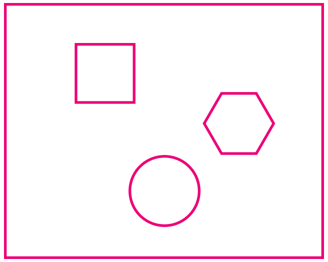
Week 4

Introduction to data visualization & d3

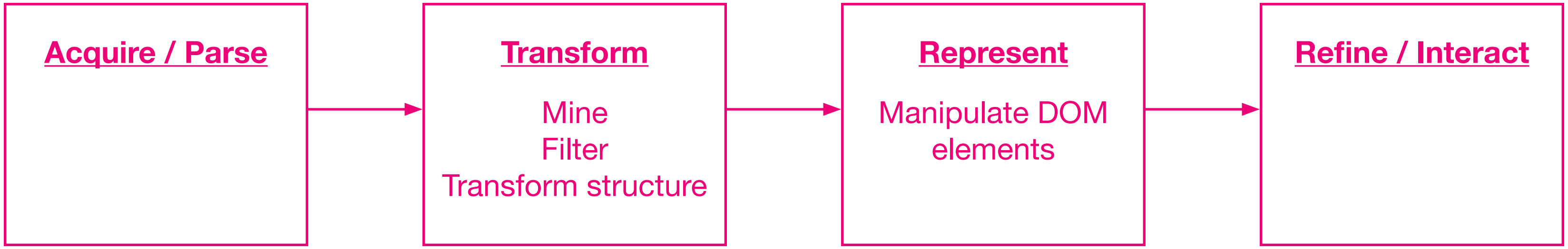
Let’s begin by thinking about a generalized “algorithm” for solving data visualization problems.

Represent

Manipulate DOM elements



Let’s begin by thinking about a generalized “algorithm” for solving data visualization problems.



d3 is a **general-purpose** data visualization library containing **modules** that deal with specific sets of tasks in the data visualization pipeline. It’s built on top of **Javascript**.

Overview of week 4

1. DOM manipulation: `d3-select`
    - What are selections?
    - Modifying and appending DOM elements
    - Iterating through selections
    - Working with `<svg>` elements
2. Data transform: `d3-math` and `d3-nest`
    - The accessor pattern
3. Data transform using arrays
4. Putting it all together

1.1 D3 selections

Open up Exercise 1. It helps to keep track of the DOM tree as we progress through the exercise with paper and a pen.

What does the DOM tree look like to start with?

1.1 D3 selections

```
d3.select( selector )
d3.selectAll ( selector )
```

returns a d3 selection

CSS selector

For now, think of a d3 selection as a **pointer** to one or a group of DOM nodes.

selection =/= DOM nodes. To get the underlying DOM node from a selection, use

```
selection.node()
```

1.1 D3 selections

Try to figure out what the following code does, before testing it out on your own:

```
d3.select(‘.container’)
   .selectAll(‘.block’)

d3.selectAll(‘.block-large’)

d3.selectAll(‘.container)
   .select(‘.block’)
```

1.2 Modifying selections: attributes and styles

With a selection, we can easily modify the attributes of the underlying DOM nodes with:

```
selection.attr()
selection.style()
selection.classed()
```

1.2 Modifying selections: attributes and styles

For example:

```
d3.select(‘#container-1’)
   .select(‘.block’)
   .attr(‘id’,’block-1’)
   .style(‘width’,’50%’)
   .classed(‘selected’, true)
```

1.2 Modifying selections: adding / removing nodes

Use `selection.append( )`

```
d3.select(‘#container-1’)
   .select(‘.block’)
   .append(‘div’)
   .attr(‘class’,’nested’)
```

Aside: method chaining

```
d3.select(“.yellow-boxes”)
    .append(“div”)
    .attr(“class”, “box”)
    .append(“div”)
    .attr(“class”, “inner”)
    .style(“width”, “50%”)
    .style(“background”,
“red”);
```

Select the <div> element with class “yellow-boxes”

Append an <div> element

Set the attributes on <div>

Append a <div> element under <div.box>

Set the attributes on <div>

Aside: method chaining

```
d3.select(“.yellow-boxes”)
    .append(“div”)
    .attr(“class”, “box”)
    .append(“div”)
    .attr(“class”, “inner”)
    .style(“width”, “50%”)
    .style(“background”,
“red”);
```

One more thing: how come we can keep “chaining” method calls one after another?

Aside: method chaining

```
d3.select(“.yellow-boxes”)
    .append(“div”)
    .attr(“class”, “box”)
    .append(“div”)
    .attr(“class”, “inner”)
    .style(“width”, “50%”)
    .style(“background”,
“red”);
```

One more thing: how come we can keep “chaining” method calls one after another?

- Each `.attr()` call returns the old selection, for you to call a new method onto it;
- Each `.append()` call returns the newly appended elements as the new selection, for you to call a new method onto it.

Aside: method chaining

How is this different from the previous example?

```
var container = d3.select(“.yellow-boxes”);
container
    .append(“div”)
    .attr(“class”, “box”);
container
    .append(“div”)
    .attr(“class”, “inner”)
    .style(“width”, “50%”)
    .style(“background”, “red”);
```

1.3 Iterating through selections

Since selections can represent a group of DOM nodes, sometimes we need to access and iterate through individual nodes in a selection using `selection.each(function)`

Let’s examine the API for selection.each carefully and answer three questions:

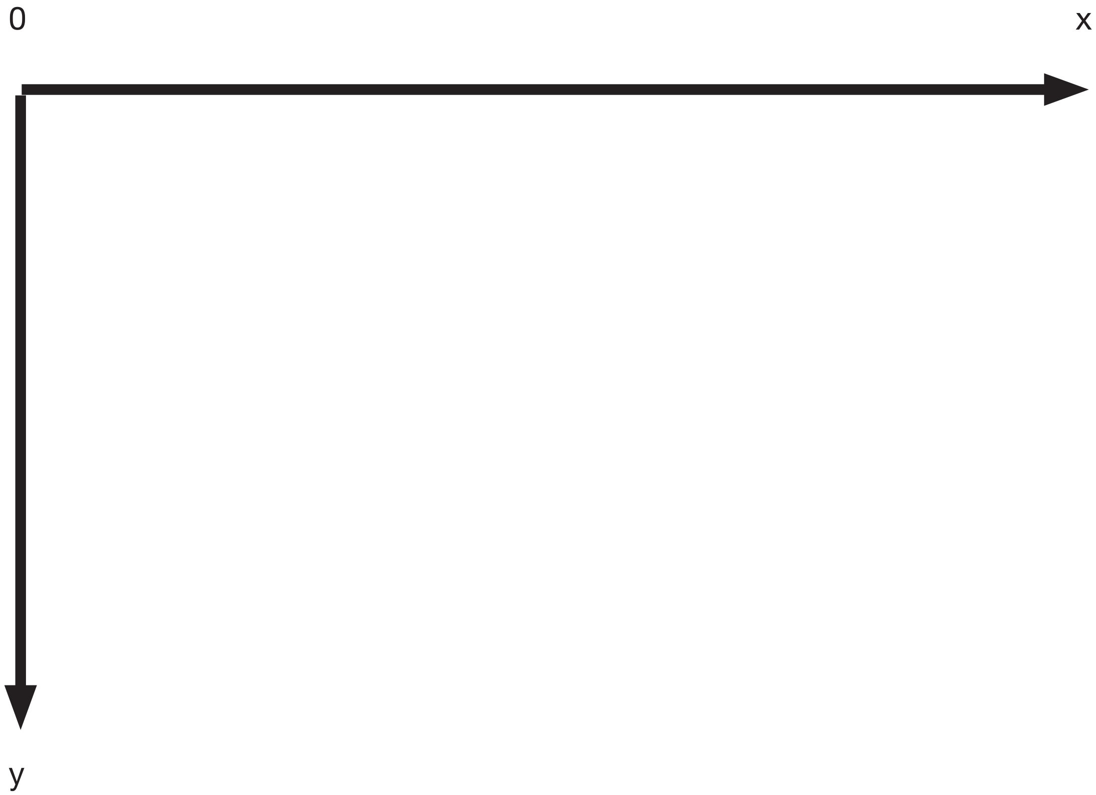
- Why is the input argument a function?
- What arguments does that function receive?
- What does the function do with these arguments?

1.4 <svg> DOM elements

	<circle>	<line>	<rect>	<text>	<path>	<g>
attr	cx cy r	x1 y1 x2 y2	x y width height	x y text	d	
	transform class					
style	fill fill-opacity stroke stroke-width stroke-opacity					

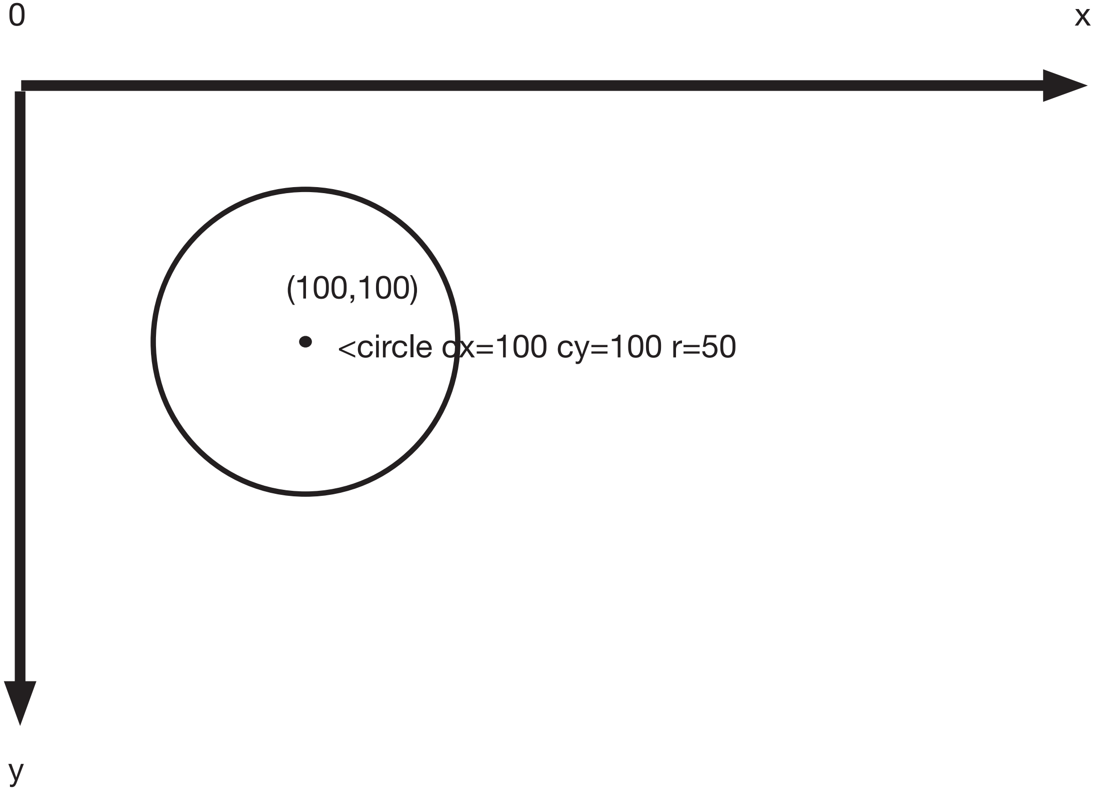
1.4 <svg> DOM elements: coordinate system

The grid system in <svg> works left to right, top to bottom



1.4 <svg> DOM elements: coordinate system

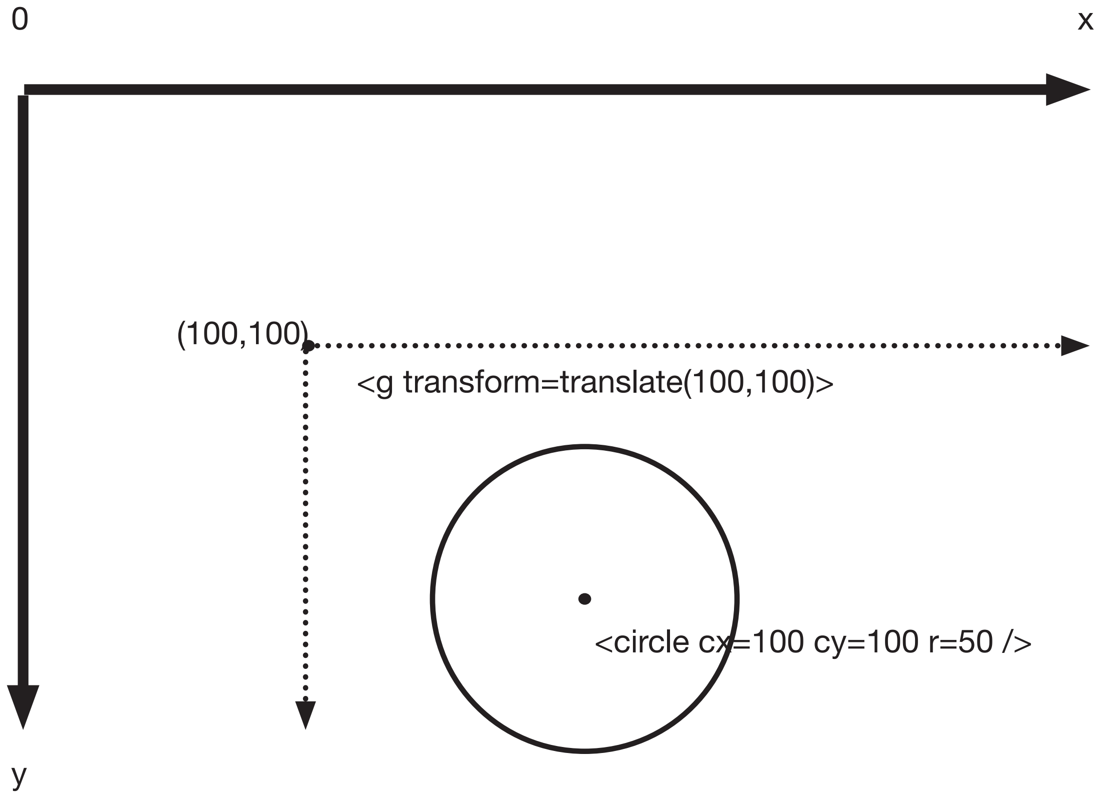
```
<svg>
    <circle ... />
</svg>
```



1.4 <svg> DOM elements: <g> element

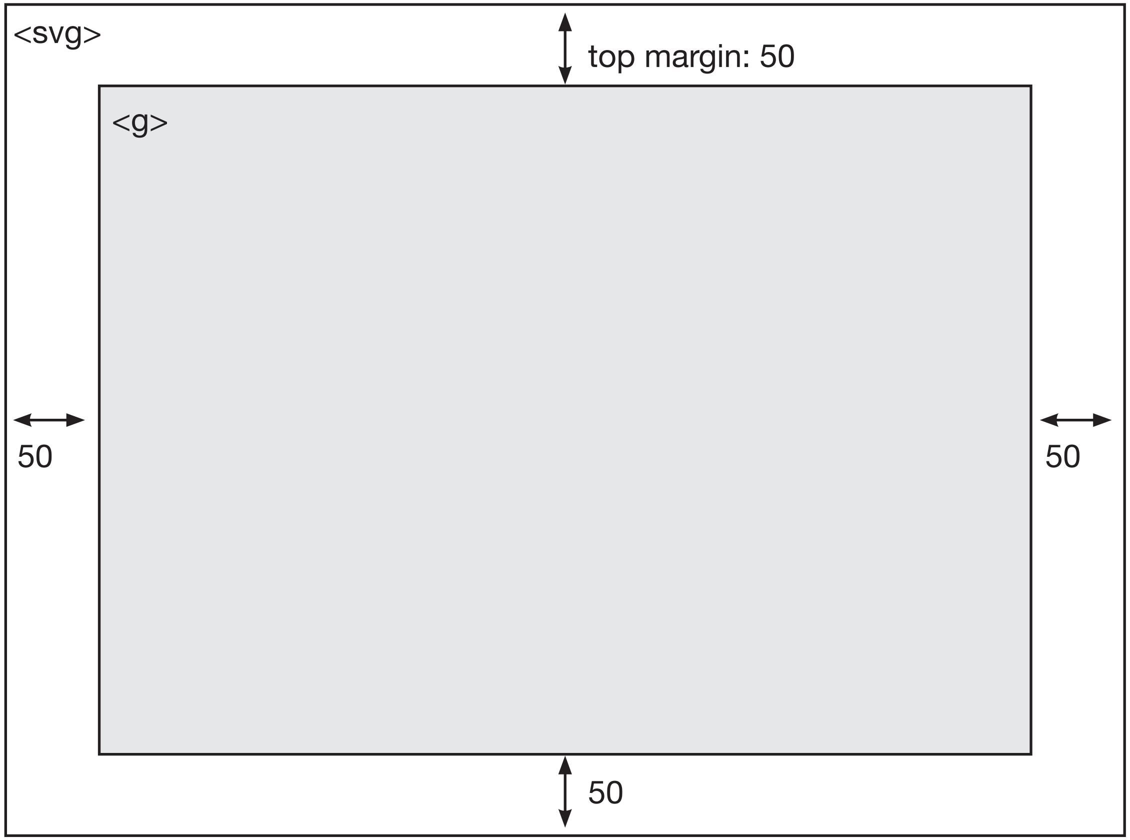
We use <g> to group individual elements; each <g> starts its own coordinate system.

In this example, we “translated” <g> by (100,100), so that the <circle> element is actually at (200,200) relative to the overall <svg>

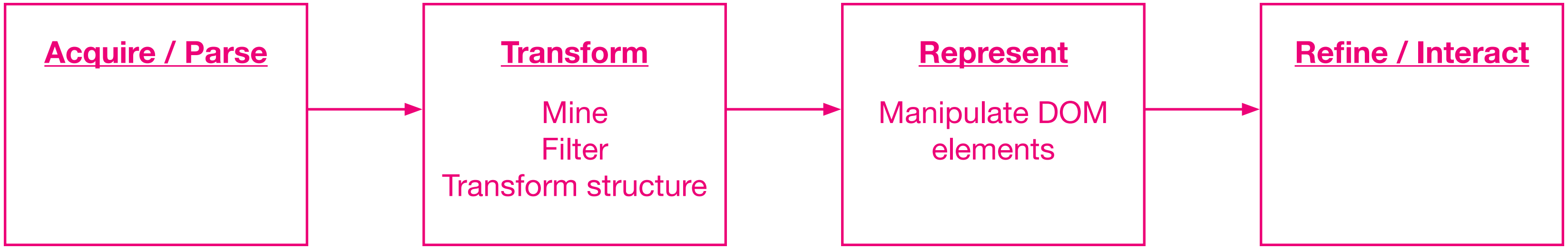


1.4 <svg> DOM elements: margin conventions

We often find it useful NOT to draw from the very edge of <svg>. Instead, we use a <g> to offset everything by a margin, so that we leave some margin between the drawing and the edges.



2. Data transformation with d3



Besides DOM manipulation, a large part of the d3 library deals with the manipulation of data.

We'll look at some fundamental building blocks of the `d3-math` module.

2. Data transformation with d3: min, max, mean

Given a simple array:

```
[3, 90, 87, 56, 90, 0, -8]
```

We can easily discover its min and max values as well as its average using:

```
d3.min(array)
d3.max(array)
d3.mean(array)
...
```

2. Data transformation with d3: min, max, mean

What if the array contains more complex values?

```
[
  {name: ‘Ashley’, age:30, tenure:2},
  {name: ‘Ben’, age: 33, tenure:5},
  {name: ‘Carol’, age:45, tenure:10}
]
```

Accessor pattern to the rescue:

```
d3.min(array, accessor)
d3.max(array, accessor)
d3.mean(array, accessor)
...
```

2. Data transformation with d3: accessor pattern

The accessor pattern “accesses”, and transforms, each element in the array.

This is a common pattern in array-related methods.

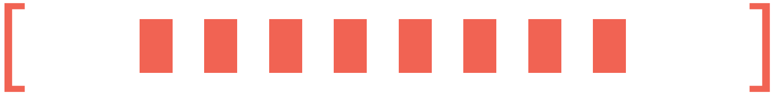
```
const avgAge = d3.min(array, function(d){
  return d.age;
});
```

2. Advanced data transformation: `d3-nest`

Groups like elements with like elements in an array, and creates a nested data structure.

3. Data transformation with arrays

Arrays and objects are fundamental data structures.



3.1 Array length and array index

Arrays, like other JavaScript objects, have properties. One key property is `.length`

```
>> var students = [‘Jessie’, ‘Audrey’,
‘Patrick’, ‘Andrew’];
>> console.log(students.length); //4
```

Individual elements of an array can be access using an index, starting from `0` and ending at `.length-1`, with

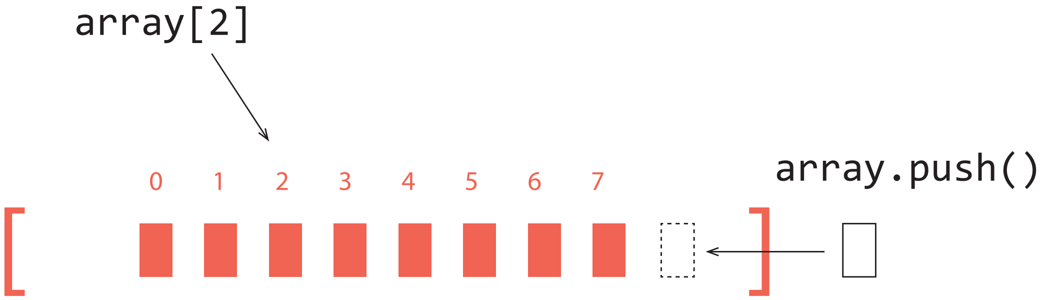
```
>> var students = [‘Jessie’, ‘Audrey’,
‘Patrick’, ‘Andrew’];
>> console.log(students[0]); // ‘Jessie’
>> console.log(students[3]); // ‘Andrew’
```

3.2 Adding elements to an array

Arrays, like other JavaScript objects, have methods. One key property is `.push()`, which adds a value to an array at the end

```
>> var students = [‘Jessie’, ‘Audrey’,
‘Patrick’, ‘Andrew’];
>> students.push(‘Nina’);
>> console.log(students[4]); // ‘Nina’
```

3.2 Adding elements to an array



3.3 Array methods

Iterating through arrays

`array.forEach()`

Per element transform

`array.map()`

Filter / sort

`array.filter()`
`array.sort()`

Other array methods: https://developer.mozilla.org/en-US/docs/Web/JavaScript/Reference/Global_Objects/Array

Putting everything together

In final exercise, let’s visualize the workings of `Math.random()`

Before you start, sketch out what this might look like. What choices are you making?

Recap

In the last exercise we encountered two typical considerations we tend to encounter in data visualization.

Visual encoding: what visual properties (position, shape, size, color) best express what we are trying to show.

Mapping domain to range: how do we effectively map numbers to screen coordinates?

A goal of this course is to help you develop better intuitions about how to address these considerations!

Recap

1. DOM manipulation: `d3-select`
    - What are selections?
    - Modifying and appending DOM elements
    - Iterating through selections
    - Working with `<svg>` elements
2. Data transform: `d3-math` and `d3-nest`
    - The accessor pattern
3. Data transform using arrays
4. Putting it all together