CMPU 100 · Programming with Data

Tables

Class 6

Where are we?

Here are some data that can be represented with what we've seen so far:

Here are some data that can be represented with what we've seen so far:

The population of NYC	*Integer*

Here are some data that can be represented with what we've seen so far:

The population of NYC	*Integer*
Average body temperature	*Floating-point number*

Here are some data that can be represented with what we've seen so far:

The population of NYC	*Integer*
Average body temperature	*Floating-point number*
Whether or not I ate breakfast this morning	*Boolean*

Here are some data that can be represented with what we've seen so far:

The population of NYC	*Integer*
Average body temperature	*Floating-point number*
Whether or not I ate breakfast this morning	*Boolean*
The complete text of *Beowulf*	*String*

Here are some data that can be represented with what we've seen so far:

The population of NYC	*Integer*
Average body temperature	*Floating-point number*
Whether or not I ate breakfast this morning	*Boolean*
The complete text of *Beowulf*	*String*
The average temperature each month	*List* or *array*

What if we wanted to write a program to look up the population of any town in New York?

We can consider the last two census years – 2010 and 2020.

```
def population(municipality: str, year: int) -> int:
    """Return population of the municipality for the given year"""
    if municipality == "New York":
        if year == 2010:
            return 8175133
        elif year == 2020:
            return 8804190
        else:
            raise Exception("Bad year")
    elif municipality == "Poughkeepsie":
        if year == 2010:
            return 43341
        elif year == 2020:
            return 45471
        else:
            raise Exception("Bad year")
    else:
        raise Exception("Bad municipality")
```

```
def population(municipality: str, year: int) -> int:
    """Return population of the municipality for the given year"""
    if municipality == "New York":
        if year == 2010:
            return 8175133
        elif year == 2020:
            return 8804190
        else:
            raise Exception("Bad year")
    elif municipality == "Poughkeepsie":
        if year == 2010:
            return 43341
        elif year == 2020:
            return 45471
        else:
            raise Exception("Bad year")
    else:
        raise Exception("Bad municipality")
```

We can nest `if` expressions!

```
def population(municipality: str, year: int) -> int:
    """Return population of the municipality for the given year"""
    if municipality == "New York":
        if year == 2010:
            return 8175133
        elif year == 2020:
            return 8804190
        else:
            raise Exception("Bad year")
    elif municipality == "Poughkeepsie":
        if year == 2010:
            return 43341
        elif year == 2020:
            return 45471
        else:
            raise Exception("Bad year")
    else:
        raise Exception("Bad municipality")
```

```
def population(municipality: str, year: int) -> int:
    """Return population of the municipality for the given year"""
    if municipality == "New York":
        if year == 2010:
            return 8175133
        elif year == 2020:
            return 8804190
        else:
            raise Exception("Bad year")
    elif municipality == "Poughkeepsie":
        if year == 2010:
            return 43341
        elif year == 2020:
            return 45471
        else:
            raise Exception("Bad year")
    else:
        raise Exception("Bad municipality")
```

Report an error that prevents the function from returning an answer

```
def population(municipality: str, year: int) -> int:
    """Return population of the municipality for the given year"""
    if municipality == "New York":
        if year == 2010:
            return 8175133
        elif year == 2020:
            return 8804190
        else:
            raise Exception("Bad year")
    elif municipality == "Poughkeepsie":
        if year == 2010:
            return 43341
        elif year == 2020:
            return 45471
        else:
            raise Exception("Bad year")
    else:
        raise Exception("Bad municipality")
```

```
def population(municipality: str, year: int) -> int:
    """Return population of the municipality for the given year"""
    if municipality == "New York":
        if year == 2010:
            return 8175133
        elif year == 2020:
            return 8804190
        else:
            raise Exception("Bad year")
    elif municipality == "Poughkeepsie":
        if year == 2010:
            return 43341
        elif year == 2020:
            return 45471
        else:
            raise Exception("Bad year")
    else:
        raise Exception("Bad municipality")
```

This isn't a great way to do this.

Why not?

```
def population(municipality: str, year: int) -> int:
    """Return population of the municipality for the given year"""
    if municipality == "New York":
        if year == 2010:
            return 8175133
        elif year == 2020:
            return 8804190
        else:
            raise Exception("Bad year")
    elif municipality == "Poughkeepsie":
        if year == 2010:
            return 43341
        elif year == 2020:
            return 45471
        else:
            raise Exception("Bad year")
    else:
        raise Exception("Bad municipality")
```

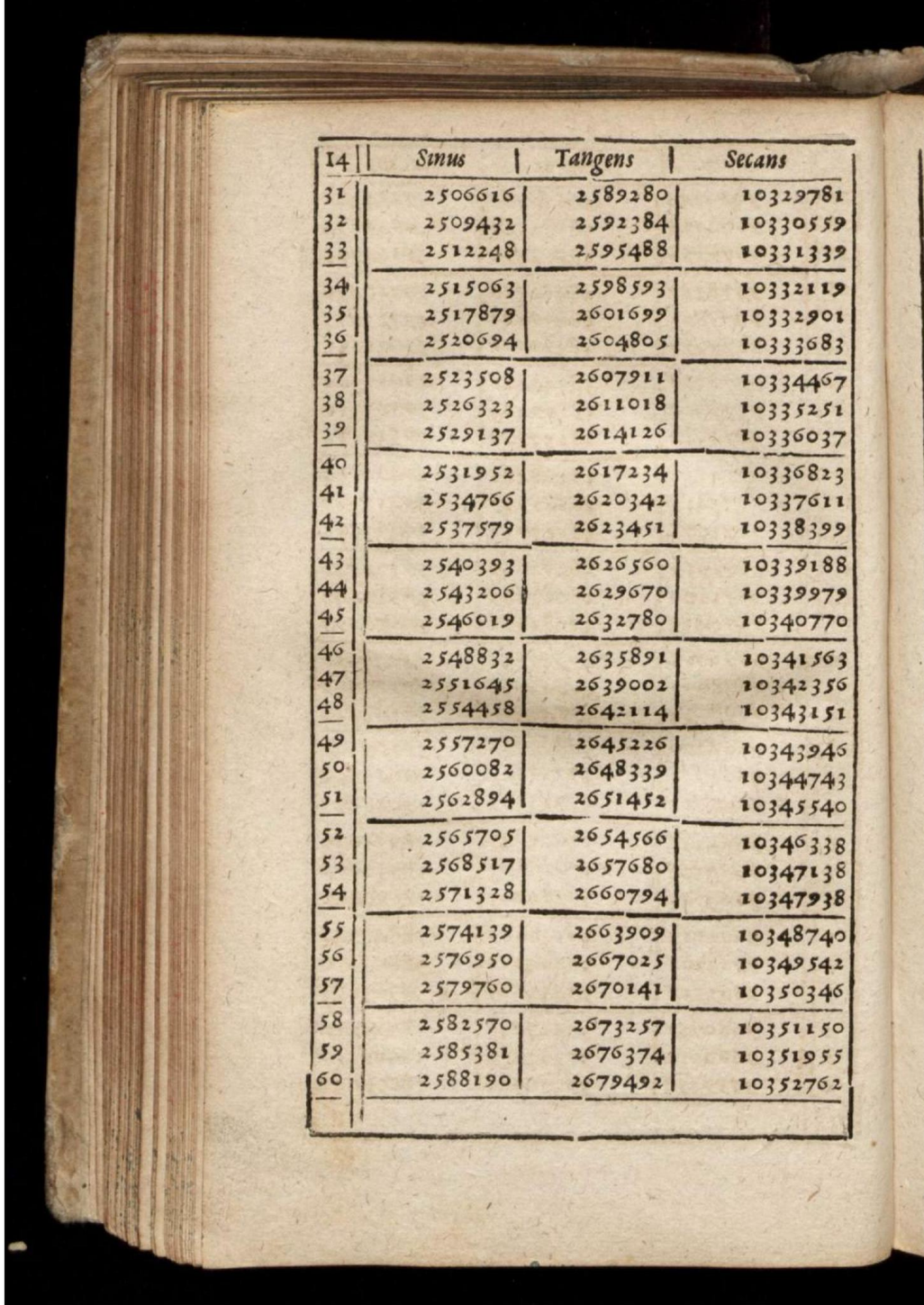
What about the rest of the state?

```
def population(municipality: str, year: int) -> int:
    """Return population of the municipality for the given year"""
    if municipality == "New York":
        if year == 2010:
            return 8175133
        elif year == 2020:
            return 8804190
        else:
            raise Exception("Bad year")
    elif municipality == "Poughkeepsie":
        if year == 2010:
            return 43341
        elif year == 2020:
            return 45471
        else:
            raise Exception("Bad year")
    else:
        raise Exception("Bad municipality")
```

KEY IDEA Separate data from computation.

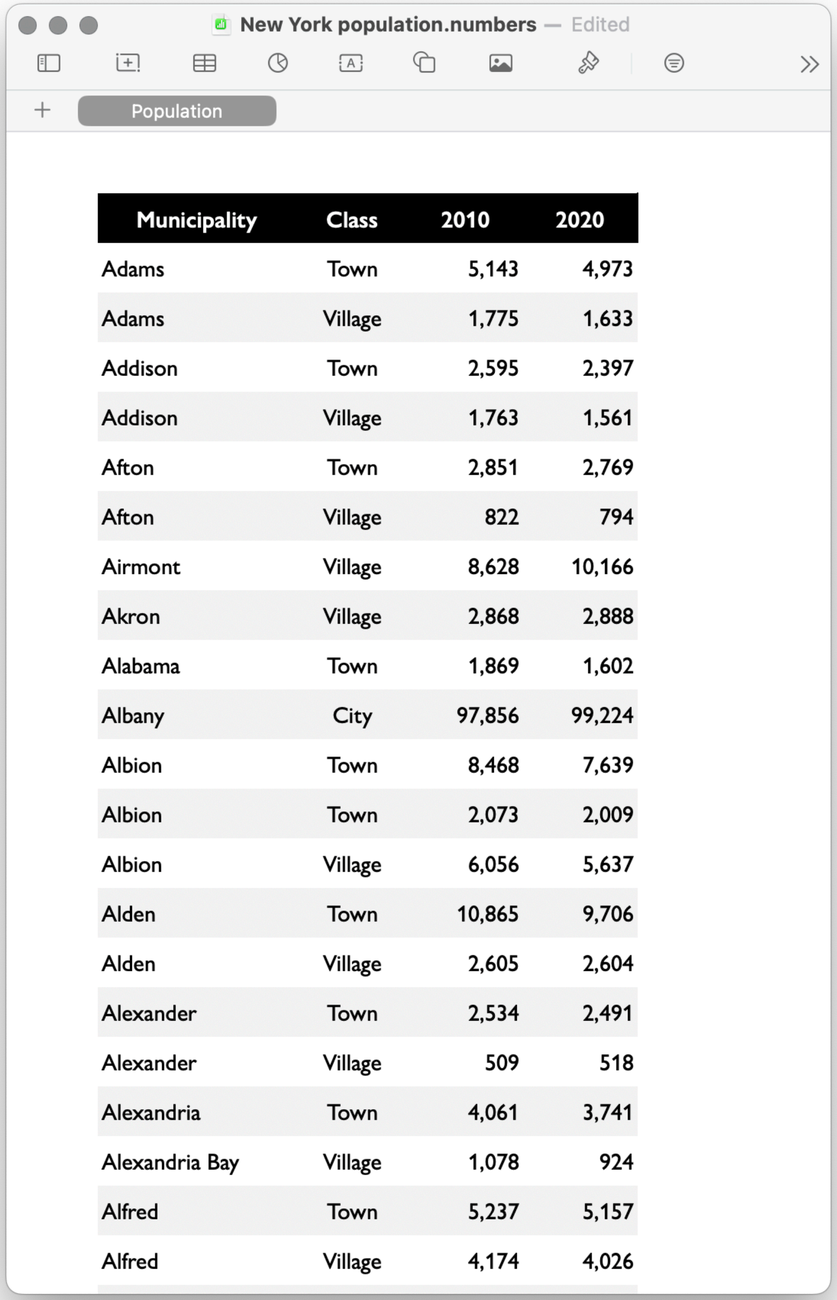
Tables

Tables are used for tabular data, like you might find printed in a book or in a spreadsheet on a computer.



14	Sinus	Tangens	Secans
31	2506616	2589280	10329781
32	2509432	2592384	10330559
33	2512248	2595488	10331339
34	2515063	2598593	10332119
35	2517879	2601699	10332901
36	2520694	2604805	10333683
37	2523508	2607911	10334467
38	2526323	2611018	10335251
39	2529137	2614126	10336037
40	2531952	2617234	10336823
41	2534766	2620342	10337611
42	2537579	2623451	10338399
43	2540393	2626560	10339188
44	2543206	2629670	10339979
45	2546019	2632780	10340770
46	2548832	2635891	10341563
47	2551645	2639002	10342356
48	2554458	2642114	10343151
49	2557270	2645226	10343946
50	2560082	2648339	10344743
51	2562894	2651452	10345540
52	2565705	2654566	10346338
53	2568517	2657680	10347138
54	2571328	2660794	10347938
55	2574139	2663909	10348740
56	2576950	2667025	10349542
57	2579760	2670141	10350346
58	2582570	2673257	10351150
59	2585381	2676374	10351955
60	2588190	2679492	10352762

Tables are used for tabular data, like you might find printed in a book or in a spreadsheet on a computer.



Municipality	Class	2010	2020
Adams	Town	5,143	4,973
Adams	Village	1,775	1,633
Addison	Town	2,595	2,397
Addison	Village	1,763	1,561
Afton	Town	2,851	2,769
Afton	Village	822	794
Airmont	Village	8,628	10,166
Akron	Village	2,868	2,888
Alabama	Town	1,869	1,602
Albany	City	97,856	99,224
Albion	Town	8,468	7,639
Albion	Town	2,073	2,009
Albion	Village	6,056	5,637
Alden	Town	10,865	9,706
Alden	Village	2,605	2,604
Alexander	Town	2,534	2,491
Alexander	Village	509	518
Alexandria	Town	4,061	3,741
Alexandria Bay	Village	1,078	924
Alfred	Town	5,237	5,157
Alfred	Village	4,174	4,026

It's common to share tabular data as a *CSV file*:

```
name,kind,pop2010,pop2020
Adams,Town,5143,4973
Adams,Village,1775,1633
Addison,Town,2595,2397
Addison,Village,1763,1561
Afton,Town,2851,2769
Afton,Village,822,794
Airmont,Village,8628,10166
Akron,Village,2868,2888
Alabama,Town,1869,1602
Albany,City,97856,99224
...
```

It's common to share tabular data as a *CSV file*:

```
name,kind,pop2010,pop2020
Adams,Town,5143,4973
Adams,Village,1775,1633
Addison,Town,2595,2397
Addison,Village,1763,1561
Afton,Town,2851,2769
Afton,Village,822,794
Airmont,Village,8628,10166
Akron,Village,2868,2888
Alabama,Town,1869,1602
Albany,City,97856,99224
...
```

It's common to share tabular data as a *CSV file*:

```
name     , kind     , pop2010 , pop2020
Adams    , Town     ,    5143 ,    4973
Adams    , Village  ,    1775 ,    1633
Addison  , Town     ,    2595 ,    2397
Addison  , Village  ,    1763 ,    1561
Afton    , Town     ,    2851 ,    2769
Afton    , Village  ,     822 ,     794
Airmont  , Village  ,    8628 ,   10166
Akron    , Village  ,    2868 ,    2888
Alabama  , Town     ,    1869 ,    1602
Albany   , City     ,   97856 ,   99224
...
```

Comma-separated values

In Python, we can use the **`datascience`** library to load a CSV file as a table that we can easily work with:

```
from datascience import *
```

In Python, we can use the **`datascience`** library to load a CSV file as a table that we can easily work with:

```
from datascience import *
```

This says, "Let us use every name defined by the **`datascience`** *module"*

In Python, we can use the **`datascience`** library to load a CSV file as a table that we can easily work with:

```
from datascience import *

url = "https://www.cs.vassar.edu/~cs100/data/
municipalities.csv"

municipalities = Table.read_table(url)
```

`Table` *is a class, which is a way of grouping related data and functions*

In Python, we can use the **`datascience`** library to load a CSV file as a table that we can easily work with:

```
from datascience import *

url = "https://www.cs.vassar.edu/~cs100/data/
municipalities.csv"

municipalities = Table.read_table(url)
```

This dot notation just means "use the **`read_table`** *function from the* **`Table`** *class" rather than one defined anywhere else.*

[3]
```
municipalities
```

name	kind	pop2010	pop2020
Adams	Town	5143	4973
Adams	Village	1775	1633
Addison	Town	2595	2397
Addison	Village	1763	1561
Afton	Town	2851	2769
Afton	Village	822	794
Airmont	Village	8628	10166
Akron	Village	2868	2888
Alabama	Town	1869	1602
Albany	City	97856	99224

... (1517 rows omitted)

```
[3] municipalities
```

name	kind	pop2010	pop2020
Adams	Town	5143	4973
Adams	Village	1775	1633
Addison	Town	2595	2397
Addison	Village	1763	1561
Afton	Town	2851	2769
Afton	Village	822	794
Airmont	Village	8628	10166
Akron	Village	2868	2888
Alabama	Town	1869	1602
Albany	City	97856	99224

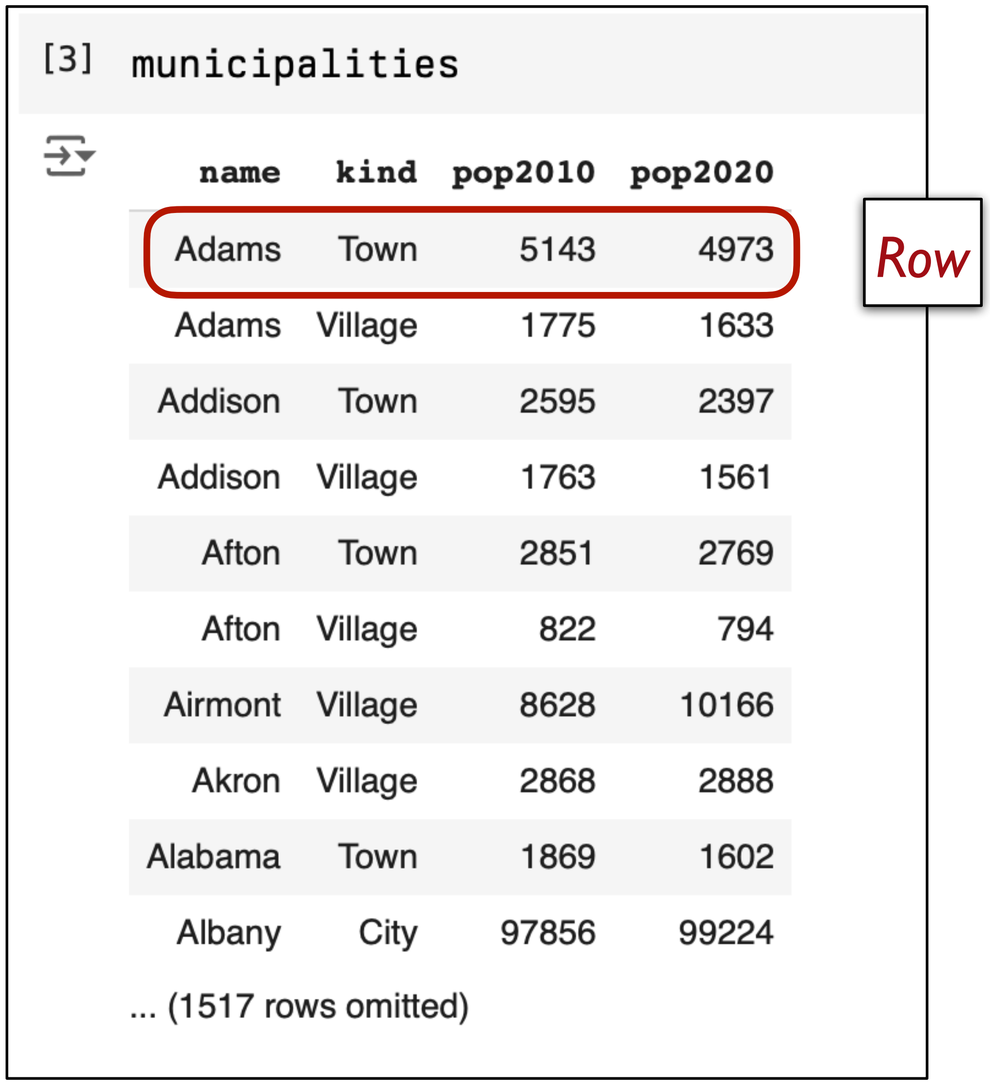
... (1517 rows omitted)

Column

```
[3] municipalities
```

name	kind	pop2010	pop2020
Adams	Town	5143	4973
Adams	Village	1775	1633
Addison	Town	2595	2397
Addison	Village	1763	1561
Afton	Town	2851	2769
Afton	Village	822	794
Airmont	Village	8628	10166
Akron	Village	2868	2888
Alabama	Town	1869	1602
Albany	City	97856	99224

... (1517 rows omitted)



```
[3] municipalities
```

name	kind	pop2010	pop2020
Adams	Town	5143	4973
Adams	Village	1775	1633
Addison	Town	2595	2397
Addison	Village	1763	1561
Afton	Town	2851	2769
Afton	Village	822	794
Airmont	Village	8628	10166
Akron	Village	2868	2888
Alabama	Town	1869	1602
Albany	City	97856	99224

... (1517 rows omitted)

Label

[3] `municipalities`

name	kind	pop2010	pop2020
Adams	Town	5143	4973
Adams	Village	1775	1633
Addison	Town	2595	2397
Addison	Village	1763	1561
Afton	Town	2851	2769
Afton	Village	822	794
Airmont	Village	8628	10166
Akron	Village	2868	2888
Alabama	Town	1869	1602
Albany	City	97856	99224

... (1517 rows omitted)

Value

Variable

```
[3] municipalities
```

name	kind	pop2010	pop2020
Adams	Town	5143	4973
Adams	Village	1775	1633
Addison	Town	2595	2397
Addison	Village	1763	1561
Afton	Town	2851	2769
Afton	Village	822	794
Airmont	Village	8628	10166
Akron	Village	2868	2888
Alabama	Town	1869	1602
Albany	City	97856	99224

... (1517 rows omitted)

Observation

Now that we have the data in Python, we can write programs to answer questions.

Filtering and ordering tables

Oftentimes, we will have a table and want to access only the rows where some condition is true.

For example, we might want to get a version of the table that only has cities where the population has decreased.

```
def filter_population_decreased(t: Table) -> Table:
    """Filter the table to only keep rows where the
    population decreased between 2010 and 2020.
    """
    if population_decreased(t.row(0)):
        ...  # Keep row 0
        if population_decreased(t.row(1)):
            ...  # Keep row 1
        else:
            ...  # Don't keep row 1
    else:
        ...  # Don't keep row 0
```

Remember how we used **`filter`** last week:

```
list(filter(is_odd, [1, 2, 3, 4, 5]))
```

→ `[1, 3, 5]`

It would be nice if we had something like this for tables!

Instead, we can filter a Table by asking for a new Table with only the rows **`where`** something is true about the value in a column, e.g.,

```
municipalities.where("kind", are.equal_to("Town"))
```

Instead, we can filter a Table by asking for a new Table with only the rows **`where`** something is true about the value in a column, e.g.,

```
municipalities.where("kind", are.equal_to("Town"))
```

What's this?

The **datascience** library gives us a convenient way to create a simple predicate functions. These begin with the prefix **are**.

So, we can write

```
is_town = are.equal_to("Town")
```

And now **is_town** is a function that we can call:

```
is_town("Town") → True
is_town("Aardvark") → False
```

There's nothing special about using **`are`** to make functions for filtering tables.

We could just define the function ourselves:

```
def is_town(s: str) -> bool:
    return s == "Town"
```

and pass that to **`where()`**:

```
municipalities.where("kind", is_town)
```

But we'll use **`are`** to make our predicates because it's quicker to write. Here are some of the predicate functions we can make with **`are`**:

Predicate	*Behavior*
`are.equal_to(z)`	Is the value from the column equal to *z*?
`are.above(x)`	Is the value from the column above *z*?
`are.below(x)`	Is the value from the column below *z*?
`are.between(x, y)`	Is the value from the column between *x* (inclusive) and *y* (exclusive)?
`are.containing(s)`	Does the value from the column contain the string *s*?
`are.contained_in(s)`	Is the value from the column inside the string/array *s*?

Add `not_` *to any of the above to negate the predicate, e.g.,* `are.not_equal_to(z)`

Where we get a real benefit from **`are`** is if we want to filter the table based on the values in *multiple* columns.

For example, if we want rows where the population decreased between 2020 and 2010, we can do that:

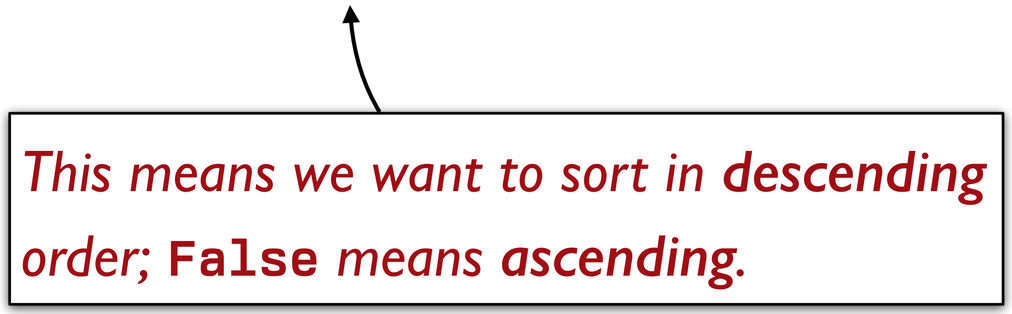
```
municipalities.where(
    "pop2020",
    are.below,
    "pop2010"
)
```

We can also order the data by the values in one column:

```
municipalities.sort("pop2020", True)
```

We can also order the data by the values in one column:

```
municipalities.sort("pop2020", True)
```



And we can combine all of these operations.

How would we get the village with the smallest population?

Example: Population change

PROBLEM Figure out what towns in New York are grew the most.

Acknowledgments

This class incorporates material from:

Kathi Fisler and Doug Woos, Brown University

Data 6, University of California, Berkeley (CC BY-NC-SA)