E-14a

Lab 5: InfoVIS with d3.js

d3.js library

Learning Objectives

- Use SVG element
- Work with JavaScript
- Create basic D3 visualizations
- Understand enter-update-exit pattern
- Use higher-order functions
- Update DOM elements with D3
- Update visualizations with new data

- Use layouts for more advanced vis types

INTRODUCTION

D3 *(Data-Driven Documents or D3.js) is a JavaScript library for visualizing data using web standards. D3 helps you bring data to life using SVG, Canvas and HTML. D3 combines powerful visualization and interaction techniques with a data-driven approach to DOM manipulation, giving you the full capabilities of modern browsers and the freedom to design the right visual interface for your data.*
Source: https://github.com/d3/d3/wiki

Like most well-maintained libraries, D3 aims to keep up with the ever-evolving web technologies it builds upon and to continue building useful tools for data visualization. So it is helpful to monitor versioning (especially when applying code from outside resources) - presently we are deep into v5. Among other improvements, the fifth version takes advantage of Javascript's Promise object for handling asynchronous programming.

Running Python Server

How the web works basics involve web servers, HTTP protocol, IP Addresses, server responses, and server port numbers. D3 has baked into it functionality that lets you load data into the browser asynchronously from a server. D3 can request and parse text files, JSON files, XML files, HTML files, CSV files, TSV files, as well as allowing you to build and use your own parser. The simplest server, Python `SimpleHTTPServer`, available on most computers comes as a module within the Python programming language. To run the Python 2.7+ `SimpleHTTPServer` on the command line, type:

```
python -m SimpleHTTPServer
```

To run the Python 3+ `SimpleHTTPServer` on the command line, type:

```
python3 -m http.server
```

Windows users should type:

```
$ winpty python -m http.server
```

The Python `SimpleHTTPServer` serves to `http://localhost:8000` (0.0.0.0:8000) all of the files in whichever directory you ran the command line commands.

Higher-order functions

Functions that operate on other functions, either by taking them as arguments or by returning them, are called higher-order functions. The term comes from mathematics, where the distinction between functions and other values is taken more seriously. Higher-order functions allow us to abstract over actions, not just values. They come in several forms. For example, we can have functions that create new functions. The most popular functions used together with d3 implementation are *forEach*, *filter* and *map*.

The following webpage gives a great examples of usage (together with executable code cells): https://eloquentjavascript.net/05_higher_order.html Knowing how to work with these functions will make any d3 visualizaton possible.

D3 setup

Before working with D3 you have to include the D3 JavaScript library. For this course we recommend the minified version (d3.min.js) which has a smaller file size and faster loading time. However, during development (and thus in the templates we provide), you might prefer to use the readable original version (d3.js). Your html file should look like the following:

```
    <head>
      <title>D3                                                    Project</title>
      <link rel="stylesheet" href="css/style.css">e

    </head>
    <body>
        <svg id="chart" width="500" height="300"></svg>

        <script                              src=""https://d3js.org/d3.v5.min.js""></script>
        <script                                        src="js/your_js_file.js"></script>
    </body>
    </html>
```

SVG

SVG drawings and images are created using a wide array of elements which are dedicated to the construction, drawing, and layout of vector images and diagrams. Here you'll find reference documentation for each of the SVG elements:
https://developer.mozilla.org/en-US/docs/Web/SVG/Element

The specifics of the SVG are the following:

1. SVG is defined using markup code similar to HTML
2. SVG elements don't lose any quality if they are resized
3. SVG elements can be included directly within any HTML document or dynamically inserted into the DOM with JavaScript
4. Before you can draw SVG elements you have to add an `<svg>` element with a specific width and height to your HTML document
5. In all our visualizations we will use pixels as the default measurement units (other supported units: em, pt, in, cm, mm)
6. The SVG coordinate system is based on x- and y-values to specify the element positioning. *(0/0) is located in the top-left corner of the drawing space and increasing x-values move to the right, while increasing y-values move down*.
7. SVG has no layering concept or depth property. The order in which elements are coded determines their depth order.
8. Basic shape elements in SVG: `rect`, `circle`, `ellipse`, `line`, `text` and `path`.

Method chaining

Method or function chaining is a common technique in JS, especially when working with D3. It can be used to simplify code in scenarios that involve calling multiple methods on the same object consecutively.

The functions are "chained" together with periods. The output type of one method has to match the input type expected by the next method in the chain. Alternative code without method chaining:

```
var body = d3.select("body");
var p = body.append("p");
p.text("Hello World!");
```

`d3` - References the D3 object, so we can access its functions.

`select()` - this method uses CSS selectors as an input to grab page elements. It will return a reference to the first element in the DOM that matches the selector.

In this example we use `d3.select("body")` to select the first DOM element that matches our CSS selector: body. Once an element is selected and handed off to the next method in the chain - you can apply operators. If you want to select more than one element, use `selectAll()`.

`append()` - After selecting a specific element we have used an operator to assign content: `.append("p")`. The `append()` operator adds a new element as the last child of the current selection. We specified `p` as the input argument, so an empty paragraph has been added to the end of the `body`. The new paragraph is automatically selected for further operations. At the end we have used the `text()` property to insert a string between the opening and closing tags of the current selection.

Anonymous functions

A simple anonymous JS function looks like the following:

```
function doSomething (d) {
  return d;
}
```

It has a function name, an input and an output variable. If the function name is missing, then it is called an anonymous function. If you want to use the function only in one place, an anonymous function is more concise than declaring a function and then doing something with it as two separate steps. We will use them very often in D3 to access individual values and to create interactive properties.

```
.text( function (d) { return d; } );
```

Also keep in mind that an arrow function expression can also be used.

```
.text((d) => { return d; } );
```

Selections

D3 selections allow DOM elements to be selected in order to do something with them, be it changing style, modifying their attributes, performing data-joins or inserting/removing elements.

For example, given 5 circles:

we can use `d3.selectAll` to select the circles and `.style` and `.attr` to modify them:

```
d3.selectAll('circle')
  .style('fill',                                                    'orange')
  .attr('r', function() {return 10 + Math.random() * 40;  });
```

D3 has two functions to make selections `d3.select` and `d3.selectAll`. `d3.select` selects the first matching element whilst `d3.selectAll` selects all matching elements. Each function takes a single argument which specifies the selector string. For example to select all elements with class item use `d3.selectAll('.item')`.

Loading External Data

Instead of typing the data in a local variable, which is only convenient for small datasets, we can load data asynchronously from external files. The D3 built-in methods make it easy to load JSON, CSV and other files.

CSV (Comma Separated Values)

Similar to JSON, CSV is a file format which is often used to exchange data. Each line in a CSV file represents a table row and as the name indicates, the values/columns are separated by a comma. In a nutshell: The use of the right file format depends on the data - JSON should be used for hierarchical data and CSV is usually a proper way to store tabular data.

By calling D3 methods like d3.csv(), d3.json(), d3.tsv() etc. we can load external data resources in the browser:

```
d3.csv("test.csv", function(d) {
  return {
```

```
    year: new Date(+d.Year, 0, 1), // convert "Year" column to Date
    make: d.Make,
    model: d.Model,
    length: +d.Length // convert "Length" column to number
  };
}).then(function(data) {
  console.log(data);
}).catch(function(err) {
  console.error(err);
});;
```

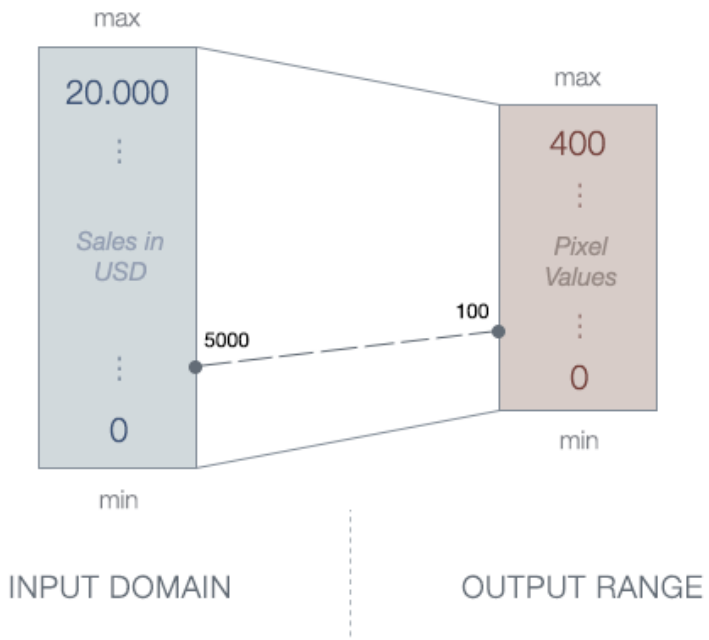
The `.csv()` function is one of many methods defined in the d3-fetch module. The method itself "fetches" the requested resource (in this case a .csv file) and allows the user to parse the input data as key/value pairs within object literals. This process occurs asynchronously and, upon successful completion. returns a Promise object (meaning that code wrapped in the chained `.then([...])` will not execute until the Promise is delivered).

Scales

Until now, when creating a D3 visualization, we used only x and y values that corresponded directly to pixel measurements on the screen, within a predefined SVG drawing area. That is not very flexible and only feasible for static data. What if our data attributes are suddenly doubled? We can not increase the size of the chart every time a value increases. At some point, the user might have to scroll through a simple bar chart to get all the information.

D3 provides built-in methods for many different scales: linear, ordinal, logarithmic, square root etc. Most of the time you will use linear scale functions, so we will focus on learning this type of scale. You can read more about D3 scales here:
https://github.com/d3/d3-scale/blob/master/README.md

Scales are functions that map from an input domain to an output range.
Example: We want to visualize the monthly sales of an ice cream store. The input data are numbers between 0 and 20,000 USD and the maximum height of the chart is 400px. We take an input interval (called Domain) and transform it into a new output interval (called Range).



We could transform the numbers from one domain into the other manually but what if the sales rise above 20.000 and the interval changes? That means a lot of manual work. Thankfully, we can use D3's built-in scaling methods to do this automatically.

D3 provides scale functions to convert the input domain to an output range. This was pretty easy, because we already knew the max value of the data. What if we load data from an external source and don't know the data range the data is going to be in? Instead of specifying fixed values for the domain, we can use the convenient array functions `d3.min()`, `d3.max()` or `d3.extent()`.

```
var quarterlyReport = [
    { month: "May", sales: 6900 },{ month: "June", sales: 14240 },
    { month: "July", sales: 25000 },{ month: "August", sales: 17500 }
];
var max = d3.max(quarterlyReport, function(d) {
    return d.sales;
});
var min = d3.min(quarterlyReport, function(d) {
    return d.sales;
});
var extent = d3.extent(quarterlyReport, function(d) {
    return d.sales;
});
```

SVG groups

In the previous section you have learned how to create basic SVG shapes, like rectangles or circles. But there is another SVG element that is very useful for programming in D3: the group element (<g></g>). In contrast to graphical elements the group element does not have a visual presence, but it helps you to organize other elements and to apply transformations. In this way, you can create hierarchical structures.

```
// Create group element
var group = svg.append("g");

// Append circle to the group
var circle = group.append("circle")
    .attr("r", 4)
    .attr("fill", "blue");
```

Group elements are invisible but you can apply transformations, for example translate() or rotate(), to the group and it will affect the rendering of all child elements!

```
var group = svg.append("g")
    .attr("transform", "translate(70, 50)");
```

Other Quantitative Scales:

1. d3.scaleSqrt() - Square root scale
2. d3.scalePow() - Power scale
3. d3.scaleLog() - Logarithmic scale

Ordinal Scales

In the previous examples we have used only quantitative scales but D3 also provides methods to create ordinal scales with a discrete domain.

```
// Create an ordinal scale function
var xScale = d3.scaleBand()
    .domain(["May", "June", "July", "August"])
    .range([0, 400]) // D3 fits n (=4) bands within this interval
    .paddingInnder(0.05);      // Adds spacing between bands
```

For example, D3's ordinal scales can be very useful to simplify the positioning of bars in a bar chart.

Color Scales

D3 has built-in color palettes that work like ordinal scales and can also be accessed like other scales:

Examples:

1. d3.scaleOrdinal(d3.schemeCategory10) - ordinal scale with a range of 10 categorical colors
2. d3.scaleOrdinal(d3.schemeCategory20) - ordinal scale with a range of 20 categorical colors

```
// Construct a new ordinal scale with a range of ten categorical colors
var colorPalette = d3.scaleOrdinal(d3.schemeCategory10);

// Print color range
console.log(colorPalette.range());
// ["#1f77b4", "#ff7f0e", "#2ca02c", "#d62728", "#9467bd", "#8c564b", "#e377c2",
"#7f7f7f", "#bcbd22", "#17becf"]
// Specify domain (optional)
colorPalette.domain(["Vanilla", "Cookies", "Chocolate", "Pistachio"]);

// Use color palette
colorPalette("Chocolate") // Returns: #2ca02c
```

Instead of using a fixed range of colors you can use linear scale functions to create color gradients:

```
var linearColor = d3.scaleLinear()
  .domain([0,100])
  .range(["darkgreen","lightgreen"]);
```

```
linearColor(0)      // Returns: #006400
linearColor(50)     // Returns: #48a948
linearColor(100)    // Returns: #90ee90
```

Axes

The current visualization does not really look like a scatterplot yet. It is just a bunch of circles that are nicely arranged. We need x- and y-axes to allow the user to actually extract meaningful insights from the visualization. You can accept an axis is the visual representation of a scale.

D3 provides four methods to create axes with different orientations and label placements (`d3.axisTop`, `d3.axisBottom`, `d3.axisLeft`, and `d3.axisRight`) which can display reference lines for D3 scales automatically. These axis components contain lines, labels and ticks.

```
// Create a horizontal axis with labels placed below the axis
var xAxis = d3.axisBottom();
// Pass in the scale function
xAxis.scale(xScale);
var xAxis = d3.axisBottom()
    .scale(xScale);
```

Finally, to add the axis to the SVG graph, we need to specify the position in the DOM tree and then we have to call the axis function.

We create an SVG group element as a selection and use the `call()` function to hand it off to the `xAxis` function. All the axis elements are getting generated within that group.

```
// Draw the axis
svg.append("g")
    .attr("class", "axis x-axis")
    .call(xAxis);
```

Recall that we can use the transform attribute to change the position and move the axis to the bottom.

```
// Draw the axis
svg.append("g")
    .attr("class", "axis x-axis")
    .attr("transform", "translate(0," + (height - padding) + ")")
    .call(xAxis);
```

Additionally, you can use the HTML class property as a selector and modify the style with:

```
.axis path,
.axis line {
    fill: none;
    stroke: #333;
    shape-rendering: crispEdges;
}

.axis text {
    font-family: sans-serif;
    font-size: 11px;
}
```

`shape-rendering` is an SVG property which specifies how the SVG elements are getting rendered. We have used it in this example to make sure that we don't get blurry axes.

D3 axis functions automatically adjust the spacing and labels for a given scale and range. Depending on the data and the type of visualization you may want to modify these settings.

```
var xAxis = d3.axisBottom()
    .scale(xScale)
    . ... // Add options here
```

There are many different options to customize axes:

1. Number of ticks: .ticks(5)
2. Tick format, e.g. as percentage: .tickFormat(d3.format(".0%"))
3. Tick values: .tickValues([0, 10, 20, 30, 40])

You can read more about D3 axis, ticks and tick formatting in the D3 API reference: https://github.com/d3/d3-axis/blob/master/README.md

Credits and Additional Resources

D3 in Depth:
https://d3indepth.com/selections/

Interactive Data Visualization for the Web: An Introduction to Designing with D3:
https://goo.gl/z9gf5c

Data Visualization at Harvard SEAS:
http://cs171.org/2019/

Eloquent JavaScript:
https://eloquentjavascript.net/