Contents

Introduction

Lab materials adapted from the UC Berkely Data 6

The objectives for this lab are to, learn more about Python including:

- Array Functions
- Numpy
- Indexing
- Tables
  - `select` and `drop`
  - Adding Columns

- Filtering with `where`
- and more ...

Warmup - Review

Arrays

Suppose we have two arrays containing the resident populations of several states in 2020 and 2021. Assume each array contains information about the same states in the same order.

States	2019 population (in millions)	2020 population (in millions)
Alabama	5.025	5.040
Alaska	0.732	0.733
Arizona	7.178	7.276
Arkansas	3.012	3.026
California	39.500	39.238

```
pop_2020 = make_array(5.025, .732, 7.178, 3.012, 39.500)
pop_2021 = make_array(5.040, .733, 7.276, 3.026, 39.238)
```

Question
How would you compute the percentage change in state populations from 2020 to 2021?

Before thinking of the code, think about the formula that gets translated to code.

$$((final - initial)/initial) * 100$$

Solution

```
100 * (pop_2021 - pop_2020) / pop_2020
```

```
array([ 0.29850746,  0.13661202,  1.36528281,  0.46480744,
    -0.66329114])
```

Array Functions

```
empty_arr = make_array()
int_arr = make_array(3, -4, 0, 5, 2)
str_arr = make_array("cm", "m", "in", "ft", "yd")
```

```
print(empty_arr)
print(int_arr)
print(str_arr)
```

```
[]
[ 3 -4  0  5  2]
['cm' 'min' 'ft' 'yd']
```

Array functions in Python, such as `len`, `min`, `max` and `sum`, are essential tools for manipulating and analyzing arrays or lists of data.

Call expression format	Example(s)
`len(arr)`	`len(str_arr)  # 5` `len(empty_arr)  # 0`
`max(arr)`	`min(int_arr)  # -4`
`min(arr)`	`max(str_arr)  # 'yd'`
`sum(arr)`	`sum(int_arr)  # 6` `sum(str_arr)  # TypeError`

While the **function names** are identical to what we saw for int/float/strs, the call expressions evaluate differently with our new array data type.

Quick Check 1

Recall the definition of an average:

"The average, or mean, of a collection of numbers is the sum of all the elements of the collection, divided by the number of elements in the collection."

How would you compute the average of an array `arr`?

```
arr = make_array(30, -40, -4.5, 0, 35)
avg = ...
avg
```

Confirm with your neighbors

Then, check in the notebook

NumPy

NumPy: A Convenient Function Library

Earlier, we computed averages using built-in Python functions:

```
In [2]: arr = make_array(30, -40, -4.5, 0, 35)
        avg = sum(arr)/len(arr)
        avg
```

```
Out [2]: 4.1
```

Computing averages of array elements happens a lot in data science!

The NumPy package function `np.average()` is human-readable and convenient.

```
In [2]: arr = make_array(30, -40, -4.5, 0, 35)
        avg = np.average(arr)
        avg
```

```
Out [2]: 4.1
```

> NumPy (pronounced "num pie") is a Python package* with convenient and powerful functions for manipulating arrays.
>
> - *For our purposes, "library", "package", and "module" all mean similar things.*

Anytime we want to use NumPy, we run

```
import numpy as np
```

We generally put this import statement at the top of our notebook, then prepend `np.` to call a NumPy function.

```
import python as np

arr = make_array(30, -40, -4.5, 0, 35)
avg = np.average(arr)
avg
```

```
4.1
```

Element-wise NumPy Functions

We'll point you to NumPy functions as they come up; you don't need to memorize them.


```
numbers_arr = make_array(5, 4, 9, 12, 100)
numbers_arr
```

```
array([  5,   4,   9,  12, 100])
```

```
np.sqrt(numbers_arr)
```

```
array([  2.23606798,   2.        ,   3.        ,   3.46410162,  10.
          ])
```

```
np.log(numbers_arr)   # natural log
```

```
array([ 1.60943791,  1.38629436,  2.19722458,  2.48490665,
   4.60517019])
```

```
np.log10(numbers_arr)   # log base 10
```

```
array([ 0.69897   ,  0.60205999,  0.95424251,  1.07918125,  2.
          ])
```

```
np.sin(numbers_arr)
```

```
array([-0.95892427, -0.7568025 ,  0.41211849, -0.53657292,
   -0.50636564])
```

```
np.sqrt(144)
```

```
12.0
```

Many of these functions work on both **arrays** and individual **numbers**.

Common NumPy Functions**

NumPy function	Return value
`np.average(arr)` `np.mean(arr)`	The average (i.e., mean) value of **`arr`**
`np.sum(arr)`	The sum of all elements in **`arr`**
`np.prod(arr)`	The product of all elements in **`arr`**
`np.count_nonzero(arr)`	The number of elements in **`arr`** that are not equal to 0

Side Note: The `datascience` Package

`datascience` **`package`** import statement:

```
from datascience import *
```

- The slightly different syntax allows us to call package functions without prepending datascience.
- The `make_array()` function is from this package!

> The `datascience` package was written by UC Berkeley specifically for data science education. It's designed to support many Python packages like NumPy.

Python References

We reference to the UC Berkeley Data 6 website for Python function references.

There is a link to this and other references and documentation on Canvas - "Python - Resources"

Indexing

Array Methods

Methods are functions that we call with "dot" syntax. There are several array methods that make it easy to calculate values of interest.

Terminology note: Method calls are where the function operates directly on the array `arr`.

```
daily_low_temps = make_array(58, 45, 38, 42, 37, 39)
daily_low_temps
```

```
array([58, 45, 38, 42, 37, 39])
```

```
# Average of all elements
# Equivalent to np.mean(daily_low_temps)
daily_low_temps.mean()
```

```
43.166666666666664
```

```
# Sum of all elements
# Equivalent to np.sum(daily_low_temps)
daily_low_temps.sum()
```

```
259
```

```
# Sum of all elements
# Equivalent to np.prod(daily_low_temps)
daily_low_temps.prod()
```

```
6010903080
```

In these examples, method calls are equivalent to the NumPy package functions.

The most common array method is `item()`, which is used for array indexing.

Array Indexing

When people stand in a line, each person has a position.

Similarly, each element (i.e., value) of an array has a position – called its index.

Python, like most programming languages, is 0-indexed. This means that in an array, the first element has index 0, not 1.

Figure 1: Array Indexing - the first element has index 0.

```
In [4]: int_arr = make_array(3, -4, 0, 5, 2)
        int_arr
Out[4]: array([ 3, -4,  0,  5,  2])
```

Indices 0 1 2 3 4

We can access an element in an array by using its index and the `item()` method:

```
arr.item(index)
```

```
int_arr = make_array(3, -4, 0, 5, 2)
int_arr
```

```
array([ 3, -4,  0,  5,  2])
```

```
int_arr.item(0)
```

```
3
```

```
int_arr.item(3)
```

```
5
```

```
int_arr.item(5)
```

```
---------------------------------------------------------------------------
IndexError                                Traceback (most recent call
    last)
/Users/lebrown/Dropbox/2023c_fall/data1201/data1201-f23-private/lab/
   week10/lab10.ipynb Cell 57 line 1
----> 1 int_arr.item(5)

IndexError: index 5 is out of bounds for axis 0 with size 5
```

Though `int_arr` has 5 elements, the largest valid index is 4.

Negative Indexing

We can also "count backwards" using negative indexes.

- -1 corresponds to the last element in a list.
- -2 corresponds to the second to last element in a list.
- And so on …

```
int_arr.item(len(int_arr) - 1)
```

```
2
```

```
int_arr.item(-1)
```

```
2
```

```
int_arr.item(-3)
```

```
0
```

Creating Arrays of Sequential Values

We can make use of the `np.arange` method to create arrays of sequential values.

There are a number of different ways to call this function:

- `np.arange(stop)` - creates an array from 0 up to but excluding `stop`, that is `[0, stop)`
- `np.arange(start, stop)` - creates an array from `start` up to but excluding `stop`, that is `[start, stop)`
- `np.arange(start, stop, step)` - creates an array from `start` up to but excluding `stop`, with spacing of `step`

```
np.arange(10)
```

```
array([0, 1, 2, 3, 4, 5, 6, 7, 8, 9])
```

```
arr = np.arange(3, 9)
arr
```

```
array([3, 4, 5, 6, 7, 8])
```

```
np.arange(3, 9, 2)
```

```
array([3, 5, 7])
```

Help with a method ?

If you know the name of a method you can use the following notation to get documentation on the function in your notebook.

```
np.arange?
```

```
Docstring:
arange([start,] stop[, step,], dtype=None, *, like=None)

Return evenly spaced values within a given interval.

``arange`` can be called with a varying number of positional arguments:

* ``arange(stop)``: Values are generated within the half-open interval
  ``[0, stop)`` (in other words, the interval including `start` but
  excluding `stop`).
...
```

Quick Check 2

What is the value of `five` after running this code?

```
threes = make_array(3, 6, 9, 12, 15)
five = threes.item(-1) + threes.item(1)
five
```

Confirm with your neighbors

Then, check in the notebook

Move on to lab10.part2.blank.ipynb

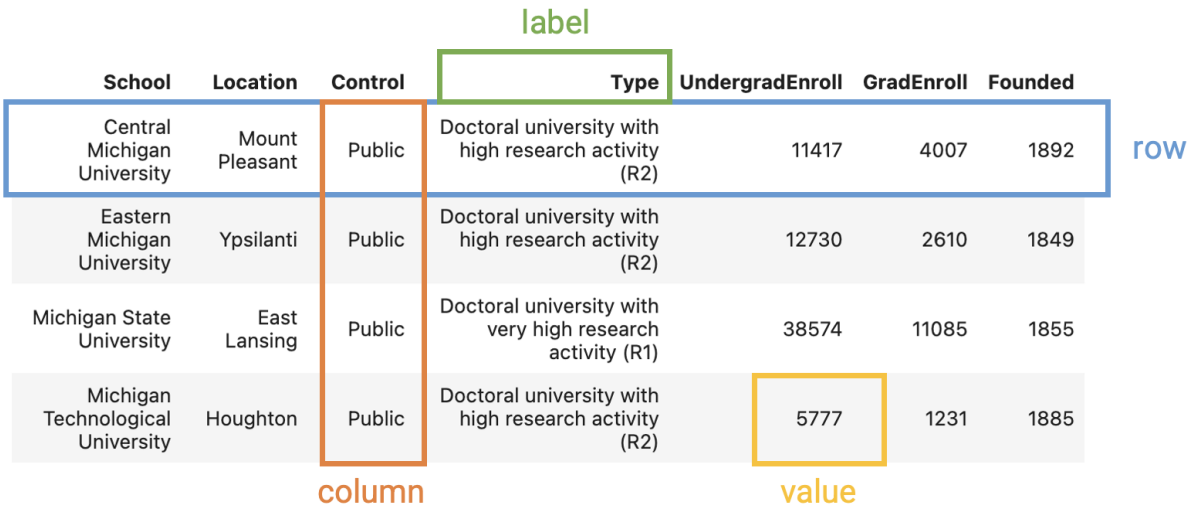
Tables

The Structure of a Table

A `table` is a sequence of labeled columns.

- Each row represents one observation or individual – also known as a "data point".
- Each column represents one attribute.

This lab we will return to the dataset of public four-year colleges and universities in Michigan.



School	Location	Control	Type	UndergradEnroll	GradEnroll	Founded
Central Michigan University	Mount Pleasant	Public	Doctoral university with high research activity (R2)	11417	4007	1892
Eastern Michigan University	Ypsilanti	Public	Doctoral university with high research activity (R2)	12730	2610	1849
Michigan State University	East Lansing	Public	Doctoral university with very high research activity (R1)	38574	11085	1855
Michigan Technological University	Houghton	Public	Doctoral university with high research activity (R2)	5777	1231	1885

Reading in Tables

To interact with tables, we first need to import the `datascience` module.

```
from datascience import *
```

Then, to load a table from a CSV, we call the function `Table.read_table(file_path)`, where `file_path` is a string containing the name and location of the CSV file.

```
schools = Table.read_table('data/michigan_universities.csv')
```

> Always assign your tables to a name!

How big is our table? `.num_rows`, `.num_columns`

One of the first things we may want to do when loading in a table is to determine the number of rows and columns. Assuming `t` is a table:

- `t.num_rows` evaluates to the number of rows in the table.
- `t.num_columns` evaluates to the number of columns in the table.

No parentheses `()` at the end! `num_rows` and `num_columns` are known as **"properties"**.

```
schools.num_rows
```

```
39
```

```
schools.num_columns
```

```
7
```

Columns

> Each **column** in a table is an **array**!

`t.column(label_or_index)` allows us to extract a single column from a table. There are two ways we can use it to access a column:

- By using the column's index.
- By using the column's label (name).

```
# Just run me
some_schools = schools.take(np.arange(6))
some_schools
```

School	Location	Control	Type	UndergradEnroll	GradEnroll	Founded
Central Michigan University	Mount Pleasant	Public	Doctoral university with high research activity (R2)	11417	4007	1892
Eastern Michigan University	Ypsilanti	Public	Doctoral university with high research activity (R2)	12730	2610	1849
Michigan State University	East Lansing	Public	Doctoral university with very high research activity (R1)	38574	11085	1855
Michigan Technological University	Houghton	Public	Doctoral university with high research activity (R2)	5777	1231	1885
Oakland University	Rochester Hills	Public	Doctoral university with high research activity (R2)	13771	3399	1957
University of Michigan	Ann Arbor	Public	Doctoral university with very high research activity (R1)	32282	17996	1817

```
some_schools.column('Location')
```

```
array(['Mount Pleasant', 'Ypsilanti', 'East Lansing', 'Houghton',
       'Rochester Hills', 'Ann Arbor'],
      dtype='<U17')
```

```
some_schools.column(1)
```

```
array(['Mount Pleasant', 'Ypsilanti', 'East Lansing', 'Houghton',
       'Rochester Hills', 'Ann Arbor'],
      dtype='<U17')
```

Quick Check 1

Fill in the blank so that the correct array is returned. Select all that apply.

a. ‘latitude’
b. -2
c. 2
d. 3

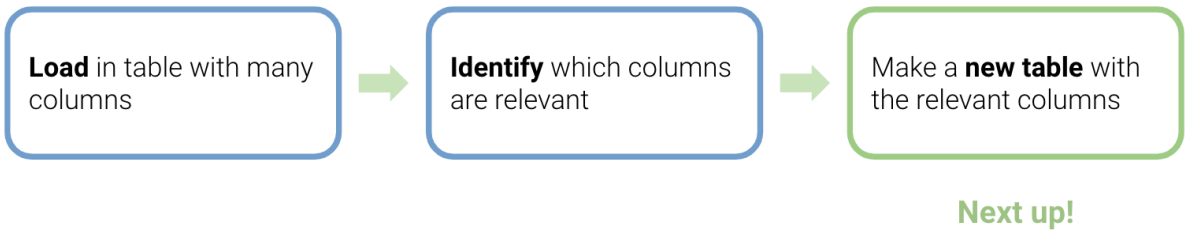
```
In [13]: states.columr(___)
Out[13]: array([ 32.377716,  58.301598,  33.448143,  34.746613,  38.576668,
                 39.739227,  41.764046,  39.157307,  21.307442,  30.438118,
                 33.749027,  43.617775,  39.798363,  39.768623,  41.591087,
                 39.048191,  38.186722,  30.457069,  44.307167,  38.978764,
                 42.358162,  42.733635,  44.955097,  32.303848,  38.579201,
                 46.585709,  40.808075,  39.163914,  43.206898,  40.220596,
                 35.68224 ,  35.78043 ,  46.82085 ,  42.652843,  39.961346,
                 35.492207,  44.938461,  40.264378,  41.830914,  34.000343,
                 44.367031,  36.16581 ,  30.27467 ,  40.777477,  44.262436,
                 37.538857,  47.035805,  38.336246,  43.074684,  41.140259])
```

Confirm with your neighbors

Then, check in the notebook

`select` and `drop`

Here is a common workflow when working with tables of data.



After we identify which columns are relevant, we may want to work with only those columns. We can narrow down our table by selecting the columns we want or dropping the columns we don't want:

- `t.select(n1, n2, ...)` takes in one or more labels (or indices) and **returns a new table** with just those columns
- `t.drop(n1, n2, ...)` takes in one or more labels (or indices) and **returns a new table** without those columns

Both methods return new tables. The original table `t` is not changed!

Consider the table `some_schools` from above.

```
[20]: 1 some_schools
```

[20]:

School	Location	Control	Type	UndergradEnroll	GradEnroll	Founded
Central Michigan University	Mount Pleasant	Public	Doctoral university with high research activity (R2)	11417	4007	1892
Eastern Michigan University	Ypsilanti	Public	Doctoral university with high research activity (R2)	12730	2610	1849
Michigan State University	East Lansing	Public	Doctoral university with very high research activity (R1)	38574	11085	1855
Michigan Technological University	Houghton	Public	Doctoral university with high research activity (R2)	5777	1231	1885
Oakland University	Rochester Hills	Public	Doctoral university with high research activity (R2)	13771	3399	1957
University of Michigan	Ann Arbor	Public	Doctoral university with very high research activity (R1)	32282	17996	1817

We can use the `select` method to create a table with only the `School` and `UndergradEnroll` columns.

[14]:
```
# Select only the columns 'School' and 'UndergradEnroll'
some_schools.select('School', 'UndergradEnroll')
```

[14]:

School	UndergradEnroll
Central Michigan University	11417
Eastern Michigan University	12730
Michigan State University	38574
Michigan Technological University	5777
Oakland University	13771
University of Michigan	32282

How can you create the same table as above, with only the `School` and `UndergradEnroll` columns, but using the `drop` method?

```
# Drop columns so that you are left with only 'School' and '
    UndergradEnroll'
some_schools.drop(...)
```

Solution

[16]:
```
# Drop columns so that you are left with only 'School' and 'UndergradEnroll'
some_schools.drop('Founded', 'Type', 'Control', 'Location', 'GradEnroll')
```

[16]:

School	UndergradEnroll
Central Michigan University	11417
Eastern Michigan University	12730
Michigan State University	38574
Michigan Technological University	5777
Oakland University	13771
University of Michigan	32282

Adding columns

The method `t.with_columns(...)` returns a new table with additional column(s). There are two ways we can call it:

- `t.with_columns(name, values)`, where `name` is a string and `values` is an array.
- `t.with_columns(n1, v1, n2, v2, ...)`, where `n1`, `n2`, … are strings and `v1`, `v2`, … are arrays.

If one of the names overlaps with an existing label, it **replaces** the column.

Add a column with the school's short nicknames.

[17]:
```
# Add a column with the nicknames for each of the six schools
#  (CMU, EMU, MSU, MTU, OU, UofM)
some_schools.with_columns(
    'Nickname', np.array(['CMU', 'EMU', 'MSU', 'MTU', 'OU', 'UofM'])
)
```

[17]:

School	Location	Control	Type	UndergradEnroll	GradEnroll	Founded	Nickname
Central Michigan University	Mount Pleasant	Public	Doctoral university with high research activity (R2)	11417	4007	1892	CMU
Eastern Michigan University	Ypsilanti	Public	Doctoral university with high research activity (R2)	12730	2610	1849	EMU
Michigan State University	East Lansing	Public	Doctoral university with very high research activity (R1)	38574	11085	1855	MSU
Michigan Technological University	Houghton	Public	Doctoral university with high research activity (R2)	5777	1231	1885	MTU
Oakland University	Rochester Hills	Public	Doctoral university with high research activity (R2)	13771	3399	1957	OU
University of Michigan	Ann Arbor	Public	Doctoral university with very high research activity (R1)	32282	17996	1817	UofM

See the notebook to add more than one column at once to a Table.

Creating Tables from Scratch

While we mostly will load in tables from CSV files, we could also create tables from scratch.

- The function `Table()` creates an empty table.
- We can couple `Table()` with `t.with_columns(...)` to add data.

```
states = Table().with_columns(
    'State', np.array(['California', 'New York', 'Florida', 'Texas',
                       'Pennsylvania', 'Michigan']),
    'Code', np.array(['CA', 'NY', 'FL', 'TX', 'PA', 'MI']),
    'Population', np.array([39.3, 19.3, 21.7, 29.3, 12.8, 6.9])
)
states
```

[18]:

State	Code	Population
California	CA	39.3
New York	NY	19.3
Florida	FL	21.7
Texas	TX	29.3
Pennsylvania	PA	12.8
Michigan	MI	6.9

Quick Check 2

Given the table `states` from above, fill in the blanks of the second cell to create the following new table.

State	Code	FedVote
California	CA	D
New York	NY	D
Florida	FL	R
Texas	TX	R
Pennsylvania	PA	D
Michigan	MI	D

```
# Fill in the three blanks to replicate the table above with state's
    federal vote
states._____('Population').with_columns(
    ____, ____
)
```

Confirm with your neighbors

Then, check in the notebook

Filtering with `where`

Filtering Rows

Often times, we will have a table and will want to access only the rows where some condition is true. For example, given our university data, we may want all of the rows where ...

- The undergrad enrollment is above 15000.
- The school name is equal to "Michigan Technological University".
- The school name contains "University".
- The year it was founded is between 1800 and 1900.

Accessing a subset of our data is called filtering. The method `t.where` will help us filter.

Filtering Rows: Exact Match

The method `t.where(label, value)` returns a new table that contains only the rows whose **label** field/attribute has value **value**.

> We will learn more complicated uses of `.where` later, but for now just remember this structure.

Filter the `schools` Table to only those schools that are `Type` has value `R2` classification.

[21]:
```
# Filter the `schools` table to only the schools with R2 research activity
schools.where("Type", "Doctoral university with high research activity (R2)")
```

[21]:

School	Location	Control	Type	UndergradEnroll	GradEnroll	Founded
Central Michigan University	Mount Pleasant	Public	Doctoral university with high research activity (R2)	11417	4007	1892
Eastern Michigan University	Ypsilanti	Public	Doctoral university with high research activity (R2)	12730	2610	1849
Michigan Technological University	Houghton	Public	Doctoral university with high research activity (R2)	5777	1231	1885
Oakland University	Rochester Hills	Public	Doctoral university with high research activity (R2)	13771	3399	1957
Western Michigan University	Kalamazoo	Public	Doctoral university with high research activity (R2)	14587	3679	1903

Filter the `schools` Table to only those schools that are `Control` has value `Public`.

[20]:
```
# Filter the `schools` table to only include Public schools
schools.where("Control", "Public")
```

[20]:

School	Location	Control	Type	UndergradEnroll	GradEnroll	Founded
Central Michigan University	Mount Pleasant	Public	Doctoral university with high research activity (R2)	11417	4007	1892
Eastern Michigan University	Ypsilanti	Public	Doctoral university with high research activity (R2)	12730	2610	1849
Michigan State University	East Lansing	Public	Doctoral university with very high research activity (R1)	38574	11085	1855
Michigan Technological University	Houghton	Public	Doctoral university with high research activity (R2)	5777	1231	1885
Oakland University	Rochester Hills	Public	Doctoral university with high research activity (R2)	13771	3399	1957
University of Michigan	Ann Arbor	Public	Doctoral university with very high research activity (R1)	32282	17996	1817
Wayne State University	Detroit	Public	Doctoral university with very high research activity (R1)	16839	8080	1868
Western Michigan University	Kalamazoo	Public	Doctoral university with high research activity (R2)	14587	3679	1903
Ferris State University	Big Rapids	Public	Doctoral/Professional university (D/PU)	9248	1113	1884
Grand Valley State University	Allendale	Public	Doctoral/Professional university (D/PU)	19379	3027	1960

... (5 rows omitted)

Additional Table Methods

`.show()`

The method `t.show(n)` displays the first `n` rows of `t`. If no argument is provided, all rows are displayed.

> Show does not return anything, it's purely for display purposes. **Never** assign it to a variable!

```
[22]: 1 schools.show(4) # Show the first 4 rows of the `schools` table
```

School	Location	Control	Type	UndergradEnroll	GradEnroll	Founded
Central Michigan University	Mount Pleasant	Public	Doctoral university with high research activity (R2)	11417	4007	1892
Eastern Michigan University	Ypsilanti	Public	Doctoral university with high research activity (R2)	12730	2610	1849
Michigan State University	East Lansing	Public	Doctoral university with very high research activity (R1)	38574	11085	1855
Michigan Technological University	Houghton	Public	Doctoral university with high research activity (R2)	5777	1231	1885

... (35 rows omitted)

`.labels` and `.relabeled()`

The property `t.labels` gives the names of all column names of `t` as a tuple (essentially a list of values).

The method `t.relabeled(old_name, new_name)` returns a new table with the label `old_name` replaced with `new_name`.

The `tbl.labels` property returns a tuple (basically a list) of the labels for each of the columns

```
# The result is a tuple, think of it as a basic list
schools.labels
```

```
('School',
 'Location',
 'Control',
 'Type',
 'UndergradEnroll',
 'GradEnroll',
 'Founded')
```

You can rename the columns with the `relabeled` method.

```
[39]: 1 schools.relabeled('School', 'Name').show(5)
```

Name	Location	Control	Type	UndergradEnroll	GradEnroll	Founded
Central Michigan University	Mount Pleasant	Public	Doctoral university with high research activity (R2)	11417	4007	1892
Eastern Michigan University	Ypsilanti	Public	Doctoral university with high research activity (R2)	12730	2610	1849
Michigan State University	East Lansing	Public	Doctoral university with very high research activity (R1)	38574	11085	1855
Michigan Technological University	Houghton	Public	Doctoral university with high research activity (R2)	5777	1231	1885
Oakland University	Rochester Hills	Public	Doctoral university with high research activity (R2)	13771	3399	1957

... (34 rows omitted)

Summary

Method or Property	Behavior
`Table.read_table(file_path)`	Loads in a table from data.
`t.num_rows`, `t.num_columns`	Returns the number of rows or columns in `t`.
`t.column(label_or_index)`	Returns an array containing the values in the corresponding column of `t`. `label_or_index` can either a string (label) or integer (index).
`t.select(n1, n2, ...)`	Returns a copy of `t` with only the specified columns.
`t.drop(n1, n2, ...)`	Returns a copy of `t` without the specified columns.
`t.with_columns(name, values)` `t.with_columns(n1, v1, n2, v2, ...)`	Returns a copy of `t` with a single additional column. Returns a copy of `t` with multiple additional columns.
`t.show(n)`	Displays the first `n` rows of `t`. Does not return anything!
`t.labels`	Returns a tuple of `t`'s column names.
`t.relabeled(old_name, new_name)`	Returns a copy of `t` with the label `old_name` replaced with `new_name`.