CS 106 Winter 2016
Craig S. Kaplan

Module 11
Structured Data

Topics

- Working with tabular data
- Working with hierarchical data
- Accessing live APIs

Readings

- TBA

Introduction

In the previous module we explored some fairly simple "data shapes" in the context of text processing. We looked at blocks of text that were almost completely unstructured, in which we could nevertheless find meaning by searching with tools like regular expressions. We also saw a few examples based on simple sequences of data, like the word list used to power a spell checker.

In this final module, we'll look at data shapes where there's more built-in structure. Specifically, we'll consider tabular data (e.g., spreadsheets) and hierarchical data (e.g., family trees). The data we process will still be stored in text files, but we'll use more powerful tools to recognize the structure in those text files and take advantage of it.

Tabular data

For our purposes, a table is a rectangular grid of values, much like a spreadsheet. The grid has rows and columns. We usually think of each row as defining a single cohesive piece of data, a collection of individual pieces of information that are all related. Each column defines one piece of data associated with each record. In an iTunes music library, the rows are records that describe each song you own; the columns are pieces of information associated with songs (title, artist, genre, etc.). In a marking spreadsheet, the rows correspond to students and the columns to things that were marked (assignments, exams, etc.). Each column may have a *heading* (a name), and probably has a type (which we'll limit to `String`, `float`, and `int`).

A standard, platform-independent format for storing tabular data is a CSV (Comma-Separated Values) file. There are many online resources from which one can download interesting CSVs, whether it's fun collections of random data or governmental sites like the Region of Waterloo's catalog of public data. From the first of those sites, we'll play with the file HairEyeColor.csv, containing basic physical attributes of the students in a statistics class. The file starts out like this:

```
"","Hair","Eye","Sex","Freq"
"1","Black","Brown","Male",32
"2","Brown","Brown","Male",53
"3","Red","Brown","Male",10
"4","Blond","Brown","Male",3
"5","Black","Blue","Male",11
“6","Brown","Blue","Male",50
…
```

The top line of the file gives heading names to the columns—the words “Hair”, “Eye”, and so on are not actually part of the dataset. Each line below that contains a row of data. For example, the record

labeled “1” tells us that the class had 32 male students with black hair and brown eyes.

Now, without any extra tools, we might still be able to extract meaning from this file using straightforward text processing. We already know how to read the file line-by-line, and we could break that line into fields by using `splitTokens()` with the comma as a delimiter. But there’s a better way, because Processing understands this file type directly. If we save this file to a sketch’s data folder, we could write code like this:

```
Table stats;

void setup()
{
  stats = loadTable(
    "HairEyeColor.csv", "header" );
}
```

The built-in function `loadTable()` reads the file from the data folder and interprets it as a table. It stores the result in an instance of a special Processing class called `Table`, and gives you back that instance as a result. The `loadTable()` function takes an optional second parameter, a `String`. In this case the value “`header`” is used to instruct Processing to treat the first line of the file as a set of heading names rather than as data.

Working with an instance of Table is a bit like using Microsoft Excel, except that you tell the spreadsheet what you want to do by writing code. Most obviously, there are methods for figuring out the dimensions of the table and for reading and writing individual cells. There are also more complicated methods for building tables from scratch and finding rows that match certain patterns. It’s not the most fully-featured implementation of a spreadsheet, but it’s good enough for our purposes. See the documentation for the `Table` and `TableRow` classes for full information.

Accessing a cell in a table is like accessing a pixel in an image: you need to provide two “coordinates”, corresponding to the row and column of the cell. But

be careful: in a table, the row (which is a bit like the *y* coordinate of a pixel) comes first, as it does in a cell name like “B5” in Excel.

For example, we might calculate the total number of students in the statistics class by adding the following code to the `setup()` function defined above:

```
int total = 0;
for ( int idx = 0; idx < stats.getRowCount(); ++idx ) {
  total += stats.getInt( idx, "Freq" );
}
println( total );
```

The expression `stats.getInt( idx, “Freq” )` asks the table to give us the contents of the `Freq` column of the `idx`th row. The table needs to know what type we expect to find in that cell, which is why there are `getInt()`, `getFloat()` and `getString()` methods. Note that we could also have said `stats.getInt( idx, 4 )`, giving the explicit column number containing the value we’re looking for. When the column has a heading, we’re allowed to use that name instead.

Or perhaps we want to know the total number of blond students, in which case we first read the “Hair” column and check that the colour is the one we’re interested in:

```
int total = 0;
for ( int idx = 0; idx < stats.getRowCount(); ++idx ) {
  if ( stats.getString( idx, "Hair" ).equals( "Blond" ) ) {
    total += stats.getInt( idx, "Freq" );
  }
}
println( total );
```

Building a table

Looking through the catalogue of available data, I became curious about "Reserved Street Names", which claims to be available as a CSV. As it turns out, that's a lie. The CSV link opens up a text file with cute boxes drawn using ASCII Art:

```
-----------------------------------------
|   FullStreetName   |    Municipality    |
-----------------------------------------
| Abbey Glen         | Kitchener          |
-----------------------------------------
| Aberle             | Woolwich           |
-----------------------------------------
| Abeth              | Kitchener          |
-----------------------------------------
| Abitibi            | Cambridge          |
-----------------------------------------
| Able               | Cambridge          |
-----------------------------------------
| Abram Clemens St   | Kitchener          |
```

OK, yes, they also make the same data available in an Excel spreadsheet, and I could easily export a real CSV file from Excel. But this turns out to be a nice lesson in handling messy real-world data. We should be able to write some Processing code to read this file despite the extraneous ASCII Art. We can use the Text Processing tools we learned previously to extract the useful bits from this file. Examining the file, it's clear that any line that starts with a hyphen ("-") can be discarded. In each useful line, the bits we care about (the street name and municipality) are separated by "|" characters, which we can treat as a delimiter. Using those ideas, we can construct a table from scratch:

```
Table table = new Table();
table.addColumn( "Name" );
table.addColumn( "Municipality" );

String[] lines =
loadStrings( "ReservedStreetnames.txt" );
for ( int idx = 0; idx < lines.length; ++idx ) {
  if ( !lines[idx].startsWith( "-" ) ) {
    String[] boxes = splitTokens( lines[idx], "|" );
    boxes = trim( boxes );

    if ( !boxes[0].equals( "FullStreetName" ) ) {
      TableRow row = table.addRow();
      row.setString( "Name", boxes[0] );
      row.setString( "Municipality", boxes[1] );
    }
  }
}
```

Once we've constructed the table above, we could then treat it like any other table and decide how we'd like to process the data.

Example sketch: ReservedStreetsTable

Let's move on to something more interesting: money. The province of Ontario requires that the salaries of all public sector employees who make over $100,000 a year be made available to the public. The data is all online. Again, though, it's not in a particularly convenient format: you can look at it as HTML on a web page, or you can download a PDF. That's a good starting point, but what if I want to perform real calculations on the data? For example:

- Who is the most highly paid person in a given sector in the province? The most highly paid at a given institution?
- What is the average salary for all the people with a given job title?
- What's the most extreme salary inversion, e.g., the highest paid junior professor?

It's perhaps inconsiderate of the province to make this information available in a format that can't easily be used as a basis for further calculation. But we can usually work around that deficiency. Computer scientists sometimes talk of "scraping", "snarfing", or "munging" data. A good starting point in this case might be to examine the HTML source of the web page, and do text processing (using regular expression?) to extract the individual fields from the HTML. It's not obvious, but there's a much easier way to do this—simply highlight the entire table on one of these HTML pages and copy it into a text file. My Chrome browser inserts tab characters at all column breaks in the table. We can then take advantage of a second option for the second argument to `loadTable()`: we can tell is to treat the file as *tab*-separated values (TSV) instead of comma-separated values.

```
Table table = loadTable(
  ”salaries.txt”, “header, tsv” );
```

Note the comma in "`header, tsv`". That's not part of Processing per se. That's something that the person who implemented `loadTable()` decided on as a way to pass more than one optional parameter to the function.

Actually this doesn't quite work, because when we cut and paste the salary table we don't get a header row. But that's a small problem. The easiest fix is to tweak the data file and add that first row manually so that the table "knows about" the names of its columns.

One final problem is that we want to collapse the two final columns of the table in the source file into a single column containing the sum of salaries and benefits. The sample sketch shows how to do this.

Example sketch: SalariesTable

As a final example of reading tabular data, consider the Region of Waterloo's food inspection reports. This dataset is more like what's called a "relational database". It comes in three separate tables:

- A *Facilities* file, in which each row gives complete information about a single food-serving facility in the region. The first column is a unique ID code associated with each facility, which will be used in the other tables to refer to it.
- An *Inspections* file, a table that lists individual inspections in which someone visited a facility. Each inspection has its own unique ID, and mentions the ID of the facility to record where the inspection took place.
- An *Infractions* file. Each inspection may result in zero or more infractions against the food safety code. These infractions are recorded one per row in this file. The infraction record refers back to the inspection ID.

The good news is that each of these files is in a proper CSV format, and can be read into a sketch in a single line of code. The difficulty is that in order to do interesting things, you have to gather information from across multiple tables. For example, here's how to list all the infractions associated with a given restaurant, given the restaurant's name:

- Iterate over the rows of the Facilities table. For each row, check if the "BUSINESS_NAME" column matches the name you're looking for. If it does, save the ID associated with that name.
- Iterate over the rows of the Inspections table. For each row, if the "FACILITYID" column matches the facility you're looking for, append the associated inspection ID to an array of strings.
- Finally, gather together an array of all the infractions in the Infractions table whose "INSPECTION_ID" is in the list of IDs set aside in the previous step.

Wow, that's a lot of work. It's more than I expected would be necessary when I started playing with the dataset, and more than I would ever ask for in this course. But the result is fairly cool—a sketch in which you can type in the name of a restaurant and see a

complete list of its infractions. That was sufficiently worthwhile that I decided to include it here. (It's still only a bit over 100 lines of code.)

Example sketch: FoodInspections

Hierarchical data

Sometimes we want to obtain data from the outside world that isn't as cleanly structured as an array of objects or a table. For example, we might want to load in all of the information about a restaurant. That data might include a wide array of heterogeneous data:

- The name, address and phone number of the restaurant (as strings)
- The opening hours, which could be an array of seven objects, each of which is made up of two strings (the opening time and closing time each day of the week). Or maybe each time is given as two integers, and hour and a minute
- A list of strings giving links to review sites
- A set of menus (breakfast, lunch, dinner), each of which contains a heading describing the menu together with an array of records giving names, descriptions and prices of dishes.
- etc.

Data shapes like arrays and tables are good for lots of applications (including sub-parts of our hypothetical restaurant information), but they just can't handle this kind of freeform structured data in full. There are numerous ways that this kind of data does get represented in practice. Two very popular forms are XML (eXtended Markup Language, which we won't talk about in this course) and JSON (JavaScript Object Notation, which will form the rest of the module).

JSON is a very small subset of the Javascript (not Java!) language, which can be used to describe collections of data. It started out as a means for a

script running on a web page to exchange data with a web server. But it was so simple and useful that it became a bit of a standard way for programs to send structured information back and forth. In particular, it's built in to Processing.

Loading JSON objects

The simplest way to get a JSON object into a sketch is to load it from a file. The format should be familiar by now, as it's analogous to reading images, vector illustrations and tables.

```
JSONObject my_obj =
  loadJSONObject( "data.json" );
```

Reading data from JSON objects

A `JSONObject` behaves a lot like an instance of a class. It has a number of named *fields*, and each field has an associated type, and stores a value. There are six types that we'll need to think about for fields. A field can have one of the familiar types `int`, `float`, `boolean` or `String`. A field can also contain an array, which is stored using the class `JSONArray`, or it can even contain a nested (smaller) `JSONObject`.

However, a `JSONObject`'s fields aren't treated like class fields by Processing. So after loading in `my_obj` above, you can't say something simple like `my_obj.fieldname` as you might with a regular class instance. Instead, you need to call a method of `my_obj` that reads the contents of the field for you. If you knew that your JSONObject had a field called "`name`" of type `String` and a field called "`weight`" of type `int`, then you could write code like this:

```
String obj_name = my_obj.getString( "name" );
int obj_weight = my_obj.getInt( "weight" );
```

In a way, this is a bit more like reading the value associated with a key in a dictionary object.

A `JSONObject` can contain another `JSONObject` in one of its fields, and it can contain a `JSONArray` in one of its fields. Similarly, a `JSONArray` can contain a `JSONObject` or another `JSONArray` in any of its numbered entries.

There's nothing magical about this, but it just means that there may be cases where you need to attach a few calls to, e.g., `getJSONObject()` together to "drill down" to the lowest level of a chunk of hierarchical data:

```
JSONArray my_arr = my_obj.getJSONArray( "data" );
JSONObject my_event = my_arr.getJSONObject( 0 );
String name = my_event.getString( "name" );
```

...or, if you're feeling a bit more intense, you can do all of this in one statement, borrowing a bit of unusual syntax from ControlP5:

```
String name = my_obj
  .getJSONArray( "data" )
  .getJSONObject( 0 )
  .getString( "name" );
```

You almost certainly won't be constructing `JSONObject` instances from scratch, so once you've acquired an instance by loading it in, you basically just need to use the methods `getString()`, `getInt()`, `getFloat()`, `getBoolean()`, `getJSONArray()`, and `getJSONObject()`. Each of these methods takes a single `String` as its argument, corresponding to the name of the field you want to retrieve. The `JSONArray` class supports exactly the same methods, except in that case they take an `int` as an argument, corresponding to the location you want to read from the array. It's a bit like reading a character from a string. The string likes to pretend it's an array but it isn't, so you must use the `charAt()` method instead of square brackets. With a `JSONArray` you must use one of the `get` methods above.

The easiest way to tell what fields a JSON object supports is to read the documentation provided by whoever gave you the object. If that doesn't work, it's helpful to look at the object itself (i.e., the raw input file)—they're pretty easy to read, which is sort of the point.

Example sketch: SimplestJSON

Web APIs

An API (Application Programming Interface) is a fancy software engineering name for something we've dealt with all term. It's the set of functions that a given library understands, through which you access its features. So far this term, all the APIs we've used have either been built in to Processing, or accessible through an `import` statement.

But here's the exciting bit—many online services offer APIs as well! We use many online services these days that organize vast amounts of data on our behalf. Social media certainly works this way (Facebook stores piles of status updates, photos, notes, lists of friends etc.), as do photo-sharing sites, cloud-based storage, and other information resources. Some of these services publish public APIs through which you can access the underlying data without having to knock on the front door by visiting the service with a web browser. This is a very powerful idea: it lets you write you own custom applications that use a pile of data without getting stuck with the service's view of how that data should be viewed. It's also useful to the company themselves, since they can standardize on how information is sent back and forth across a number of devices and operating systems.

You can think of accessing a Web API as "calling a function over the internet". You're calling a function in order to obtain some piece of information as a result, but instead of your computer working out the answer to the function on its own, it sends the request off to a second computer. That computer figures out the answer and ships it back to you in the form of a `JSONObject` (or some other structured data value).

In order to call the function you want, you need to package up your request into a form that can be sent off to the other computer. The nice bit is that this doesn't require any new ideas or code. Web API calls

look just like URLs, and you can “call the function” by giving the URL you want to `loadJSONObject()`.

A good source of examples is api.uwaterloo.ca, the University of Waterloo’s own internal open data API. It supports a number of different function calls that return information about the campus and its environs. Visit the API’s documentation page to see the list of queries you can make. If you want to find out the current weather, for example, you can access the following URL: https://api.uwaterloo.ca/v2/weather/current.json. Try it now in your browser! Hopefully it will show you the text of the JSON as output (it works in Chrome, at least). So, if you store that JSON object in a variable:

```
JSONObject weather =
  loadJSONObject( "https://api.uwaterloo.ca/v2/weather/current.json" );
```

Then you can start querying the variable `weather` to find out things like the current temperature and precipitation.

If you try other UW API calls, you’ll find that they return an error object saying that you need an API key. Most Web APIs require you to pass in a key along with your request. The key identifies you, and allows the web service to track how much you’re using their data (which is useful, for example, if they want to bill you for your use of their service, or at least cut you off if you abuse the service).

If you try searching the internet for the name of your favourite online service together with “API”, you’ll see the range of tools available to programmers. Twitter’s API is a particularly well known one. Facebook has one for the graph of your connections to your friends, but not for status updates. The Google Maps API powers a large number of online tools by third parties. Even IMDB and Rotten Tomatoes publish APIs for getting at live movie data. Many mobile apps are really just thin user interfaces wrapped around accesses to Web APIs. Waterloo’s getting in on the game too. As of last year, the API powers the new Waterloo Student Portal.