CS 106 Winter 2016
Craig S. Kaplan

Module 09
The shape of data

Topics

- The scale of data
- Data shapes

Readings

- None

Introduction

We utterly awash in data. Collectively, humanity generates a truly staggering amount of information on a continual basis. One doesn't have to look too hard to find estimates of the magnitude of this information. A good site that provides an overview is www.internetlivestats.com. We send hundreds of billions of email messages per day (mostly spam, but still), and hundreds of billions of instant messages on top of that. Of course, the amount of data in these messages is quite small—video is the real bandwidth hog on the internet. It's estimated that 37% of all internet traffic in North America is due to Netflix alone.

What's the real challenge here? Obviously there are serious engineering challenges involved in creating a worldwide infrastructure that can handle this torrent of information. It takes big technology to collect and generate information, to store it, to move it from place to place, and to access it efficiently. All of those operations consume electricity and generate waste heat, and so the energy efficiency of large data

centres is rapidly becoming a major technological problem for the 21st century. Of course, this course will not address these engineering challenges! For us, there's a more fundamental problem that dominates: how do we find *meaning* in this data? How do we find and keep the tiny slice of the global information feed that's of interest to us, and build tools to increase our understanding of it?

In this course we won't solve those problems either, but we can start to examine some of the programming tools that will help us, and that lead naturally to the ones used out in the world for data manipulation, analysis, and visualization. Here are a few small examples of ways that we allow programmers to help us make sense of data:

- *Searching and indexing*: computers can track complex relationships between documents and the pieces that make them up. Your email client can index a message based on the words it contains, so that you can easily filter emails later and find one that you care about. Your computer might maintain an index of all the files you've edited based on filename and time, so that you can go back later and find that assignment submission after you forget what folder you dropped it into.

- *Collecting, correlating, recommending*: computers can build simple models of our interests and preferences based on explicit clues we provide and a number of hints in our online habits. We can codify our collective wisdom and use it to figure out where to go out for dinner, what movies and music we might like, and even who we might hit it off with romantically. The same technology is used to try to show you online ads that you're more likely to click on, since more ad clicks generates more revenue for websites. (I'm not judging whether these technologies are good or bad—I find many of them creepy myself—I'm just giving examples of what's out there today.)

- *Patterns, trends, predictions*: If we feed a large enough volume of data into analysis or visualization software, we can sometimes spot trends that would

have been too difficult to see when looking at information at a normal human scale. We use this sort of analysis to decide whether an incoming email message is spam, so that we never need see it in our inbox. We now use large-scale data analysis to look for trends in human health, to predict the spread of disease or find causes of illnesses. Governments use profiling technology to identify potential terrorists. The media uses it to make predictions about the outcomes of elections. (Again, not judging, just enumerating.)

Data shapes

Let's set aside rich multimedia data like images, sound and video for the time being and think only about data that comes to us in text form. Even in this simple form, it helps to think about the "shape" of the data. How is it organized? How do the individual parts of the data relate to the whole, and to each other? Answers to questions like these can have a profound effect on the kind of code we end up writing to process that data, and the structures we use to represent it. Of course, these are high-level design questions, and therefore they don't necessarily have one right answer. But it's important to have *some* answer in mind when setting out to write a program, to avoid getting lost in manifold options.

With that in mind, here are a few typical "data shapes" that we might encounter when dealing with real-world information, together with some thoughts about how programmers think about dealing with such data.

Raw text

Raw text doesn't have any particular structure apart from being one large block of undifferentiated text. This text might represent a chapter from a book, the body of an email, a transcript of a speech, or many other things. The most natural way to store a block of

raw text in Processing is undoubtedly to use a `String`. Of course, we may often end up using an array of `Strings` instead, since that is the type of information given back to us by the built-in `loadStrings()` function and many of our textual information will come to us from external files.

With raw text, we might ask questions like search queries: does this text contain a given word? Of course, more elaborate visualizations are always possible. A fun recent analysis extracted the relative frequencies of punctuation marks in passages from different novels, revealing that authors differed significantly in their punctuation styles. Perhaps the most famous visualization style for raw text is the word cloud, as exemplified by wordle.net. Duke university has a page with a number of additional text visualizations.

Sequences

We are often presented a collection of related pieces of information that flow in a natural sequence. Earlier in the term we drew a bar graph of oil prices, given a list of prices over a two-month period (as a text file). We might record the positions of objects over time by writing their *x* and *y* coordinates to a text file. We might be interested in processing a list of words, or of names, or ingredients for recipes, and so on.

Any time we have a sequence of information, we'll probably want to store it in some kind of sequence data structure. The most obvious choice, and the one we've used throughout this term, is the array, though there are in fact many other options available to us. A skilled programmer will pick the best sequence type to use in a given context; for this course, we'll be able to do just about everything with arrays.

When dealing with an array, you should immediately be thinking about using a `for` loop at some point: you'll probably want to visit every element in the array

and extract something from it or modify it in some way. We might walk through an array to find an element of interest to us, or perhaps apply the same transformation (e.g., adding one) to every element.

Dictionaries

A dictionary is a way to create an association or mapping from a set of *keys* to a set of *values*. A key is a unique identifier that we use to keep track of a set of objects that we're interested in, and the value is the information associated with that key.

In an actual dictionary, the keys are words and the values are the definitions of those words. A more apt example of a dictionary in this course would be the mapping from Clicker IDs to student ID numbers, which ensures that we know who to assign marks to when you answer a clicker question. Every key (i.e., every Clicker ID in our dictionary) should map to exactly one student ID. Another example is the exam seating chart, which maps student IDs to seat numbers in the exam room.

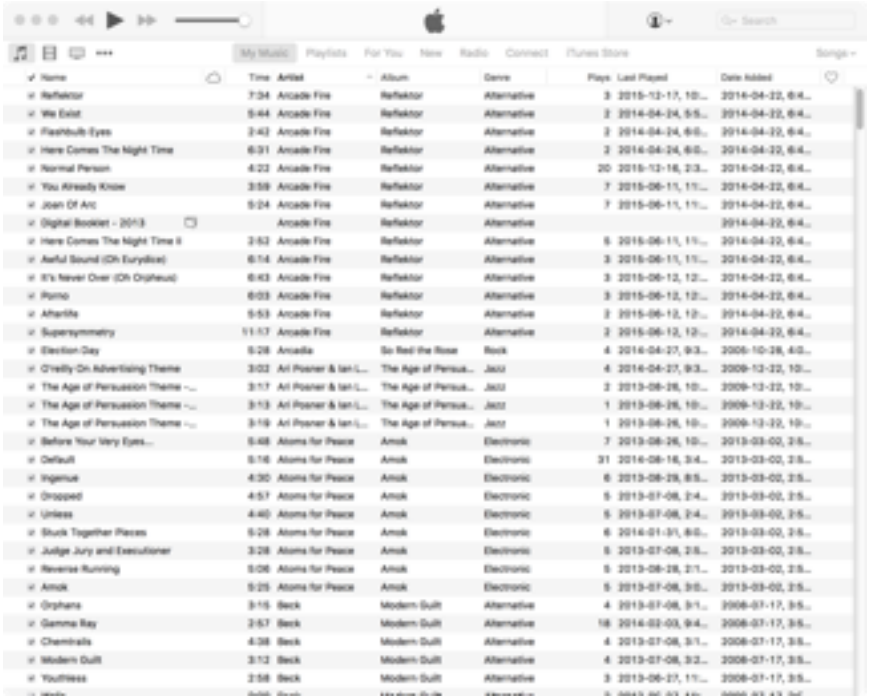
In a dictionary, we expect to perform operations like looking up the value associated with a key, adding a new key/value pair, and removing a key and its associated value. In theory we could imagine storing a dictionary as an array, where every element of the array keeps track of both a key and a value. If we were mapping `String` keys to `int` values, we might end up with code like this:

```
class Assoc
{
  String my_key;
  int my_value;
};

Assoc[] my_dictionary;

int lookup( String a_key )
{
  for( int idx = 0; idx < my_dictionary.length; ++idx ) {
    if( a_key.equals( my_dictionary[idx].my_key ) ) {
      return my_value;
    }
  }
  // Key not found, must return something.
  return -1;
}
```

This works, but it's actually quite slow. In practice there are far better ways to represent dictionaries. We won't try to implement them ourselves! In Processing we have easy access to classes like `IntDict` (which maps `Strings` to `ints`) and `StringDict` (`Strings` to `Strings`). See the Processing documentation for more, including sample code that uses these built-in classes.

Tables

A simple spreadsheet is basically a grid of numbers. Many databases also look more or less like grids—they consist of sequences of *records*, where each record is some kind of aggregate piece of data containing many small fields. The most obvious and straightforward example of a database of this kind would be the database of all your music, as maintained by iTunes.



We still have a 2D grid here, and we can think about the meaning of the rows (individual songs) and of the columns (properties of songs). Each cell in the grid consists of a single property of one song, like its title or duration. Another example might be your list of contacts on your mobile phone: a list of people, where each person has a first and last name, an email address, a phone number, and so on.

There are two natural ways to thing about organizing this sort of “table-shaped” data. The first is to use a class that represents a row of the table, with one field for each piece of data in a record. In the case of songs in iTunes, we might come up with this:

```
class Song
{
  String title;
  String artist;
  int duration;
  String genre;
  // ... Other pieces of information here
}
```

Then, the entire database can be represented as an array of `Song` instances.

The other approach would be to use a data structure that explicitly recognizes the 2D grid. In Processing, there is a built-in class called `Table` that does exactly this. Working with a `Table` instance is a bit like working with an Excel spreadsheet: you can read and write the values of individual cells, add and remove rows and columns, and so on. And as we'll see, this class can read and write comma-separated values (.csv) files, which are compatible with most spreadsheet packages.

Hierarchical data

Sometimes, information is stored *hierarchically*: a given piece of information may have "children" underneath it, which themselves may have further children, and on and on. The family tree of a person's descendants is a good first example: any given person may be the "end" of the hierarchy, or the tree may continue underneath them if they have children. The organizational chart of a company behaves similarly (though the arrangement gets complicated if an employee has multiple supervisors). The filesystem on your computer is very much hierarchical: every folder might contain files and still more folders. A final example is threaded online discussions, as one might find on reddit. There, every comment has a unique parent (the comment it replied to), and may be the bottom of the chain or have further replies beneath it.

In all of these cases, we need to represent the information in a kind of tree, where each "node" in the tree is aware of the nodes that are underneath it. Often, when we want to process this tree, we'll want to visit every node. If that node has children, we must visit them too, but we can't predict the "shape" of the tree in advance. Therefore, in this context we often write recursive functions.

We won't work a lot with trees in this course, but at the very end we'll look briefly at JSON as a means of

receiving live information from Web APIs. JSON (Javascript Object Notation) is a way to represent hierarchical information as text; the other popular way to do so is XML.

Graph-structured data

Thinking still more generally, sometimes the connections between pieces of information is more or less arbitrary. Consider friend networks in social media. Facebook consists of a large block of data for each user. Users are connected via a “friendship” link. But there’s no strict hierarchy or organization. Any individual could potentially be friends with any other, or not. The only thing we know is that friendship is always mutual on Facebook: if I’m friends with you, then you’re friends with me. Mathematicians refer to this kind of structure as an *undirected graph*: a set of nodes and a set of two-way connections between those nodes. The alternative, a *directed graph*, is modelled perfectly by Twitter. Twitter’s connection is “following”, and I can follow you without you following me. When drawing a network to represent connections in a directed graph, we would usually include an arrowhead to indicate the “direction” of the connection.

We won’t attempt to do anything with graphs in this course. I just wanted to include one final example in this section of notes to show that there are more complicated data shapes that you can discover after moving beyond the material you’ll see here.