From Tables to Lists

Data-Centric Introduction to Coding, Chapter 5

event-data starter file

Just as we have used the notation `.row-n` to pull a single row from a table, we use a similar dot-based notion to pull out a single column. Here's how we extract the `tickcount` column:

```
cleaned-data.get-column("tickcount")
```

In response, Pyret produces the following value:

```
[list: 2, 1, 5, 0, 3, 10, 3]
```

Now, we seem to have only the values that were in the cells in the column, without the enclosing table. Yet the numbers are still bundled up, this time in the `[list: ...]` notation. What is that?

5.1.3 Understanding Lists

A list has much in common with a single-column table:

- The elements have an order, so it makes sense to talk about the “first”, “second”, “last”—and so on—element of a list.
- All elements of a list are expected to have the same type.

The crucial difference is that a list does not have a “column name”; it is *anonymous*. That is, by itself a list does not describe what it represents; this interpretation is done by our program.

5.1.3.2 Creating Literal Lists

We have already seen how we can create lists from a table, using `get-column`. As you might expect, however, we can also create lists directly:

```
[list: 1, 2, 3]
[list: -1, 5, 2.3, 10]
[list: "a", "b", "c"]
[list: "This", "is", "a", "list", "of", "words"]
```

Of course, lists are values so we can name them using variables—

```
shopping-list = [list: "muesli", "fiddleheads"]
```

—pass them to functions (as we will soon see), and so on.

5.1.3.2 Creating Literal Lists

Do Now!

Based on these examples, can you figure out how to create an empty list?

Did anyone use filter?

Found certain data in our original tables to make a new table to graph

Looking at college majors, I wanted to narrow down the types. I made a function to filter out the “humanities” majors to look at population and men/women numbers.

If there is a “category” column that lists the categories in Strings, we can make a helper that checks if category is “humanities.” If it is, we keep that row.

Remove

```
remove :: List<A>, A → List<A>

a = [list: “a”, “b”, “c”]
b = [list:

example:

remove(a, “a”) -> [list: “b”, “c”]

remove(c, “c”) -> [list: “Aa”, “bb”, “cC”]
```

5.1.4.2 Design Challenge

Exercise

Write a function that takes a list of words and removes those words in which all letters are in lowercase. (Hint: combine `string-to-lower` and `==`).

Create a task plan:

Write some examples for your new function.

This will probably need a helper. Can you use your examples to come up with the helper?

Task Plan: Group Ideas

- Make a function that checks if it's all lowercase. If it is, remove it
- Helper function: looks at just one string to see if it's all lowercase
- `is-lowercase ::`
- `filter`
- We want to KEEP the ones that have uppercase
- `has-uppercase :: String -> Boolean`
-

Helper Function Contract and Examples

5.1.4.4

Let's look at a new analysis question: the events company recently ran an advertising campaign on `web.com`, and they are curious whether it paid off. To do this, they need to determine how many sales were made by people with `web.com` email addresses.

Strategy: Creating a Task Plan

1. Develop a concrete example showing the desired output on a given input (you pick the input: a good one is large enough to show different features of your inputs, but small enough to work with manually during planning. For table problems, roughly 4-6 rows usually works well in practice).

2. Mentally identify functions that you already know (or that you find in the documentation) that might be useful for transforming the input data to the output data.

3. Develop a sequence of steps—whether as pictures, textual descriptions of computations, or a combination of the two—that could be used to solve the problem. If you are using pictures, draw out the intermediate data values from your concrete example and make notes on what operations might be useful to get from one intermediate value to the next. The functions you identified in the previous step should show up here.

4. Repeat the previous step, breaking down the subtasks until you believe you could write expressions or functions to perform each step or data transformation.

Task Plan Ideas

- Separate the email string apart
- Find the web.coms
- Count them

- Filter by the email addresses containing web.com
- Get web.com by itself

- Make a list of all the emails that made sales
- Sort by web.com email addresses
- Count them
- Collected data in a table, with an emails column
- Sorting by web.com

- Emails in strings in a column
- Turn that into a List
- Identify the strings that have web.com in them
- Filter the email list to only contain the web.com emails
- Use length() to figure out how many we have

The plan…

1. Get the list of email addresses (use `get-column`)
2. Extract those that came from `web.com` (use `L.filter`)
3. Count how many email addresses remain (using `L.length`, which we hadn't discussed yet, but it is in the documentation)

5.1.4.4 Wolf’s Handout Questions

1.
2. .get(1) took an element from a List of Strings
3.

What have we done with lists so far?

- They are basically columns but they are their own thing
- We can sort them
- They can contain all of the same data type
- Apparently they can be used for video games
- We can split lists, remove and filter them
- You can easily organize a lot of information from them
- When you do filter you switch the inputs
- Organizing information is simplified for readers
- We can count how many are in a list
- Distinct and remove

5.1.4.6 Recap: Summary of List Operations

At this point, we have seen several useful built-in functions for working with lists:

- `filter :: (A -> Boolean), List<A> -> List<A>`, which produces a list of elements from the input list on which the given function returns `true`.
- `map :: (A -> B), List<A> -> List<B>`, which produces a list of the results of calling the given function on each element of the input list.
- `distinct :: List<A> -> List<A>`, which produces a list of the unique elements that appear in the input list.
- `length :: List<A> -> Number`, which produces the number of elements in the input list.

Here, a type such as `List<A>` says that we have a list whose elements are of some (unspecified) type which we'll call `A`. A *type variable* such as this is useful when we want to show *relationships* between two types in a function contract. Here, the type variable `A` captures that the type of elements is the same in the input and output to `filter`. In `map`, however, the type of element in the output list could differ from that in the input list.

One additional built-in function that is quite useful in practice is:

- `member :: List<A>, Any -> Boolean`, which determines whether the given element is in the list. We use the type `Any` when there are no constraints on the type of value provided to a function.