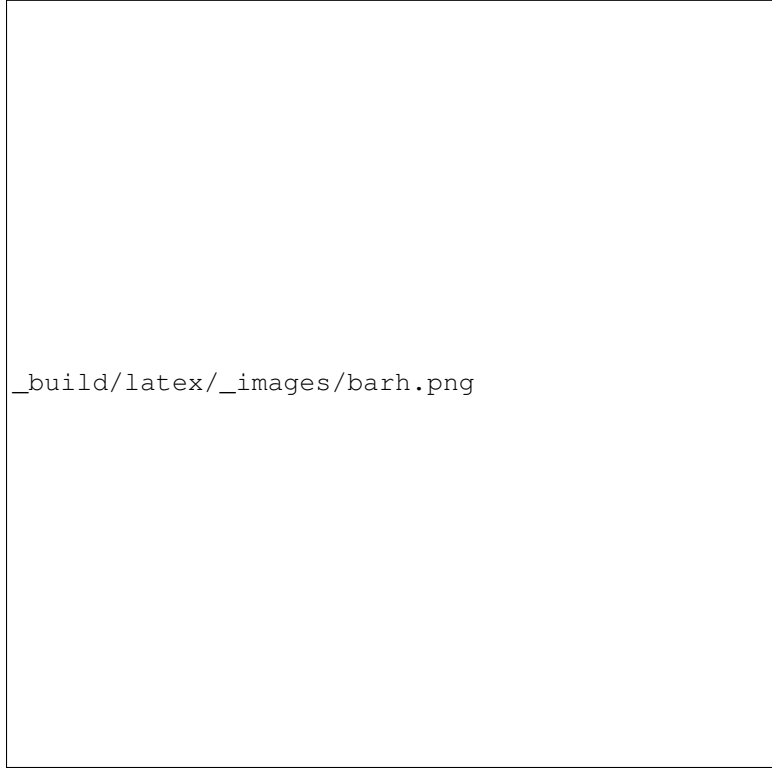
```
In [53]: normal_data.scatter('data1', fit_line = True)
```

_build/latex/_images/scatter_line.png

Use `barh()` to display categorical data.

```
In [54]: t
Out[54]:
count | letter | points
9     | a      | 1
3     | b      | 2
3     | c      | 2
1     | z      | 10

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


_build/latex/_images/barh.png

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
1.39757  | 7.59503
1.2359   | 3.84659
0.533464 | 5.41543
-2.98546 | 0.463486
1.22306  | 2.1895
-0.82011 | 1.60256
3.19549  | 5.37406
1.24943  | 5.57266
1.81792  | 1.55282
5.40878  | 5.53846
... (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 3 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 birthwt contains the birth weight of the baby, in ounces; gest_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('http://data8.org/textbook/notebooks/baby.csv')

In [59]: baby # Let's take a peek at the table
Out[59]:
Birth Weight | Gestational Days | Maternal Age | Maternal Height | Maternal Pregnancy Weight | Matern
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(['m_smoker', 'birthwt'])
\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\
KeyError                                  Traceback (most recent call last)
<ipython-input-60-98d2eeccb488> in <module>()
----> 1 smoker_and_wt = baby.select(['m_smoker', 'birthwt'])

../datascience/tables.py in select(self, *column_label_or_labels)
    584         table = Table()
    585         for label in labels:
--> 586             self._add_column_and_format(table, label, np.copy(self[label]))
    587         return table
    588

../datascience/tables.py in __getitem__(self, index_or_label)
    167     def __getitem__(self, index_or_label):
    168         label = self._as_label(index_or_label)
--> 169         return self.column(label)
    170
    171     def __setitem__(self, label, values):

../datascience/tables.py in column(self, index_or_label)
```

```
    265              An instance of ``numpy.array``.
    266          """
--> 267          return self._columns[self._as_label(index_or_label)]
    268
    269      # Deprecated


KeyError: 'm_smoker'


In [61]: smoker_and_wt
\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\
NameError                                 Traceback (most recent call last)
<ipython-input-61-1c8046ed122a> in <module>()
----> 1 smoker_and_wt

NameError: name 'smoker_and_wt' is not defined
```

Let's compare the number of smokers to non-smokers.

```
In [62]: smoker_and_wt.select('m_smoker').hist(bins = [0, 1, 2]);
```


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('m_smoker', 0).select('birthwt').hist()
---------------------------------------------------------------------------
NameError                                 Traceback (most recent call last)
<ipython-input-63-ac023d72d786> in <module>()
----> 1 smoker_and_wt.where('m_smoker', 0).select('birthwt').hist()

NameError: name 'smoker_and_wt' is not defined
```

```
# Smokers
In [64]: smoker_and_wt.where('m_smoker', 1).select('birthwt').hist()
---------------------------------------------------------------------------
NameError                                 Traceback (most recent call last)
<ipython-input-64-14a160d7b1d6> in <module>()
----> 1 smoker_and_wt.where('m_smoker', 1).select('birthwt').hist()

NameError: name 'smoker_and_wt' is not defined
```

_build/latex/_images/not_m_smoker_weights.png

_build/latex/_images/m_smoker_weights.png

What's the difference in mean birth weight of the two categories?

```
In [65]: nonsmoking_mean = smoker_and_wt.where('m_smoker', 0).column('birthwt').mean()
---------------------------------------------------------------------------
NameError                                 Traceback (most recent call last)
<ipython-input-65-f41eeafe6bd7> in <module>()
----> 1 nonsmoking_mean = smoker_and_wt.where('m_smoker', 0).column('birthwt').mean()

NameError: name 'smoker_and_wt' is not defined

In [66]: smoking_mean = smoker_and_wt.where('m_smoker', 1).column('birthwt').mean()
\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\
NameError                                 Traceback (most recent call last)
<ipython-input-66-025a08c0d2c6> in <module>()
----> 1 smoking_mean = smoker_and_wt.where('m_smoker', 1).column('birthwt').mean()

NameError: name 'smoker_and_wt' is not defined

In [67]: observed_diff = nonsmoking_mean - smoking_mean
\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\
NameError                                 Traceback (most recent call last)
<ipython-input-67-1dc0201ce02b> in <module>()
----> 1 observed_diff = nonsmoking_mean - smoking_mean

NameError: name 'nonsmoking_mean' is not defined

In [68]: observed_diff
\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\
NameError                                 Traceback (most recent call last)
<ipython-input-68-dbee69efdaa6> in <module>()
----> 1 observed_diff
```

```
NameError: name 'observed_diff' is not defined
```

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

```
In [69]: num_nonsmokers = smoker_and_wt.where('m_smoker', 0).num_rows
---------------------------------------------------------------------------
NameError                                 Traceback (most recent call last)
<ipython-input-69-66887dbcfe42> in <module>()
----> 1 num_nonsmokers = smoker_and_wt.where('m_smoker', 0).num_rows

NameError: name 'smoker_and_wt' is not defined

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('birthwt')[:num_nonsmokers].mean() - \
    ....:         resample.column('birthwt')[num_nonsmokers:].mean()
    ....:     return bootstrap_diff
    ....:

In [71]: repetitions = 1000

In [72]: bootstrapped_diff_means = np.array(
    ....:     [ bootstrap_once() for _ in range(repetitions) ])
    ....:
---------------------------------------------------------------------------
NameError                                 Traceback (most recent call last)
<ipython-input-72-167c9ef03394> in <module>()
      1 bootstrapped_diff_means = np.array(
----> 2     [ bootstrap_once() for _ in range(repetitions) ])

<ipython-input-72-167c9ef03394> in <listcomp>(.0)
      1 bootstrapped_diff_means = np.array(
----> 2     [ bootstrap_once() for _ in range(repetitions) ])

<ipython-input-70-0af534563c70> in bootstrap_once()
      5     We then split according to the number of nonsmokers in the original sample.
      6     """
----> 7     resample = smoker_and_wt.sample(with_replacement = True)
      8     bootstrap_diff = resample.column('birthwt')[:num_nonsmokers].mean() -        resample.co
      9     return bootstrap_diff

NameError: name 'smoker_and_wt' is not defined

In [73]: bootstrapped_diff_means[:10]
\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\
NameError                                 Traceback (most recent call last)
<ipython-input-73-7ce34a8f5ddf> in <module>()
----> 1 bootstrapped_diff_means[:10]

NameError: name 'bootstrapped_diff_means' is not defined

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

```
NameError                                 Traceback (most recent call last)
<ipython-input-74-fea00db6f1ce> in <module>()
----> 1 num_diffs_greater = (abs(bootstrapped_diff_means) > abs(observed_diff)).sum()

NameError: name 'bootstrapped_diff_means' is not defined

In [75]: p_value = num_diffs_greater / len(bootstrapped_diff_means)
\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\
NameError                                 Traceback (most recent call last)
<ipython-input-75-053c4de37c34> in <module>()
----> 1 p_value = num_diffs_greater / len(bootstrapped_diff_means)

NameError: name 'num_diffs_greater' is not defined

In [76]: p_value
\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\
NameError                                 Traceback (most recent call last)
<ipython-input-76-1336a4a7d880> in <module>()
----> 1 p_value

NameError: name 'p_value' is not defined
```

1.8 Drawing Maps

To come.

Reference

2.1 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`:

```
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:

```
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.

At the time of this writing, most methods only have one or two sentences of documentation, so what you see here is all that you'll get for the time being. We are actively working on documentation, prioritizing the most complicated methods (mostly visualizations).

Creation

Table.__init__([labels, _deprecated, formatter])	Create an empty table with column labels.
Table.from_records(records)	Create a table from a sequence of records (dicts with fixed keys).
Table.read_table(filepath_or_buffer, *args, ...)	Read a table from a file or web address.
Table.from_df(df)	Convert a Pandas DataFrame into a Table.
Table.from_array(arr)	Convert a structured NumPy array into a Table.

2.1.1 datascience.tables.Table.__init__

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

```
>>> tiles = Table(['letter', 'count', 'points'])
>>> tiles
letter | count | points
```

　　Args: `labels` (list of strings): The column labels.

　　　　formatter (Formatter): An instance of `Formatter` that formats the columns' values.

2.1.2 datascience.tables.Table.from_records

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

2.1.3 datascience.tables.Table.read_table

classmethod Table.**read_table**(*filepath_or_buffer*, **args*, ***vargs*)
　　Read a table from a file or web address.

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

2.1.4 datascience.tables.Table.from_df

classmethod Table.**from_df**(*df*)
　　Convert a Pandas DataFrame into a Table.

2.1.5 datascience.tables.Table.from_array

classmethod Table.**from_array**(*arr*)
　　Convert a structured NumPy array into a Table.

Extension (does not modify original table)

Table.with_column(label, values)	Return a 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)	Returns a table with label changed to new_label.

2.1.6 datascience.tables.Table.with_column

Table.**with_column**(*label*, *values*)
　　Return a table with an additional or replaced column.

　　Args:

label (str): The column label. If an existing label is used, that column will be replaced in the returned table.

values (single value or sequence): If a single value, every value in the new column is `values`.

If a sequence, the new column contains the values in `values`. `values` must be the same length as the table.

Raises:

ValueError: If

- `label` is not a valid column name
- `values` is a list/array and does not have the same length as the number of rows in the table.

```
>>> tiles = Table().with_columns([
...     'letter', ['c', 'd'],
...     'count',  [2, 4],
... ])
>>> tiles.with_column('points', [3, 2])
letter | count | points
c      | 2     | 3
d      | 4     | 2
>>> tiles.with_column('count', 1)
letter | count
c      | 1
d      | 1
```

2.1.7 datascience.tables.Table.with_columns

Table.**with_columns**(*labels_and_values*)
    Return a table with additional or replaced columns.

Args:

labels_and_values: An alternating list of labels and values or a list of label-values pairs.

```
>>> Table().with_columns([
...     'letter', ['c', 'd'],
...     'count',  [2, 4],
... ])
letter | count
c      | 2
d      | 4
>>> Table().with_columns(
...     'letter', ['c', 'd'],
...     'count',  [2, 4],
... )
letter | count
c      | 2
d      | 4
>>> Table().with_columns([
...     ['letter', ['c', 'd']],
...     ['count',  [2, 4]],
... ])
letter | count
c      | 2
d      | 4
>>> Table().with_columns(
```

```
...        ['letter', ['c', 'd']],
...        ['count',  [2, 4]],
... )
letter | count
c      | 2
d      | 4
>>> Table().with_columns([
...        ['letter', ['c', 'd']],
... ])
letter
c
d
>>> Table().with_columns(
...        'letter', ['c', 'd'],
... )
letter
c
d
>>> Table().with_columns(
...        ['letter', ['c', 'd']],
... )
letter
c
d
>>> Table().with_columns({'letter': ['c', 'd']})
letter
c
d
```

2.1.8 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(['letter', 'count', 'points'])
>>> tiles.with_row(['c', 2, 3]).with_row(['d', 4, 2])
letter | count | points
c      | 2     | 3
d      | 4     | 2
```

2.1.9 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(['letter', 'count', 'points'])
>>> tiles.with_rows([['c', 2, 3], ['d', 4, 2]])
```

```
letter | count | points
c      | 2     | 3
d      | 4     | 2
```

2.1.10 datascience.tables.Table.relabeled

Table.**relabeled**(*label*, *new_label*)
    Returns a table with label changed to new_label.

    label and new_label may be single values or lists specifying column labels to be changed and their new corresponding labels.

    Args:

        label (str or sequence of str): The label(s) of columns to be changed.

        new_label (str or sequence of str): The new label(s) of columns to be changed. Same number of elements as label.

```
>>> tiles = Table(['letter', 'count'])
>>> tiles = tiles.with_rows([['c', 2], ['d', 4]])
>>> tiles.relabeled('count', 'number')
letter | number
c      | 2
d      | 4
```

Accessing values

Table.num_columns	Number of columns.
Table.columns	
Table.column(index_or_label)	Return the values of a column as an array.
Table.num_rows	Number of rows.
Table.rows	Return a view of all rows.
Table.row(index)	Return a row.
Table.labels	Return a tuple of column labels.
Table.column_index(column_label)	Return the index of a column.
Table.apply(fn[, column_label])	Returns an array where fn is applied to each set of elements by row from the specified c

2.1.11 datascience.tables.Table.num_columns

Table.**num_columns**
    Number of columns.

2.1.12 datascience.tables.Table.columns

Table.**columns**

2.1.13 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].

```
>>> tiles = Table().with_columns([
...     'letter', ['c', 'd'],
...     'count',  [2, 4],
... ])
>>> 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`.

2.1.14 datascience.tables.Table.num_rows

Table.**num_rows**
    Number of rows.

2.1.15 datascience.tables.Table.rows

Table.**rows**
    Return a view of all rows.

2.1.16 datascience.tables.Table.row

Table.**row**(*index*)
    Return a row.

2.1.17 datascience.tables.Table.labels

Table.**labels**
    Return a tuple of column labels.

2.1.18 datascience.tables.Table.column_index

Table.**column_index**(*column_label*)
    Return the index of a column.

2.1.19 datascience.tables.Table.apply

Table.**apply**(*fn*, *column_label=None*)
    Returns an array where `fn` is applied to each set of elements by row from the specified columns in `column_label`. If no column_label is specified, then each row is passed to fn.

    Args:

        fn (function): The function to be applied to elements specified by `column_label`.

        column_label (single string or list of strings): Names of columns to be passed into function `fn`. Length must match number of elements `fn` takes.

Raises:

ValueError: column name in `column_label` is not an existing column in the table.

Returns: A numpy array consisting of results of applying `fn` to elements specified by `column_label` in each row.

```
>>> t = Table().with_columns([
...       'letter', ['a', 'b', 'c', 'z'],
...       'count',  [9, 3, 3, 1],
...       'points', [1, 2, 2, 10]])
>>> t
letter | count | points
a      | 9     | 1
b      | 3     | 2
c      | 3     | 2
z      | 1     | 10
>>> t.apply(lambda x: x - 1, 'points')
array([0, 1, 1, 9])
>>> t.apply(lambda x, y: x * y, ['count', 'points'])
array([ 9,  6,  6, 10])
```

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

```
>>> t.apply(lambda row: row.item('count') * 2)
array([18,  6,  6,  2])
```

Mutation (modifies table in place)

Table.set_format(column_label_or_labels, ...)	Set the format of a column.
Table.move_to_start(column_label)	Move a column to the first in order.
Table.move_to_end(column_label)	Move a column to the last in order.
Table.append(row_or_table)	Append a row or all rows of a table.
Table.append_column(label, values)	Appends a column to the table or replaces a column.
Table.relabel(column_label, new_label)	Change the labels of columns specified by `column_label` to labels in new_

2.1.20 datascience.tables.Table.set_format

Table.**set_format**(*column_label_or_labels*, *formatter*)
    Set the format of a column.

2.1.21 datascience.tables.Table.move_to_start

Table.**move_to_start**(*column_label*)
    Move a column to the first in order.

2.1.22 datascience.tables.Table.move_to_end

Table.**move_to_end**(*column_label*)
    Move a column to the last in order.

2.1.23 datascience.tables.Table.append

Table.**append**(*row_or_table*)

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

2.1.24 datascience.tables.Table.append_column

Table.**append_column**(*label*, *values*)

    Appends a column to the table or replaces a column.

        __setitem__ is aliased to this method: `table.append_column('new_col', [1, 2, 3])` is equivalent to `table['new_col'] = [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.

    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.

```
>>> table = Table().with_columns([
...     'letter', ['a', 'b', 'c', 'z'],
...     'count',  [9, 3, 3, 1],
...     'points', [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', [10, 20, 30, 40])
>>> table
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')
>>> table
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
>>> table.append_column(123, [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.
```

2.1.25 datascience.tables.Table.relabel

Table.**relabel**(*column_label*, *new_label*)
Change the labels of columns specified by column_label to labels in new_label.

Args:

column_label (single str or list/array of str): The label(s) of columns to be changed. Must be str.

new_label (single str or list/array of str): The new label(s) of columns to be changed. Must be str.

Number of elements must match number of elements in column_label.

Returns: Original table with modified labels

```
>>> table = Table().with_columns([
...      'points', (1, 2, 3),
...      'id',     (12345, 123, 5123)])
>>> table.relabel('id', 'yolo')
points | yolo
1      | 12345
2      | 123
3      | 5123
>>> table.relabel(['points', 'yolo'], ['red', 'blue'])
red  | blue
1    | 12345
2    | 123
3    | 5123
>>> table.relabel(['red', 'green', 'blue'], ['cyan', 'magenta', 'yellow', 'key'])
Traceback (most recent call last):
    ...
ValueError: Invalid arguments. column_label and new_label must be of equal length.
>>> table.relabel(['red', 'blue'], ['blue', 'red'])
blue | red
1    | 12345
2    | 123
3    | 5123
```

Transformation (creates a new table)

Table.copy(*[, shallow])	Return a copy of a Table.
Table.select(*column_label_or_labels)	Return a Table with selected column or columns by label or index.
Table.drop(*column_label_or_labels)	Return a Table with only columns other than selected label or labels.
Table.take()	Return a new Table of a sequence of rows taken by number.
Table.exclude()	Return a new Table without a sequence of rows excluded by number.
Table.where(column_or_label[, ...])	Return a Table of rows for which the column is value or a non-zero value.
Table.sort(column_or_label[, descending, ...])	Return a Table of rows sorted according to the values in a column.
Table.group(column_or_label[, collect])	Group rows by unique values in a column; count or aggregate others.
Table.groups(labels[, collect])	Group rows by multiple columns, count or aggregate others.
Table.pivot(columns, rows[, values, ...])	Generate a table with a column for rows (or a column for each row in rows list)
Table.stack(key[, labels])	Takes k original columns and returns two columns, with col.
Table.join(column_label, other[, other_label])	Generate a table with the columns of self and other, containing rows for all value
Table.stats([ops])	Compute statistics for each column and place them in a table.

Table 2.5 – continued from previous page

Table.percentile(p)	Returns a new table with one row containing the pth percentile for each column.
Table.sample([k, with_replacement, weights])	Returns a new table where k rows are randomly sampled from the original table.
Table.split(k)	Returns a tuple of two tables where the first table contains k rows randomly sam
Table.bin([select])	Group values by bin and compute counts per bin by column.

2.1.26 datascience.tables.Table.copy

Table.**copy**(**, shallow=False*)
    Return a copy of a Table.

2.1.27 datascience.tables.Table.select

Table.**select**(**column_label_or_labels*)
    Return a Table with selected column or columns by label or index.

    Args: column_label_or_labels (string, list of strings, or several separate argument strings): The header names or indices of the columns to be selected. column_label_or_labels must be an existing header name, or a valid column index, or a list thereof.

    Returns: An instance of Table containing only selected columns.

```
>>> t = Table().with_columns([
...     'burgers',  ['cheeseburger', 'hamburger', 'veggie burger'],
...     'prices',   [6, 5, 5],
...     'calories', [743, 651, 582]])
>>> t
burgers        | prices | calories
cheeseburger   | 6      | 743
hamburger      | 5      | 651
veggie burger  | 5      | 582
>>> t.select(['burgers', 'calories'])
burgers        | calories
cheeseburger   | 743
hamburger      | 651
veggie burger  | 582
>>> t.select('prices')
prices
6
5
5
>>> t.select('burgers', 'calories')
burgers        | calories
cheeseburger   | 743
hamburger      | 651
veggie burger  | 582
>>> t.select(1)
prices
6
5
5
>>> t.select([2, 0])
calories | burgers
743      | cheeseburger
651      | hamburger
582      | veggie burger
```

```
>>> t.select(2, 0)
calories | burgers
743      | cheeseburger
651      | hamburger
582      | veggie burger
```

2.1.28 datascience.tables.Table.drop

Table.**drop**(*column_label_or_labels*)

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

> **Args:** column_label_or_labels (string or list of strings): The header names or indices of the columns to be dropped. column_label_or_labels must be an existing header name, or a valid column index.
>
> **Returns:** An instance of Table with given columns removed.

```
>>> t = Table().with_columns([
...     'burgers',  ['cheeseburger', 'hamburger', 'veggie burger'],
...     'prices',   [6, 5, 5],
...     'calories', [743, 651, 582]])
>>> t
burgers       | prices | calories
cheeseburger  | 6      | 743
hamburger     | 5      | 651
veggie burger | 5      | 582
>>> t.drop('prices')
burgers       | calories
cheeseburger  | 743
hamburger     | 651
veggie burger | 582
>>> t.drop(['burgers', 'calories'])
prices
6
5
5
>>> t.drop('burgers', 'calories')
prices
6
5
5
>>> t.drop([0, 2])
prices
6
5
5
>>> t.drop(0, 2)
prices
6
5
5
>>> t.drop(1)
burgers       | calories
cheeseburger  | 743
hamburger     | 651
veggie burger | 582
```

2.1.29 datascience.tables.Table.take

Table.**take**()

Return a new Table of a sequence of rows taken 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 selected.

Returns: A new instance of `Table`.

```
>>> t = Table().with_columns([
...     'letter grade', ['A+', 'A', 'A-', 'B+', 'B', 'B-'],
...     'gpa', [4, 4, 3.7, 3.3, 3, 2.7]])
>>> t
letter grade | gpa
A+           | 4
A            | 4
A-           | 3.7
B+           | 3.3
B            | 3
B-           | 2.7
>>> t.take(0)
letter grade | gpa
A+           | 4
>>> t.take(5)
letter grade | gpa
B-           | 2.7
>>> t.take(-1)
letter grade | gpa
B-           | 2.7
>>> t.take([2, 1, 0])
letter grade | gpa
A-           | 3.7
A            | 4
A+           | 4
>>> t.take([1, 5])
letter grade | gpa
A            | 4
B-           | 2.7
>>> t.take(range(3))
letter grade | gpa
A+           | 4
A            | 4
A-           | 3.7
```

Note that `take` also supports NumPy-like indexing and slicing:

```
>>> t.take[:3]
letter grade | gpa
A+           | 4
A            | 4
A-           | 3.7
```

```
>>> t.take[2, 1, 0]
letter grade | gpa
A-           | 3.7
A            | 4
A+           | 4
```

2.1.30 datascience.tables.Table.exclude

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.

```
>>> t = Table().with_columns([
...     'letter grade', ['A+', 'A', 'A-', 'B+', 'B', 'B-'],
...     'gpa', [4, 4, 3.7, 3.3, 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
B-           | 2.7
>>> t.exclude(-1)
letter grade | gpa
A+           | 4
A            | 4
A-           | 3.7
B+           | 3.3
B            | 3
>>> t.exclude([1, 3, 4])
letter grade | gpa
A+           | 4
A-           | 3.7
B-           | 2.7
>>> t.exclude(range(3))
letter grade | gpa
B+           | 3.3
B            | 3
B-           | 2.7
```

Note that exclude also supports NumPy-like indexing and slicing:

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

```
>>> t.exclude[1, 3, 4]
letter grade | gpa
A+           | 4
```

```
A-           | 3.7
B-           | 2.7
```

2.1.31 datascience.tables.Table.where

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

Return a Table of rows for which the column is `value` or a non-zero value.

If `column_or_label` contains Boolean values, returns rows corresponding to True.

Args: `column_or_label`: The header name of a column in the table or an array.

   `value_or_predicate`: Value to compare to items in column or function to apply to items in column.

Returns: An instance of `Table` containing rows for which the `column_or_label` column or `column_or_label` itself is non-zero or True, or is equal to `value`, if provided.

```
>>> marbles = Table().with_columns([
...     "Color", ["Red", "Green", "Blue", "Red", "Green", "Green"],
...     "Shape", ["Round", "Rectangular", "Rectangular", "Round", "Rectangular", "Round"],
...     "Amount", [4, 6, 12, 7, 9, 2],
...     "Price", [1.30, 1.20, 2.00, 1.75, 1.40, 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      | 1.4
Green | Round       | 2      | 3
>>> marbles.where("Shape", "Round")
Color | Shape | Amount | Price
Red   | Round | 4      | 1.3
Red   | Round | 7      | 1.75
Green | Round | 2      | 3
>>> marbles.where(marbles.column("Shape") == "Round") # equivalent to previous example
Color | Shape | Amount | Price
Red   | Round | 4      | 1.3
Red   | Round | 7      | 1.75
Green | Round | 2      | 3
>>> marbles.where(marbles.column("Price") > 1.5)
Color | Shape       | Amount | Price
Blue  | Rectangular | 12     | 2
Red   | Round       | 7      | 1.75
Green | Round       | 2      | 3
```

You can also use predicates to simplify single-column comparisons.

```
>>> from datascience.predicates import are
>>> marbles.where("Price", are.above(1.5)) # equivalent to previous example
Color | Shape       | Amount | Price
Blue  | Rectangular | 12     | 2
Red   | Round       | 7      | 1.75
Green | Round       | 2      | 3
```

And apply some predicates to compare columns.

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

2.1.32 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`.

```
>>> marbles = Table().with_columns([
...     "Color", ["Red", "Green", "Blue", "Red", "Green", "Green"],
...     "Shape", ["Round", "Rectangular", "Rectangular", "Round", "Rectangular", "Round"],
...     "Amount", [4, 6, 12, 7, 9, 2],
...     "Price", [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.sort("Amount")
Color | Shape       | Amount | Price
Green | Round       | 2      | 1
Red   | Round       | 4      | 1.3
Green | Rectangular | 6      | 1.3
Red   | Round       | 7      | 1.75
Green | Rectangular | 9      | 1.4
Blue  | Rectangular | 12     | 2
>>> marbles.sort("Amount", descending = True)
Color | Shape       | Amount | Price
Blue  | Rectangular | 12     | 2
Green | Rectangular | 9      | 1.4
Red   | Round       | 7      | 1.75
Green | Rectangular | 6      | 1.3
Red   | Round       | 4      | 1.3
Green | Round       | 2      | 1
>>> marbles.sort(3) # the Price column
Color | Shape       | Amount | Price
Green | Round       | 2      | 1
Red   | Round       | 4      | 1.3
Green | Rectangular | 6      | 1.3
Green | Rectangular | 9      | 1.4
Red   | Round       | 7      | 1.75
Blue  | Rectangular | 12     | 2
>>> 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
```

2.1.33 datascience.tables.Table.group

Table.**group**(*column_or_label*, *collect=None*)

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

Args: column_or_label: values to group (column label or index, or array)

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

Returns: A Table with each row corresponding to a unique value in column_or_label, where the first column contains the unique values from column_or_label, and the second contains 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 column_or_label, then applying collect to each set of grouped values in the other columns.

Note: The grouped column 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.

```
>>> marbles = Table().with_columns([
...     "Color", ["Red", "Green", "Blue", "Red", "Green", "Green"],
...     "Shape", ["Round", "Rectangular", "Rectangular", "Round", "Rectangular", "Round"],
...     "Amount", [4, 6, 12, 7, 9, 2],
...     "Price", [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") # just gives counts
Color | count
Blue  | 1
Green | 3
Red   | 2
>>> marbles.group("Color", max) # takes the max of each grouping, in each column
Color | Shape max   | Amount max | Price max
Blue  | Rectangular | 12         | 2
Green | Round       | 9          | 1.4
Red   | Round       | 7          | 1.75
>>> marbles.group("Shape", sum) # sum doesn't make sense for strings
Shape       | Color sum | Amount sum | Price sum
Rectangular |           | 27         | 4.7
Round       |           | 13         | 4.05
```

2.1.34 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.

```
>>> marbles = Table().with_columns([
...     "Color", ["Red", "Green", "Blue", "Red", "Green", "Green"],
...     "Shape", ["Round", "Rectangular", "Rectangular", "Round", "Rectangular", "Round"],
...     "Amount", [4, 6, 12, 7, 9, 2],
...     "Price", [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.groups(["Color", "Shape"])
Color | Shape       | count
Blue  | Rectangular | 1
Green | Rectangular | 2
Green | Round       | 1
Red   | Round       | 2
>>> marbles.groups(["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
```

2.1.35 datascience.tables.Table.pivot

`Table.`**`pivot`**(*columns*, *rows*, *values=None*, *collect=None*, *zero=None*)
Generate a table with a column for rows (or a column for each row in rows list) and a column for each unique value in columns. Each row counts/aggregates the values that match both row and column.

columns – column label in self rows – column label or a list of column labels values – column label in self (or None to produce counts) collect – aggregation function over values zero – zero value for non-existent row-column combinations

2.1.36 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.

2.1.37 datascience.tables.Table.join

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

> Generate a table with the columns of self and other, containing rows for all values of a column that appear in both tables. If a join value appears more than once in self, each row will be used, but in the other table, only the first of each will be used.
>
> If the result is empty, return None.

2.1.38 datascience.tables.Table.stats

Table.**stats**(*ops=(<built-in function min>, <built-in function max>, <function median at 0x7fc9c6359620>, <built-in function sum>)*)

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

2.1.39 datascience.tables.Table.percentile

Table.**percentile**(*p*)

> Returns 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 at least as large as the p% of numbers in the column.

```
>>> table = Table().with_columns([
...     'count', [9, 3, 3, 1],
...     'points', [1, 2, 2, 10]])
>>> table
count | points
9     | 1
3     | 2
3     | 2
1     | 10
>>> table.percentile(80)
count | points
9     | 10
```

2.1.40 datascience.tables.Table.sample

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

> Returns a new table where k rows are randomly sampled from the original table.
>
> **Kwargs:**
>
> > **k (int or None): If None (default), all the rows in the table are** sampled. If an integer, k rows from the original table are sampled.
> >
> > **with_replacement (bool): If False (default), samples the rows** without replacement. If True, samples the rows with replacement.
> >
> > **weights (list/array or None): If None (default), samples the rows** using a uniform random distribution. If a list/array is passed in, it must be the same length as the number of rows in the table and the values must sum to 1. The rows will then be sampled according the the probability distribution in `weights`.

Returns: A new instance of `Table`.

```
>>> jobs = Table().with_columns([
...     'job',  ['a', 'b', 'c', 'd'],
...     'wage', [10, 20, 15, 8]])
>>> jobs
job  | wage
a    | 10
b    | 20
c    | 15
d    | 8
>>> jobs.sample()
job  | wage
b    | 20
c    | 15
a    | 10
d    | 8
>>> jobs.sample(k = 2)
job  | wage
b    | 20
c    | 15
>>> jobs.sample(k = 2, with_replacement = True,
...      weights = [0.5, 0.5, 0, 0])
job  | wage
a    | 10
a    | 10
```

2.1.41 datascience.tables.Table.split

Table.**split**(*k*)

Returns 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`.

```
>>> jobs = Table().with_columns([
...     'job',  ['a', 'b', 'c', 'd'],
...     'wage', [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
a    | 10
b    | 20
>>> rest
```

```
    job   | wage
    d     | 8
```

2.1.42 datascience.tables.Table.bin

Table.**bin**(*select=None*, ***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:

> **select (columns): Columns to be binned. If None, 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 the value of the probability density function at the bin, 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.

Exporting / Displaying

Table.show([max_rows])	Display the table.
Table.as_text([max_rows, sep])	Format table as text.
Table.as_html([max_rows])	Format table as HTML.
Table.index_by(column_or_label)	Return a dict keyed by values in a column that contains lists of rows corresponding to each
Table.to_array()	Convert the table to a NumPy array.
Table.to_df()	Convert the table to a Pandas DataFrame.
Table.to_csv(filename)	Creates a CSV file with the provided filename.

2.1.43 datascience.tables.Table.show

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

 Display the table.

2.1.44 datascience.tables.Table.as_text

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

 Format table as text.

2.1.45 datascience.tables.Table.as_html

Table.**as_html**(*max_rows=0*)
    Format table as HTML.

2.1.46 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.

2.1.47 datascience.tables.Table.to_array

Table.**to_array**()
    Convert the table to a NumPy array.

2.1.48 datascience.tables.Table.to_df

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

2.1.49 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`.

```
>>> jobs = Table().with_columns([
...     'job',  ['a', 'b', 'c', 'd'],
...     'wage', [10, 20, 15, 8]])
>>> jobs
job  | wage
a    | 10
b    | 20
c    | 15
d    | 8
>>> jobs.to_csv('my_table.csv')
<outputs a file called my_table.csv in the current directory>
```

Visualizations

Table.plot([column_for_xticks, select, overlay])	Plot line charts for the table.
Table.bar([column_for_categories, select, ...])	Plot bar charts for the table.
Table.barh([column_for_categories, select, ...])	Plot horizontal bar charts for the table.
Table.pivot_hist(pivot_column_label, ...[, ...])	Draw histograms of each category in a column.
Table.hist([select, overlay, bins, counts, unit])	Plots one histogram for each column in the table.
Table.points(column__lat, column__long[, ...])	Draw points from latitude and longitude columns.
Continued on next page	

Table 2.7 – continued from previous page	
Table.scatter(column_for_x[, select, ...])	Creates scatterplots, optionally adding a line of best fit.
Table.boxplot(**vargs)	Plots a boxplot for the table.

2.1.50 datascience.tables.Table.plot

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

Plot line charts for the table.

Each plot is labeled using the values in *column_for_xticks* and one plot is produced for every other column (or for the columns designated by *select*).

Every selected column except for *column_for_xticks* must be numerical.

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 will be displayed separately.

> **vargs: Additional arguments that get passed into *plt.plot*.** See http://matplotlib.org/api/pyplot_api.html#matplotlib.pyplo for additional arguments that can be passed into vargs.

2.1.51 datascience.tables.Table.bar

Table.**bar**(*column_for_categories=None*, *select=None*, *overlay=True*, ***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 except column for *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.pyplo for additional arguments that can be passed into vargs.

2.1.52 datascience.tables.Table.barh

Table.**barh**(*column_for_categories=None*, *select=None*, *overlay=True*, ***vargs*)

Plot horizontal 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 except column for *column_for_categories* must be numerical.

Args: column_for_categories (str): A column containing y-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.barh*.** See http://matplotlib.org/api/pyplot_api.html#matplotlib.pypl
> for additional arguments that can be passed into vargs.

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

2.1.53 datascience.tables.Table.pivot_hist

Table.**pivot_hist**(*pivot_column_label*, *value_column_label*, *overlay=True*, ***vargs*)
    Draw histograms of each category in a column.

2.1.54 datascience.tables.Table.hist

Table.**hist**(*select=None*, *overlay=True*, *bins=None*, *counts=None*, *unit=None*, ***vargs*)
    Plots one histogram for each column in the table.

    Every column must be numerical.

    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.

> **bins (column name or list): Lower bound for each bin in the** histogram. If None, bins will be chosen automatically.

> **counts (column name or column): A column of counted values.** All other columns are treated as counts of these values. If None, each value in each row is assigned a count of 1.

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

```
>>> t = Table().with_columns([
...     'count',  [9, 3, 3, 1],
...     'points', [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>
```

```
>>> t = Table().with_columns([
...     'value',      [101, 102, 103],
...     'proportion', [0.25, 0.5, 0.25]])
>>> t.hist(counts='value')
<histogram of values in prop weighted by corresponding values in value>
```

2.1.55 datascience.tables.Table.points

Table.**points**(*column__lat*, *column__long*, *labels=None*, *colors=None*, ***kwargs*)
    Draw points from latitude and longitude columns. [Deprecated]

2.1.56 datascience.tables.Table.scatter

Table.**scatter**(*column_for_x*, *select=None*, *overlay=True*, *fit_line=False*, *colors=None*, *labels=None*, ***vargs*)
    Creates scatterplots, optionally adding a line of best fit.

    Each plot uses the values in *column_for_x* for horizontal positions. One plot is produced for every other column as y (or for the columns designated by *select*).

    Every selected except column for *column_for_categories* must be numerical.

    Args:

    > `column_for_x` (str): The name to use for the x-axis values of the scatter plots.

    Kwargs:

    > `overlay` (bool): create a chart with one color per data column; if False, each 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.p
    > for additional arguments that can be passed into vargs. These include: *marker* and *norm*, to name a couple.

    > `colors`: A column of colors (labels or numeric values)

    > `labels`: A column of text labels to annotate dots

```
>>> table = Table().with_columns([
...     'x', [9, 3, 3, 1],
...     'y', [1, 2, 2, 10],
...     'z', [3, 4, 5, 6]])
>>> table
x    | y    | z
9    | 1    | 3
3    | 2    | 4
3    | 2    | 5
1    | 10   | 6
>>> table.scatter('x')
<scatterplot of values in y and z on x>
```

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

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

2.1.57 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 http://matplotlib.org/api/pyplot_api.html#matplotlib.py
            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.

```
>>> table = Table().with_columns([
...      'test1', [92.5, 88, 72, 71, 99, 100, 95, 83, 94, 93],
...      'test2', [89, 84, 74, 66, 92, 99, 88, 81, 95, 94]])
>>> table
test1 | test2
92.5  | 89
88    | 84
72    | 74
71    | 66
99    | 92
100   | 99
95    | 88
83    | 81
94    | 95
93    | 94
>>> table.boxplot()
<boxplot of test1 and boxplot of test2 side-by-side on the same figure>
```

2.2 Maps (`datascience.maps`)

Draw maps using folium.

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:

            'BuGn', 'BuPu', 'GnBu', 'OrRd', 'PuBu', 'PuBuGn', 'PuRd', 'RdPu', 'YlGn', 'YlGnBu', 'YlOrBr', and 'YlOrRd'.

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].

`features`

`format`(***kwargs*)
   Apply formatting.

`geojson`()
   Render features as a FeatureCollection.

classmethod **`read_geojson`**(*path_or_json_or_string*)
   Read a geoJSON string, object, or file. 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 – fill color

   Defaults from Folium:

   marker_icon: string, default 'info-sign' icon from (http://getbootstrap.com/components/) you want on the marker

   clustered_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

   `copy`()
      Return a deep copy

   `format`(***kwargs*)
      Apply formatting.

   `geojson`(*feature_id*)
      GeoJSON representation of the marker as a point.

   `lat_lons`

   classmethod **`map`**(*latitudes*, *longitudes*, *labels=None*, *colors=None*, *radii=None*, ***kwargs*)
      Return markers from columns of coordinates, labels, & colors.

      The radii column is not applicable to markers, but sets circle radius.

   classmethod **`map_table`**(*table*, ***kwargs*)
      Return markers from the colums of a table.

class datascience.maps.**`Circle`**(*lat*, *lon*, *popup=''*, *color='blue'*, *radius=10*, ***kwargs*)
   A marker displayed with Folium's circle_marker method.

   popup – text that pops up when marker is clicked color – fill color radius – pixel radius of the circle

Defaults from Folium:

fill_opacity: float, default 0.6 Circle fill opacity

For example, to draw three circles:

t = Table().with_columns([

> 'lat', [37.8, 38, 37.9], 'lon', [-122, -122.1, -121.9], 'label', ['one', 'two', 'three'], 'color', ['red', 'green', 'blue'], 'radius', [3000, 4000, 5000],

>])

Circle.map_table(t)

class `datascience.maps.`**`Region`**(*geojson*, ***kwargs*)
   A GeoJSON feature displayed with Folium's geo_json method.

   `copy`()
      Return a deep copy

   `format`(***kwargs*)
      Apply formatting.

   `geojson`(*feature_id*)
      Return GeoJSON with ID substituted.

   `lat_lons`
      A flat list of (lat, lon) pairs.

   `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.

   `properties`

   `type`
      The GEOJSON type of the regions: Polygon or MultiPolygon.

2.3 Formats (`datascience.formats`)

String formatting for table entries.

class `datascience.formats.`**`Formatter`**(*min_width=None*, *max_width=None*, *etc=None*)
   String formatter that truncates long values.

   static **`convert`**(*value*)
      Identity conversion (override to convert values).

   `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=', '*)
>   Format numbers that may have delimiters.

>   **convert**(*value*)
>   >   Convert string 93,000.00 to float 93000.0.

>   **converts_values** = **True**

>   **format_value**(*value*)

class datascience.formats.**CurrencyFormatter**(*symbol='$'*, **args*, ***vargs*)
>   Format currency and convert to float.

>   **convert**(*value*)
>   >   Convert value to float. If value is a string, ensure that the first character is the same as symbol ie. the value
>   >   is in the currency this formatter is representing.

>   **converts_values** = **True**

>   **format_value**(*value*)
>   >   Format currency.

class datascience.formats.**DateFormatter**(*format='%Y-%m-%d   %H:%M:%S.%f'*,   **args*,
>   >   ***vargs*)
>   Format date & time and convert to UNIX timestamp.

>   **convert**(*value*)
>   >   Convert 2015-08-03 to a Unix timestamp int.

>   **converts_values** = **True**

>   **format_value**(*value*)
>   >   Format timestamp as a string.

class datascience.formats.**PercentFormatter**(*decimals=2*, **args*, ***vargs*)
>   Format a number as a percentage.

>   **converts_values** = **False**

>   **format_value**(*value*)
>   >   Format number as percentage.

2.4 Utility Functions (`datascience.util`)

Utility functions

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.

```
>>> make_array(0)
array([0])
>>> make_array(2, 3, 4)
```

```
    array([2, 3, 4])
    >>> make_array("foo", "bar")
    array(['foo', 'bar'],
          dtype='<U3')
    >>> make_array()
    array([], dtype=float64)
```

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

```
    >>> 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`.

    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.**table_apply**(*table*, *func*, *subset=None*)
    Applies a function to each column and returns a Table.

    Uses pandas *apply* under the hood, then converts back to a Table

    Args:

        table [instance of Table] The table to apply your function to

        func [function] Any function that will work with DataFrame.apply

        subset [list | None] A list of columns to apply the function to. If None, function will be applied to all columns in table

    tab [instance of Table] A table with the given function applied. It will either be the shape == shape(table), or shape (1, table.shape[1])

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 either:

1. the minimizing argument of a one-argument function

2. an array of minimizing arguments of a multi-argument function

d

Symbols

A

B

C

D

E

F

G