Data visualization

COSC 480B

Reyan Ahmed

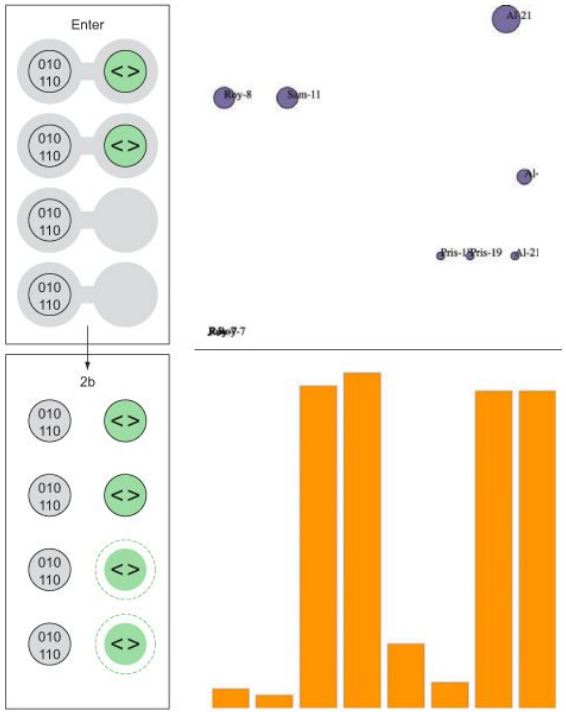
rahmed1@colgate.edu

Lecture 6

Information visualization data flow

Data binding

A diagram of how data-binding works, a scatterplot with labels, and a bar chart

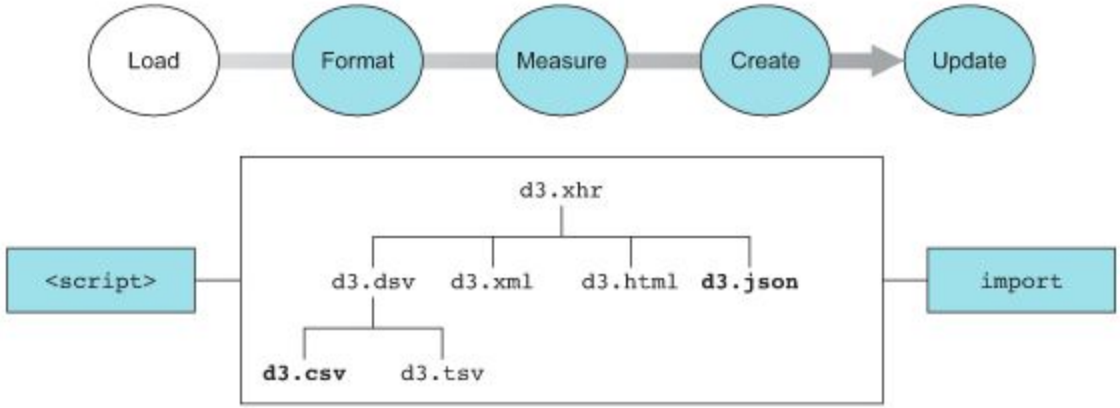


Working with data

- We will touch on everything you need to do with and to data to turn it into a data visualization
- Will consider two datasets:
  - Geographic data
  - Fictional tweets

Loading data

- Data formats
  - XML (nested, XML format)
  - CSV (not nested, JSON format)
  - JSON (nested, JSON format)
- The root technique is XML Http Request (XHR)



File formats

- d3.text()
- d3.xml()
- d3.json()
- d3.csv()
- d3.html()

```
d3.csv("cities.csv", (error,data) => {console.log(error,data)});
```

```
d3.csv("cities.csv", d => console.log(d));
```

File formats

- XHR is asynchronous
- If you need synchronous use:
  - Script tag
  - Import/require keyword
- For static data use synchronous method
- For asynchronous data use:
  - Callback functions
  - Global variables to access later

File formats

File contents of cities.csv

```
"label","population","country","x","y"
"San Francisco", 750000,"USA",122,-37
"Fresno", 500000,"USA",119,-36
"Lahore",12500000,"Pakistan",74,31
"Karachi",13000000,"Pakistan",67,24
"Rome",2500000,"Italy",12,41
"Naples",1000000,"Italy",14,40
"Rio",12300000,"Brazil",-43,-22
"Sao Paolo",12300000,"Brazil",-46,-23
```

File formats

File contents of tweets.json


```
{
"tweets": [
{"user": "Al", "content": "I really love seafood.",
  "timestamp": " Mon Dec 23 2013 21:30 GMT-0800 (PST)",
  "retweets": ["Raj","Pris","Roy"], "favorites": ["Sam"]},
{"user": "Al", "content": "I take that back, this doesn't taste so good.",
  "timestamp": "Mon Dec 23 2013 21:55 GMT-0800 (PST)",
  "retweets": ["Roy"], "favorites": []},
{"user": "Al",
  "content": "From now on, I'm only eating cheese sandwiches.",
  "timestamp": "Mon Dec 23 2013 22:22 GMT-0800 (PST)",
  "retweets": [],"favorites": ["Roy","Sam"]},
{"user": "Roy", "content": "Great workout!",
  "timestamp": " Mon Dec 23 2013 7:20 GMT-0800 (PST)",
  "retweets": [],"favorites": []},
{"user": "Roy", "content": "Spectacular oatmeal!",
  "timestamp": " Mon Dec 23 2013 7:23 GMT-0800 (PST)",
  "retweets": [],"favorites": []},
{"user": "Roy", "content": "Amazing traffic!",
  "timestamp": " Mon Dec 23 2013 7:47  GMT-0800 (PST)",
  "retweets": [],"favorites": []},
{"user": "Roy", "content": "Just got a ticket for texting and driving!",
  "timestamp": " Mon Dec 23 2013 8:05 GMT-0800 (PST)",
  "retweets": [],"favorites": ["Sam", "Sally", "Pris"]},
{"user": "Pris", "content": "Going to have some boiled eggs.",
  "timestamp": " Mon Dec 23 2013 18:23 GMT-0800 (PST)",
  "retweets": [],"favorites": ["Sally"]},
{"user": "Pris", "content": "Maybe practice some gymnastics.",
  "timestamp": " Mon Dec 23 2013 19:47  GMT-0800 (PST)",
  "retweets": [],"favorites": ["Sally"]},
{"user": "Sam", "content": "@Roy Let's get lunch",
  "timestamp": " Mon Dec 23 2013 11:05 GMT-0800 (PST)",
  "retweets": ["Pris"], "favorites": ["Sally", "Pris"]}
]
}
```

File formats

With these two files, we can access the data by using the appropriate function to load them:

```
d3.csv("cities.csv", data => console.log(data));
d3.json("tweets.json", data => console.log(data));
```

NOTE THAT, the cities are in tabular format, hence, we have to access it through “data”. On the other hand, tweets are in json format, so we have to access it through “data.tweets”.

File formats

- Again, data loading is asynchronous
- Use call back functions

```
d3.csv("somefiles.csv", function(data) {doSomethingWithData(data)});
```

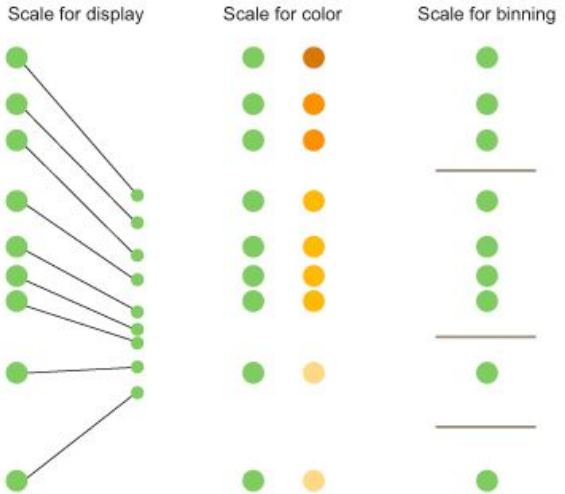
- Another technology is to use “promises” which we will see in future
- For text blocks (instead of file) use d3.csv().parse()
- Use d3-request for more fine grained control

Formatting data

- Datatypes
  - Quantitative (numerical values, represented by size, color, position...)
  - Categorical (nationality or gender…, represented by colors)
  - Geometric (rivers, cities, …, represented by latitude longitude)
  - Temporal (dates, times, …, functions in d3 are available)
  - Topological (relationship like genealogical connection, represented by hierarchies)
  - Raw (text, tweets, …)

Formatting data

After loading data, you need to make sure it's formatted in such a way that it can be used to create graphics. This includes mapping the data to positions on the screen, colors that indicate quantity, or bins to nest the data visually.



Formatting data

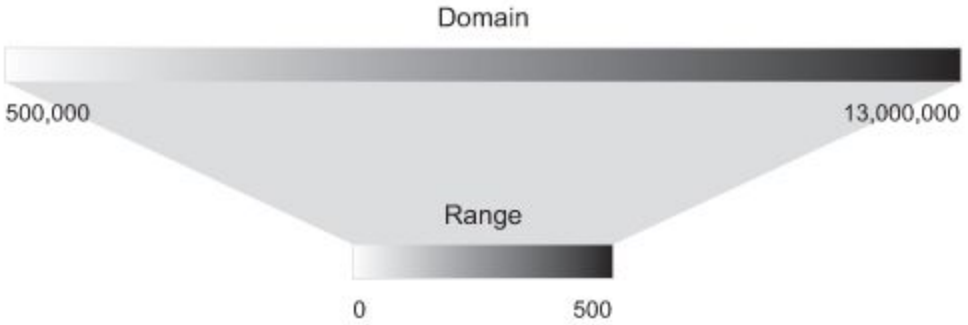
- We can change data types, for example from string to date

```
parseInt("77"); +"77";
parseFloat("3.14"); +"3.14"
Date.parse("Sun, 22 Dec 2013 08:00:00 GMT");
text = "alpha,beta,gamma"; text.split(",");
```

- If you use “==”, you need to convert to appropriate data type
- If you use “===”, automatically data is converted

Scales and scaling

Scales in D3 map one set of values (the domain) to another set of values (the range) in a relationship determined by the type of scale you create.



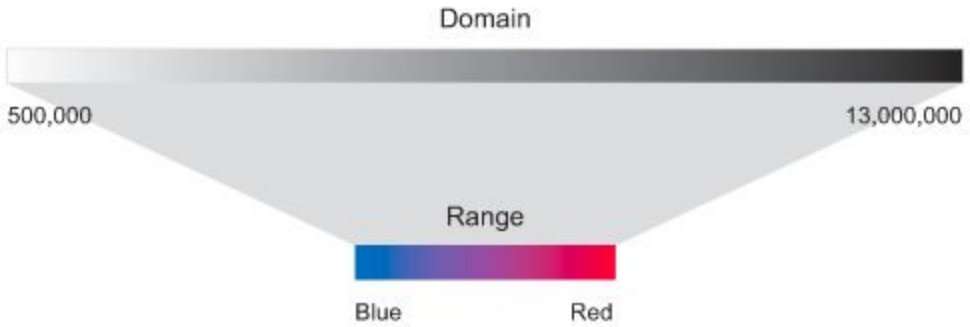
Scales and scaling

```
var newRamp = d3.scaleLinear().domain([500000,13000000]).range([0, 500]);
newRamp(1000000);                    1
newRamp(9000000);                    2
newRamp.invert(313);                 3
```

- 1 Returns 20, allowing you to place a country with population 10,000,000 at 20 px
- 2 Returns 340
- 3 The invert function reverses the transformation, in this case returning 8325000

Scales and scaling

Scales can also be used to map numerical values to color bands, to make it easier to denote values using a color scale.



Scales and scaling

```
var newRamp = d3.scaleLinear().domain([500000,13000000]).range(["blue",
"red"]);
newRamp(1000000);                    1
newRamp(9000000);                    2
newRamp.invert("#ad0052");              3
```

- 1 Returns “#0a00f5”, allowing you to draw a city with population 1,000,000 as dark purple
- 2 Returns “#ad0052”
- 3 The invert function only works with a numeric range, so inverting in this case returns NaN

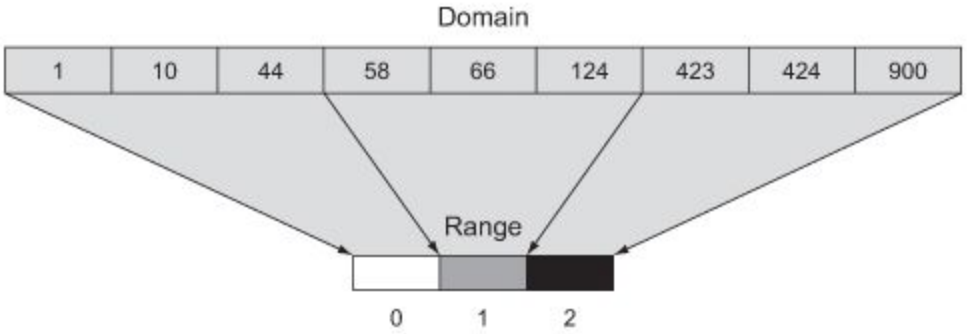
Scales and scaling

```
var sampleArray = [423,124,66,424,58,10,900,44,1];
var qScale = d3.scaleQuantile().domain(sampleArray).range([0,1,2]);
qScale(423);                                                    1
qScale(20);                                                     2
qScale(10000);                                                  3
```

- 1 Returns 2
- 2 Returns 0
- 3 Returns 2

Scales and scaling

Quantile scales take a range of values and reassign them into a set of equally sized bins.



Scales and scaling

```
var qScaleName =
d3.scaleQuantile()
   .domain(sampleArray).range(["small","medium","large"]);
qScaleName (68);                                    1
qScaleName (20);                                    2
qScaleName (10000);                                   3
```

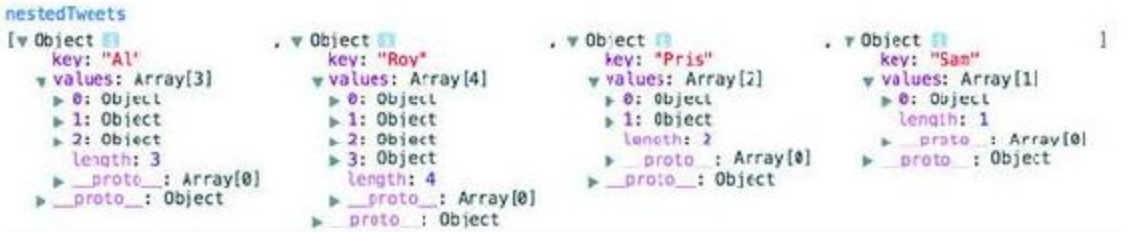
- 1 Returns “medium
- 2 Returns “small”
- 3 Returns “large”

Nesting

```
d3.json("tweets.json", data => {
  var tweetData = data.tweets;
  var nestedTweets = d3.nest()
    .key(d => d.user)
    .entries(tweetData);
});
```

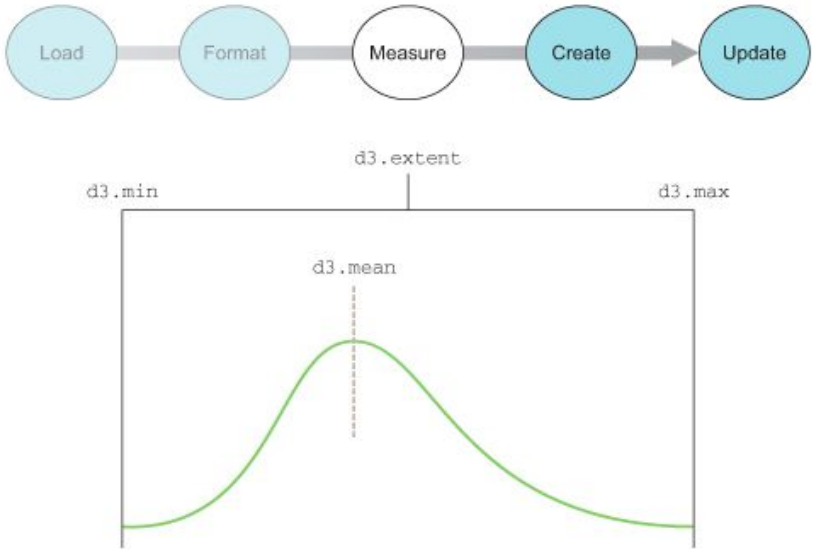
Nesting

Objects nested into a new array are now child elements of a values array of newly created objects that have a key attribute set to the value used in the d3.nest.key function.



Measuring data

After formatting your data, you'll need to measure it to ensure that the graphics you create are appropriately sized and positioned based on the parameters of the dataset. You'll use d3.extent, d3.min, d3.mean, and d3.max all the time.



Measuring data

```
var testArray =  [88,10000,1,75,12,35];
d3.min(testArray, el => el);                1
d3.max(testArray, el => el);                2
d3.mean(testArray, el => el);               3
```

- 1 Returns the minimum value in the array, 1
- 2 Returns the maximum value in the array, 10000
- 3 Returns the average of values in the array, 1701.8333333333335

Measuring data

```
d3.csv("cities.csv", data => {
d3.min(data, el => +el.population);          1
d3.max(data, el => +el.population);          2
d3.mean(data, el => +el.population);         3
d3.extent(data, el => +el.population);       4
});
```

- 1 Returns the minimum value of the population attribute of each object in the array, 500000
- 2 Returns the maximum value of the population attribute of each object in the array, 1300000
- 3 Returns the average value of the population attribute of each object in the array, 6856250
- 4 Returns [500000, 1300000]

Selections and binding

To create graphics in D3, you use selections that bind data to DOM elements.

Selections and binding

```
d3.csv("cities.csv", (error,data) => {
   if (error) {
   console.error(error)
   }
   else {
   dataViz(data)
   }
   });
function dataViz(incomingData) {
  d3.select("body").selectAll("div.cities")          1
    .data(incomingData)                              2
    .enter()                                    3
    .append("div")                                   4
    .attr("class","cities")                          5
    .html(d => d.label);                              6
}
```

1 An empty selection because there are no elements in <body> with class of “cities”

2 Binds the data to your selection

3 Defines how to respond when there’s more data than DOM elements in a selection

4 Creates an element in the current selection

5 Sets the class of each newly created element

6 Sets the content of the created

Selections and binding

When our selection binds the cities.csv data to our web page, it creates eight new divs, each of which is classed with "cities" and with content drawn from our data.

Selections and binding

- d3.selectAll()
  - Often no elements match the identifier
  - Use .enter() function to create elements
  - Not that enter function selects the empty elements, does not create elements
  - We need to call .append() to create elements

Selections and binding

- data()
  - associate an array with the DOM elements you selected
  - We could access these values manually using Java-Script like so:

```
document.getElementsByClassName("cities")[0].__data__
```

- Returns a pointer to the object representing San Francisco

Selections and binding

- enter() and .exit()
  - .enter() define behavior for every value that doesn’t have a DOM
  - When fewer data elements exist, then .exit() behavior is triggered
- append() and .insert()
  - Both creates element, but enter gives you control to decide where to add
- attr()
  - To set class property, applies to all new elements
- html()
  - To add html content

Accessing data with inline functions

```
d3.select("svg")
  .selectAll("rect")
  .data([15, 50, 22, 8, 100, 10])
  .enter()
  .append("rect")
  .attr("width", 10)                    1
  .attr("height", d => d);               2
```

- 1 Sets the width of the rectangles to a fixed value
- 2 Sets the height equal to the value of the data associated with each element

Accessing data with inline functions

The default setting for any shape in SVG is black fill with no stroke, which makes it hard to tell when the shapes overlap each other.

Accessing data with inline functions

By changing the opacity settings, you can see the overlapping rectangles.

Accessing data with inline functions

```
d3.select("svg")
  .selectAll("rect")
  .data([15, 50, 22, 8, 100, 10])
  .enter()
  .append("rect")
  .attr("width", 10)
  .attr("height", d => d)
  .style("opacity", .25);
```

Accessing data with inline functions

What about changing other variable?

```
...
.style("opacity", .25)
  .attr("x", (d,i) => i * 10);
```

Accessing data with inline functions

SVG rectangles are drawn from top to bottom.

Accessing data with inline functions

It seems like, it’s reversed, let’s negate the y values.

```
...
  .attr("height", d => d)
  .style("fill", "#FE9922")
  .style("stroke", "#9A8B7A")
  .style("stroke-width", "1px")
  .attr("x", (d,i) => i * 10)
  .attr("y", d => 100 - d);
```

Accessing data with inline functions

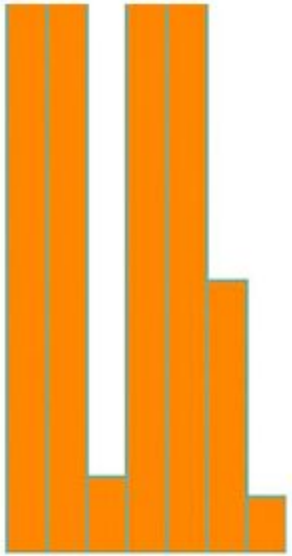
When we set the y position of the rectangle to the desired y position minus the height of the rectangle, the rectangle is drawn from bottom to top from that y position.

Integrating scales

```
[14, 68, 24500, 430, 19, 1000, 5555]
...
  .selectAll("rect")
  .data([14, 68, 24500, 430, 19, 1000,
5555])
  .enter()
 ...
```

Integrating scales

SVG shapes will continue to be drawn offscreen.



Integrating scales

And it works no better if you set a y offset equal to the maximum:

```
...
  .selectAll("rect")
  .data([14, 68, 24500, 430, 19, 1000,
5555])
  .enter()
  .append("rect")
  .attr("y", d => 24500 - d)
...
```

Integrating scales

We can use the d3 scalings.

```
var yScale =
d3.scaleLinear().domain([0,24500]).range([0,100]);
yScale(0);                                                    1
yScale(100);                                                   2
yScale(24000);                                                  3
```

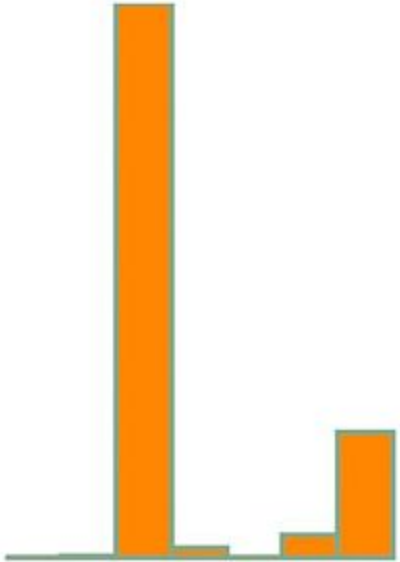
- 1 Returns 0
- 2 Returns 0.4081632653061224
- 3 Returns 97.95918367346938

Integrating scales

```
var yScale = d3.scaleLinear()
.domain([0,24500]).range([0,100]);
...
  .attr("width", 10)
  .attr("height", d => yScale(d))
  .attr("y", d => 100 - yScale(d));
  .style("fill", "#FE9922")
...
```

Integrating scales

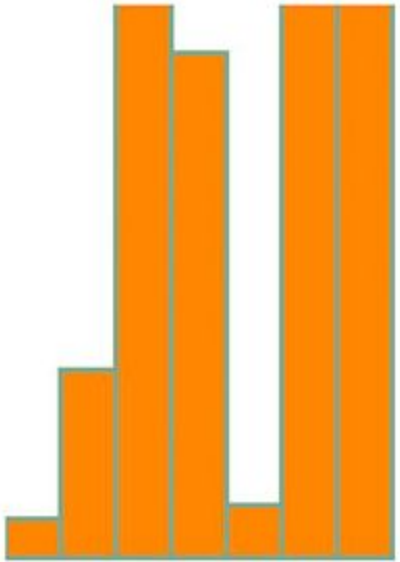
A bar chart drawn using a linear scale



Integrating scales

The same bar chart from figure 2.17 drawn with a polylinear scale

```
var yScale = d3.scaleLinear().domain([0,100,1000,24500]).range([0,50,75,100]);
```



Integrating scales

You may assume that selecting a cut-off value will resolve the problem.

```
var yScale = d3.scaleLinear().domain([0,100,500]).range([0,50,100]);
```

Integrating scales

By default d3 doesn’t clamp:

```
yScale(1000);          1
```

- 1 Returns 162.5

Integrating scales

A bar chart drawn with values in the dataset greater than the maximum value of the domain of the scale, but with the clamp() function set to true

```
var yScale = d3.scaleLinear()
                .domain([0,100,500])
                .range([0,50,100])
                .clamp(true);
```

```
yScale(1000);          1
```

- 1 Returns 100