Designing Programs for Tables

27 September 2021
Where are we?
We can represent complex data as tables.

These can be encoded directly in a program or loaded from an external source.

Real data may need clean-up, which can be manual or automatic.
Automatic data clean-up includes using *sanitizers*, which ensure all data in a column is of the desired type, providing default values for empty cells.
We can modify table data later using `transform-column`, and we can remove (apparent) bad data using `filter-with`. 
We saw how this clean-up process works last week looking at real data recorded by the London Fire Brigade on animal rescues they performed.
include shared-gdrive("dcic-2021", "1wyQj_L0qqV9Ekgr9au6RX2iqt2Ga8Ep")
include gdrive-sheets

# To get the sanitizers
include data-source

# The ID of the Google Sheets file, which appears in the URL
ssid = "1zC4aSvcQpvWH-q0ie25fhoKfa0sAc8mL7skUvj4mY_Y"

rescue-data =
load-table:
  cal-year, final-description, animal-group-parent, origin-of-call,
  property-type, property-category, special-service-type-category,
  special-service-type, ward, borough, stn-ground-name, street,
  postcode-district
source: load-spreadsheet(ssid).sheet-by-name("rescues-simple", true)
sanitize borough using string-sanitizer
sanitize ward using string-sanitizer
end

rescue-data-clean = transform-column(rescue-data, "borough", string-to-upper)
<table>
<thead>
<tr>
<th></th>
<th>final-description</th>
<th>animal-group-parent</th>
<th>origin-of-call</th>
<th>property-type</th>
<th>property-type</th>
</tr>
</thead>
<tbody>
<tr>
<td>2009</td>
<td>Redacted</td>
<td>Dog</td>
<td>Person (land line)</td>
<td>House - single occupancy</td>
<td>Dwelling</td>
</tr>
<tr>
<td>2009</td>
<td>Redacted</td>
<td>Fox</td>
<td>Person (land line)</td>
<td>Railings</td>
<td>Outdoor</td>
</tr>
<tr>
<td>2009</td>
<td>Redacted</td>
<td>Dog</td>
<td>Person (mobile)</td>
<td>Pipe or drain</td>
<td>Outdoor</td>
</tr>
<tr>
<td>2009</td>
<td>Redacted</td>
<td>Horse</td>
<td>Person (mobile)</td>
<td>Intensive Farming Sheds (chickens, pigs etc)</td>
<td>Non Resi</td>
</tr>
<tr>
<td>2009</td>
<td>Redacted</td>
<td>Rabbit</td>
<td>Person (mobile)</td>
<td>House - single occupancy</td>
<td>Dwelling</td>
</tr>
<tr>
<td>2009</td>
<td>Redacted</td>
<td>Unknown - Heavy Livestock Animal</td>
<td>Person (land line)</td>
<td>House - single occupancy</td>
<td>Dwelling</td>
</tr>
<tr>
<td>2009</td>
<td>Redacted</td>
<td>Dog</td>
<td>Person (land line)</td>
<td>Park</td>
<td>Outdoor</td>
</tr>
<tr>
<td>2009</td>
<td>Redacted</td>
<td>Dog</td>
<td>Person (mobile)</td>
<td>Lake/pond/reservoir</td>
<td>Outdoor</td>
</tr>
<tr>
<td>2009</td>
<td>Redacted</td>
<td>Squirrel</td>
<td>Person (land line)</td>
<td>House - single occupancy</td>
<td>Dwelling</td>
</tr>
<tr>
<td>2009</td>
<td>Redacted</td>
<td>Dog</td>
<td>Person (mobile)</td>
<td>River/canal</td>
<td>Outdoor</td>
</tr>
</tbody>
</table>

Click to show the remaining 7863 rows...
Task plans
If you aren’t sure how to approach a problem, don’t start by trying to write code.

Plan until you understand the problem.
1 Develop a concrete example of desired output
   Typically a table with 4–6 rows

2 Identify functions useful to transform data
   Functions you already know or look up in the documentation

3 Develop a sequence of steps to transform data
   Draw as pictures, use textual descriptions, or a combination of the two
   Use functions from previous step

4 Repeat Step 3 to further break down steps until you can write expressions/functions to perform each step
Let’s say we want to make a pie chart of the distribution of rescues for the different boroughs each year.
<table>
<thead>
<tr>
<th>Cal-year</th>
<th>Borough</th>
</tr>
</thead>
<tbody>
<tr>
<td>2009</td>
<td>&quot;HACKNEY&quot;</td>
</tr>
<tr>
<td>2015</td>
<td>&quot;CAMDEN&quot;</td>
</tr>
<tr>
<td>2020</td>
<td>&quot;HACKNEY&quot;</td>
</tr>
<tr>
<td>2020</td>
<td>&quot;LAMBETH&quot;</td>
</tr>
</tbody>
</table>

Using `filter-with`:

<table>
<thead>
<tr>
<th>Cal-year</th>
<th>Borough</th>
</tr>
</thead>
<tbody>
<tr>
<td>2020</td>
<td>&quot;HACKNEY&quot;</td>
</tr>
<tr>
<td>2020</td>
<td>&quot;LAMBETH&quot;</td>
</tr>
</tbody>
</table>

Using `count`:

<table>
<thead>
<tr>
<th>Value</th>
<th>Count</th>
</tr>
</thead>
<tbody>
<tr>
<td>&quot;HACKNEY&quot;</td>
<td>1</td>
</tr>
<tr>
<td>&quot;LAMBETH&quot;</td>
<td>1</td>
</tr>
</tbody>
</table>
Programming with tables
Our examples from the task plan can become the examples (tests) for our functions:

\[
\text{test-table} =
\begin{array}{ll}
\text{table: } & \text{cal-year, borough} \\
\text{row: } & 2009, \ "HACKNEY" \\
\text{row: } & 2015, \ "CAMDEN" \\
\text{row: } & 2020, \ "HACKNEY" \\
\text{row: } & 2020, \ "LAMBETH" \\
\end{array}
\]
So, here’s how we might do it for 2020:

```plaintext
fun in-2020(r :: Row) -> Boolean:
    doc: "Return true if the row is a rescue from 2020"
    r["cal-year"] == 2020
where:
    in-2020(test-table.row-n(0)) is false
    in-2020(test-table.row-n(2)) is true
end

fun rescues-in-2020(rescues :: Table) -> Table:
    count(filter-with(rescues, in-2020), "borough")
end

rescue-counts-2020 = rescues-in-2020(rescue-data-clean)
pie-chart(rescue-counts-2020, "value", "count")
```
Oh no!
To start with, let's filter our data to only keep good rows:

```plaintext
test-table =
  table: cal-year, borough
  row: 2009, "HACKNEY"
  row: 2015, "CAMDEN"
  row: 2020, "HACKNEY"
  row: 2020, "LAMBETH"
  row: 2020, ""
end

fun good-row(r :: Row) -> Boolean:
  doc: "Return true if a row has a borough"
  not(r["borough"] == "")
where:
  good-row(test-table.row-n(0)) is true
  good-row(test-table.row-n(4)) is false
end

rescue-data-clean =
  filter-with(
    transform-column(rescue-data, "borough", string-to-upper),
    good-row)
```
fun `in-2020(r :: Row) -> Boolean:
    doc: "Return true if the row is a rescue from 2020"
    r["cal-year"] == 2020
where:
in-2020(test-table.row-n(0)) is false
in-2020(test-table.row-n(2)) is true
end

fun `rescues-in-2020(rescues :: Table) -> Table:
    count(filter-with(rescues, in-2020), "borough")
end

rescue-counts-2020 = rescues-in-2020(rescue-data-clean)
pie-chart(rescue-counts-2020, "value", "count")

To test this function, we need to know what order the output counts will be!
fun **in-2020**(r :: Row) -> Boolean:
    doc: "Return true if the row is a rescue from 2020"
    r["cal-year"] == 2020

where:
    in-2020(test-table.row-n(0)) is false
    in-2020(test-table.row-n(2)) is true
end

test-table-2020-counts =
    table: value, count
    row: "", 1
    row: "HACKNEY", 1
    row: "LAMBETH", 1
end

fun **rescues-in-2020**(rescues :: Table) -> Table:
    order-by(
        count(filter-with(rescues, in-2020), "borough"),
        "value", true)

where:
    rescues-in-2020(test-table) is test-table-2020-counts
end
Ah, better!
What if we want to do the same for 2019?

We could copy the code, replacing “2020” with “2019”, but that’s a little unsatisfying!

Instead, let’s create a function:

```latex
fun rescues-in-year(rescues :: Table, year :: Number) -> Table:
  fun in-year(r :: Row) -> Boolean:
    doc: "Return true if the row is a rescue from the given year"
    r["cal-year"] == year
  end

  data-for-year = filter-with(rescues, in-year)
  counts = count(data-for-year, "borough")
  order-by(counts, "value", true)
  where:
    rescues-in-year(test-table, 2020) is test-table-2020-counts
end
```
Now for any year we want to chart, we can just do this:

```plaintext
table{rescues-in-year(rescue-data-clean, 2019), "value", "count")
```
Nested functions simplified
fun rescues-in-year(rescues :: Table, year :: Number) -> Table:
    fun in-year(r :: Row) -> Boolean:
        doc: "Return true if the row is a rescue from the given year"
        r["cal-year"] == year
    end

data-for-year = filter-with(rescues, in-year)
counts = count(data-for-year, "borough")
order-by(counts, "value", true)
where:
    rescues-in-year(test-table, 2020) is test-table-2020-counts
end

This function is boring!
fun rescues-in-year(rescues :: Table, year :: Number) -> Table:

  data-for-year = filter-with(rescues,
    lam(r): r["cal-year"] == year end)

  counts = count(data-for-year, "borough")
  order-by(counts, "value", true)

where:

  rescues-in-year(test-table, 2020) is test-table-2020-counts

dend
A *lambda expression* defines an anonymous function – a function that can be passed as an argument but doesn’t have an associated name.
Lecture code:

https://code.pyret.org/editor#share=19jT5l1SzsWh-34NNn6UQC_n5G0Tgt_kt&v=1904b2c
Acknowledgments

This lecture incorporates material from:

Kathi Fisler, Brown University
Marc Smith, Vassar College