CMPU 101 § 51 · Computer Science I

Working with Tables

14 September 2022

Lab 2

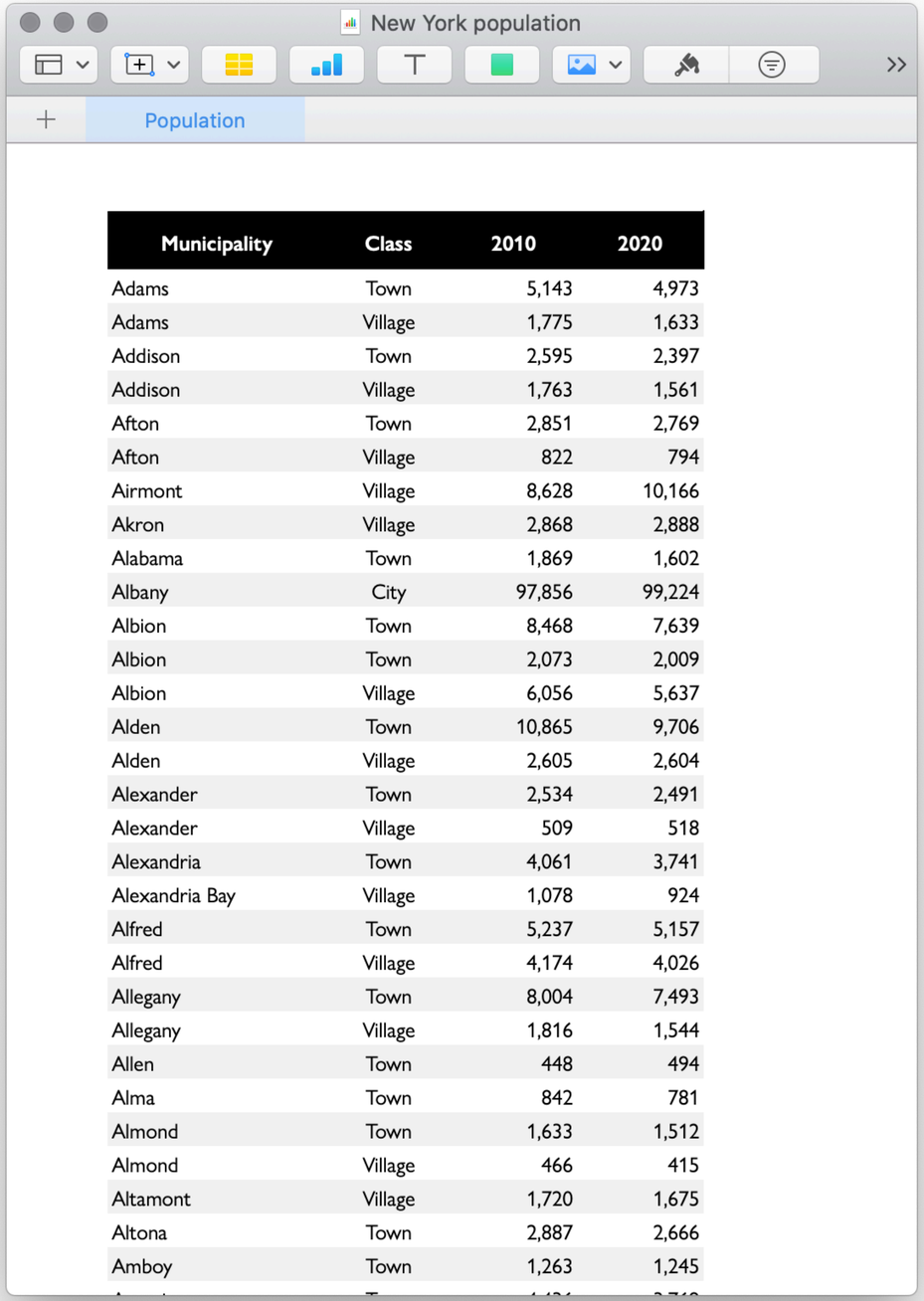
Due Friday

Assignment 2

Due Wednesday

Where are we?

Lots of real-world data is naturally represented as tables.



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
Allegany	Town	8,004	7,493
Allegany	Village	1,816	1,544
Allen	Town	448	494
Alma	Town	842	781
Almond	Town	1,633	1,512
Almond	Village	466	415
Altamont	Village	1,720	1,675
Altona	Town	2,887	2,666
Amboy	Town	1,263	1,245

Lots of real-world data is naturally represented as tables.

```
municipalities =
  table: name, kind, pop-2010, pop-2020
    row: "Adams", "Town", 5143, 4973
    row: "Adams", "Village", 1775, 1633
    row: "Addison", "Town", 2595, 2397
    row: "Addison", "Village", 1763, 1561
    row: "Afton", "Town", 2851, 2769
    ...
  end
```

Lots of real-world data is naturally represented as tables.

```
>>> municipalities
```

name	kind	pop-2010	pop-2020
"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

Click to show the remaining 1517 rows...

Recap: Accessing parts of a table

To get a particular row from a table, we use its numeric index *n*, counting from 0:

⟨***table***⟩`.row-n(0)`

```
>>> municipalities
```

name	kind	pop-2010	pop-2020
"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

Click to show the remaining 1517 rows...

```
>>> municipalities.row-n(0)
```

"name"	"Adams"	"kind"	"Town"	"pop-2010"	5143	"pop-2020"	4973

```
>>> municipalities
```

name	kind	pop-2010	pop-2020
"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

Click to show the remaining 1517 rows...

```
>>> municipalities.row-n(1)
```

"name"	"Adams"	"kind"	"Village"	"pop-2010"	1775	"pop-2020"	1633

```
>>> municipalities
```

name	kind	pop-2010	pop-2020
"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

Click to show the remaining 1517 rows...

```
>>> municipalities.row-n(2)
```

"name"	"Addison"	"kind"	"Town"	"pop-2010"	2595	"pop-2020"	2397

To get a particular column's value from a row, we specify the column name using square brackets:

```
⟨row⟩["column name"]
```

```
>>> municipalities.row-n(0)
```

"name"	"Adams"	"kind"	"Town"	"pop-2010"	5143	"pop-2020"	4973

```
>>> municipalities.row-n(0)
```

"name"	"Adams"	"kind"	"Town"	"pop-2010"	5143	"pop-2020"	4973

```
>>> municipalities.row-n(0)["name"]
"Adams"

>>> municipalities.row-n(0)["pop-2020"]
4973
```

Recap: Ordering tables

To do more with tabular data, first include the textbook library:

```
include shared-gdrive("dcic-2021",
  "1wyQZj_L0qqV9Ekgr9au6RX2iqt2Ga8Ep")
```

We can transform tabular data to get a particular view. E.g., to order the rows from the highest 2010 population to the lowest:

```
››› order-by(municipalities, "pop-2010", false)
```

name	kind	pop-2010	pop-2020
"New York"	"City"	8175133	8804190
"Hempstead"	"Town"	759757	793409
"Brookhaven"	"Town"	486040	485773
"Islip"	"Town"	335543	339938
"Oyster Bay"	"Town"	293214	301332
"Buffalo"	"City"	261310	278349
"North Hempstead"	"Town"	226322	237639

We can transform tabular data to get a particular view. E.g., to order the rows from the *lowest* 2010 population to the *highest*:

```
››› order-by(municipalities, "pop-2010", true)
```

name	kind	pop-2010	pop-2020
"Dering Harbor"	"Village"	11	50
"Saltaire"	"Village"	37	113
"Red House"	"Town"	38	27
"West Hampton Dunes"	"Village"	55	126
"Montague"	"Town"	78	97
"Ocean Beach"	"Village"	79	153
"Morehouse"	"Town"	86	92

```
>>> municipalities.row-n(0)
```

```
"name" "Adams" "kind" "Town" "pop-2010" 5143 "pop-2020" 4973
```

```
>>> order-by(municipalities, "pop-2010", false).row-n(0)
```

```
"name" "New York" "kind" "City" "pop-2010" 8175133 "pop-2020" 8804190
```

This makes it easy to get the row for the municipality with the highest 2010 population.

```
>>> municipalities.row-n(0)
```

```
"name" "Adams" "kind" "Town" "pop-2010" 5143 "pop-2020" 4973
```

```
>>> order-by(municipalities, "pop-2010", false).row-n(0)
```

```
"name" "New York" "kind" "City" "pop-2010" 8175133 "pop-2020" 8804190
```

```
>>> ordered = order-by(municipalities, "pop-2010", false)
>>> ordered.row-n(0)
```

```
"name" "New York" "kind" "City" "pop-2010" 8175133 "pop-2020" 8804190
```

Or, to make it more readable, we can split the computation into parts, using names

So, to see the 2010 population for the biggest municipality, we could write the computation like this.

```
››› ordered = order-by(municipalities, "pop-2010", false)
››› biggest = ordered.row-n(0)
››› biggest["pop-2010"]
8175133
```

Recap: Filtering tables

Make a table keeping only those municipalities with a 2010 population over 10,000:

```
fun big-muni(r :: Row) -> Boolean:
  doc: "Return true if the municipality had over
10,000 people had in 2010"
  r["pop-2010"] > 10000
end
```

```
››› filter-with(municipalities, big-muni)
```

name	kind	pop-2010	pop-2020
"Albany"	"City"	97856	99224
"Alden"	"Town"	10865	9706
"Amherst"	"Town"	122366	129595

Building columns

At the end of last class, we saw that we can also have functions on rows that don't return Booleans:

```
fun percent-change(r :: Row) -> Number:
  doc: "Compute the percentage change for the
population of the given municipality between 2010
and 2020"
  (r["pop-2020"] - r["pop-2010"]) /
  r["pop-2010"]
end
```

And we can use such functions to compute the values for a new column:

```
build-column(municipalities, "percent-change",
  percent-change)
```

And we can use such functions to compute the values for a new column:

```
build-column(municipalities, "percent-change",
  percent-change)
```

Name of the new column

And we can use such functions to compute the values for a new column:

```
build-column(municipalities, "percent-change",
  percent-change)
```

Name of the new column

Name of the function to use

So, if we have this table, **t**,

a	b
"dog"	2
"cat"	3

then the result of calling `build-column(t, "c", builder)` is:

a	b	c
"dog"	2	builder(<"dog", 2>)
"cat"	3	builder(<"cat", 3>)

For example, if we have

```
fun builder(r :: Row) -> Number:
  string-length(row["a"]) + row["b"]
end
```

Then we end up with the following table:

a	b	c
"dog"	2	5
"cat"	3	6

The values that the builder function returns will be the values in the new column we're adding to each row.

```
build-column ::
  (t :: Table,
   colname :: String,
   builder :: (Row -> A))
  -> Table
```

```
build-column ::
   (t :: Table,
    colname :: String,
    builder :: (Row -> A))
   -> Table
```

This is the second time we've seen a function that takes a function as one of its inputs!

Both **`filter-with`** and **`build-column`** need a helper function that tells them *how* to do what we want.

Just as a function is an abstraction over specific computations, **`filter-with`** and **`build-column`** are abstractions over more specific functions.

They provide the common functionality and the arguments we give provide the specifics.

Interlude: Functional programming

We can

- sort the rows a table with **`order-with`**,
- select certain rows using **`filter-with`**, and
- add a new column of values with **`build-column`**

but none of these functions change the original table!

Just as the expression **2 + 3** doesn't change the value of **2** or of **3**, functions that take a table as input don't change the original table.

Instead, they return a *new* table.

This is a paradigm called *functional programming*.

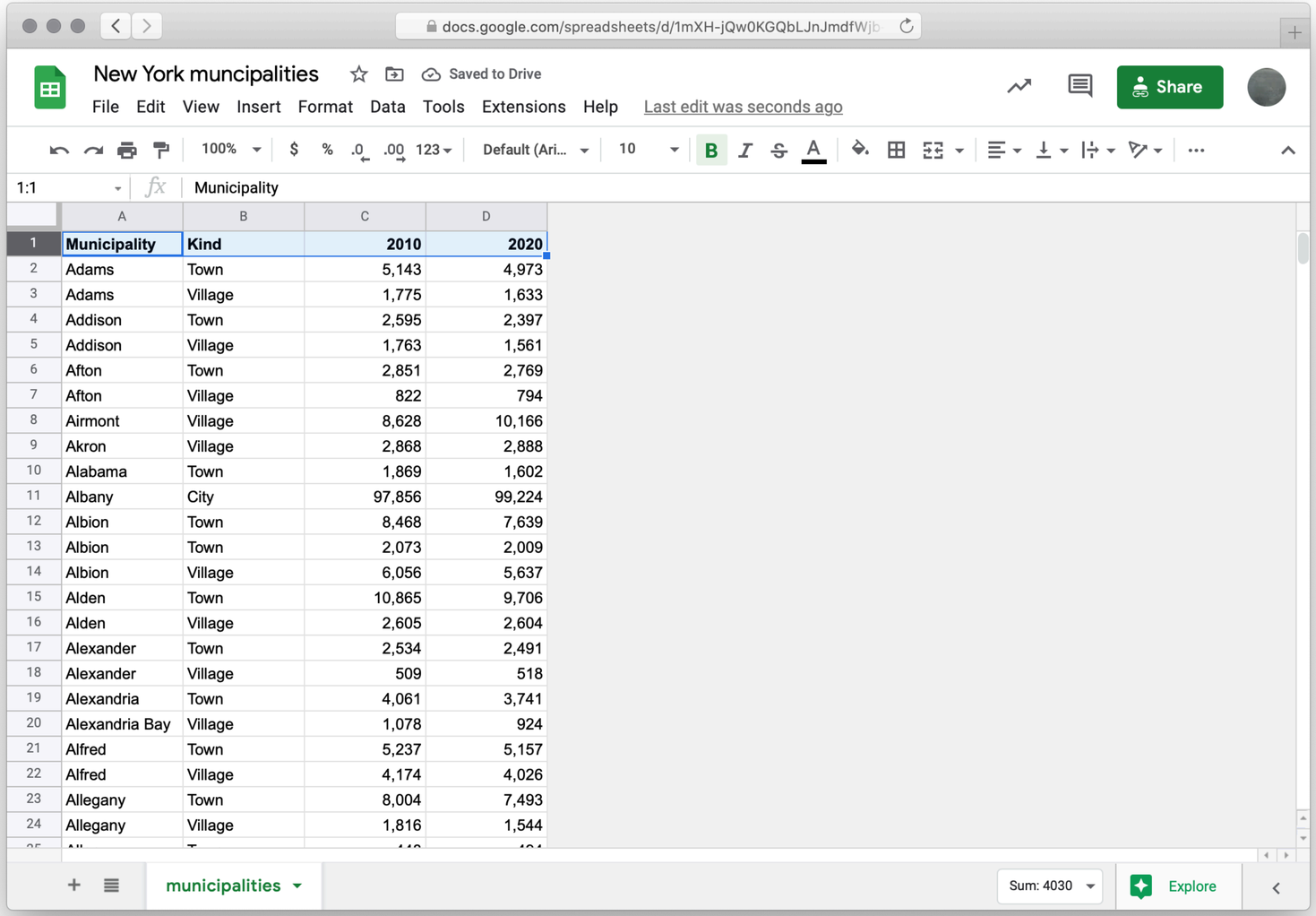
If you have experience working in other languages, this may seem strange, but it can be extremely useful!

We'll explore the idea of functional programming more in the coming weeks.

Loading Google Sheets into Pyret

We've seen that it's inconvenient to type a large table into a Pyret program. Last time, we loaded the municipalities table from a separate Pyret file that I prepared ahead of time.

More often, we'll want to load our data from outside of Pyret.



	A	B	C	D
1	**Municipality**	**Kind**	**2010**	**2020**
2	Adams	Town	5,143	4,973
3	Adams	Village	1,775	1,633
4	Addison	Town	2,595	2,397
5	Addison	Village	1,763	1,561
6	Afton	Town	2,851	2,769
7	Afton	Village	822	794
8	Airmont	Village	8,628	10,166
9	Akron	Village	2,868	2,888
10	Alabama	Town	1,869	1,602
11	Albany	City	97,856	99,224
12	Albion	Town	8,468	7,639
13	Albion	Town	2,073	2,009
14	Albion	Village	6,056	5,637
15	Alden	Town	10,865	9,706
16	Alden	Village	2,605	2,604
17	Alexander	Town	2,534	2,491
18	Alexander	Village	509	518
19	Alexandria	Town	4,061	3,741
20	Alexandria Bay	Village	1,078	924
21	Alfred	Town	5,237	5,157
22	Alfred	Village	4,174	4,026
23	Allegany	Town	8,004	7,493
24	Allegany	Village	1,816	1,544

```
include gdrive-sheets

# The ID of the Google Sheets file, which appears
# in the URL
ssid = "1yZ-TeVJbmMyOGzVVI3FWxRS8Sd6uu-rrB5b-WIEdRAY"

spreadsheet = load-spreadsheet(ssid)
```

A spreadsheet might have more than one sheet (the tabs at the bottom of Google Sheets). But, in this case, we just have one:

```
>>> spreadsheet
spreadsheet("municipalities")
```

To load a table from a spreadsheet, we need to tell Pyret which sheet to load it from and what we want the columns to be called (which can be different from what is in the spreadsheet):

```
municipalities =
  load-table:
    name :: String, kind :: String,
    pop-2010 :: Number, pop-2020 :: Number
    source:
      spreadsheet.sheet-by-name("municipalities",
        true)
  end
```

To load a table from a spreadsheet, we need to tell Pyret which sheet to load it from and what we want the columns to be called (which can be different from what is in the spreadsheet):

```
municipalities =
  load-table:
    name :: String, kind :: String,
    pop-2010 :: Number, pop-2020 :: Number
    source:
      spreadsheet.sheet-by-name("municipalities",
        true)
  end
```

This means there's a header row that Pyret should skip

Using our table loaded from Google Sheets, let's revisit our code from last class for finding the fastest growing towns.

```
fun is-town(r :: Row) -> Boolean:
  doc: "Check if a row is for a town"
  r["kind"] == "Town"
end

fun percent-change(r :: Row) -> Number:
  doc: "Compute the percentage change for the population of the
given municipality between 2010 and 2020"
  (r["pop-2020"] - r["pop-2010"]) / r["pop-2010"]
end

towns = filter-with(municipalities, is-town)

towns-with-percent-change =
  build-column(towns, "percent-change", percent-change)

fastest-growing-towns =
  order-by(towns-with-percent-change,
    "percent-change", false)

fastest-growing-towns
```

```
fun is-town(r :: Row) -> Boolean:
  doc: "Check if a row is for a town"
  r["kind"] == "Town"
end

fun percent-change(r :: Row) -> Number:
  doc: "Compute the percentage change for the population of the
given municipality between 2010 and 2020"
  (r["pop-2020"] - r["pop-2010"]) / r["pop-2010"]
end

towns = filter-with(municipalities, is-town)

towns-with-percent-change =
  build-column(towns, "percent-change", percent-change)

fastest-growing-towns =
  order-by(towns-with-percent-change,
    "percent-change", false)

fastest-growing-towns
```

Let's take these loose expressions and put them in a function!

```
fun is-town(r :: Row) -> Boolean:
  doc: "Check if a row is for a town"
  r["kind"] == "Town"
end

fun percent-change(r :: Row) -> Number:
  doc: "Compute the percentage change for the population of the
given municipality between 2010 and 2020"
  (r["pop-2020"] - r["pop-2010"]) / r["pop-2010"]
end

fun fastest-growing-towns(munis :: Table) -> Table:
  doc: "Return a table of towns ordered by their growth"

  towns = filter-with(munis, is-town)

  towns-with-percent-change =
    build-column(towns, "percent-change", percent-change)

  order-by(towns-with-percent-change, "percent-change", false)
end
```

We’ve done a bit of a bad thing here: We’ve written three functions, but we don't have tests for any of them!

Let’s see how we can rectify this.

Testing table functions

We can test table program by using *test tables*.

These are tables that have the same *structure* as the table for our real data, but which are *smaller* and contain data that are useful for testing.

```
test-municipalities =
  table: name, kind, pop-2010, pop-2020
    row: "Osgiliath", "City", 100, 101
    row: "Lake-town", "Town", 100, 102
    row: "Bee", "Village", 100, 99
    row: "Hobbiton", "Town", 50, 54
  end
```

Let’s see how we use these test data to write examples for our table functions.

```
test-municipalities =
  table: name, kind, pop-2010, pop-2020
    row: "Osgiliath", "City", 100, 101
    row: "Lake-town", "Town", 100, 102
    row: "Bee", "Village", 100, 99
    row: "Hobbiton", "Town", 50, 54
  end

fun is-town(r :: Row) -> Boolean:
  doc: "Check if a row is for a town"
  r["kind"] == "Town"
end
```

```
test-municipalities =
  table: name, kind, pop-2010, pop-2020
    row: "Osgiliath", "City", 100, 101
    row: "Lake-town", "Town", 100, 102
    row: "Bee", "Village", 100, 99
    row: "Hobbiton", "Town", 50, 54
  end

fun is-town(r :: Row) -> Boolean:
  doc: "Check if a row is for a town"
  r["kind"] == "Town"
where:
  is-town(test-municipalities.row-n(0)) is false
  is-town(test-municipalities.row-n(1)) is true
  is-town(test-municipalities.row-n(2)) is false
end
```

```
test-municipalities =
  table: name, kind, pop-2010, pop-2020
    row: "Osgiliath", "City", 100, 101
    row: "Lake-town", "Town", 100, 102
    row: "Bee", "Village", 100, 99
    row: "Hobbiton", "Town", 50, 54
  end

fun percent-change(r :: Row) -> Number:
  doc: "Compute the percentage change for the population of the given
municipality between 2010 and 2020"
  (r["pop-2020"] - r["pop-2010"]) / r["pop-2010"]
end
```

```
test-municipalities =
  table: name, kind, pop-2010, pop-2020
    row: "Osgiliath", "City", 100, 101
    row: "Lake-town", "Town", 100, 102
    row: "Bee", "Village", 100, 99
    row: "Hobbiton", "Town", 50, 54
  end

fun percent-change(r :: Row) -> Number:
  doc: "Compute the percentage change for the population of the given
municipality between 2010 and 2020"
  (r["pop-2020"] - r["pop-2010"]) / r["pop-2010"]
where:
  percent-change(test-municipalities.row-n(0)) is 0.01
  percent-change(test-municipalities.row-n(1)) is 0.02
  percent-change(test-municipalities.row-n(2)) is -0.01
end
```

```
test-municipalities =
  table: name, kind, pop-2010, pop-2020
    row: "Osgiliath", "City", 100, 101
    row: "Lake-town", "Town", 100, 102
    row: "Bee", "Village", 100, 99
    row: "Hobbiton", "Town", 50, 54
  end

fun fastest-growing-towns(munis :: Table) -> Table:
  doc: "Return a table of towns ordered by their growth"
  towns = filter-with(munis, is-town)
  towns-with-percent-change =
    build-column(towns, "percent-change", percent-change)
  order-by(towns-with-percent-change, "percent-change", false)
end
```

```
test-municipalities =
  table: name, kind, pop-2010, pop-2020
    row: "Osgiliath", "City", 100, 101
    row: "Lake-town", "Town", 100, 102
    row: "Bee", "Village", 100, 99
    row: "Hobbiton", "Town", 50, 54
  end

fun fastest-growing-towns(munis :: Table) -> Table:
  ...
where:
  test-municipalities-after =
    table: name, kind, pop-2010, pop-2020, percent-change
      row: "Hobbiton", "Town", 50, 54, 0.08
      row: "Lake-town", "Town", 100, 102, 0.02
    end
  fastest-growing-towns(test-municipalities) is test-municipalities-after
end
```

Don't just copy the function's output; think through what it's ***supposed*** *to do!*

Program from today's class:

https://code.pyret.org/editor#share=1rkjGgObH5sBbbOV2XzRRQqM3ixFCiuxd&v=6d122f0

Acknowledgments

This class incorporates material from:

Kathi Fisler, Brown University

Doug Woos, Brown University