Week 10

ADVANCED LAYOUT

Review of the Week before Spring Break

We looked at d3 SVG generators and d3 layouts.

Raw data → Parse → Layout → Bind to DOM → Represent

Raw data	Parse	Layout	Bind to DOM	Represent

`d3.csv( )`
`d3.json( )`
`...`
		`d3.layout`
		`(or custom`
		`function)`
			`.data( )`
			`...`
				`d3.svg`

Generators

Collected under the `d3.svg` namespace

Generats the geometry description for complex `<path>` elements

Different svg generators expect input data of different formats. For example, `d3.svg.line()` expects some kind of serialized data, with a series of [x,y] coordinates.

Layout Functions

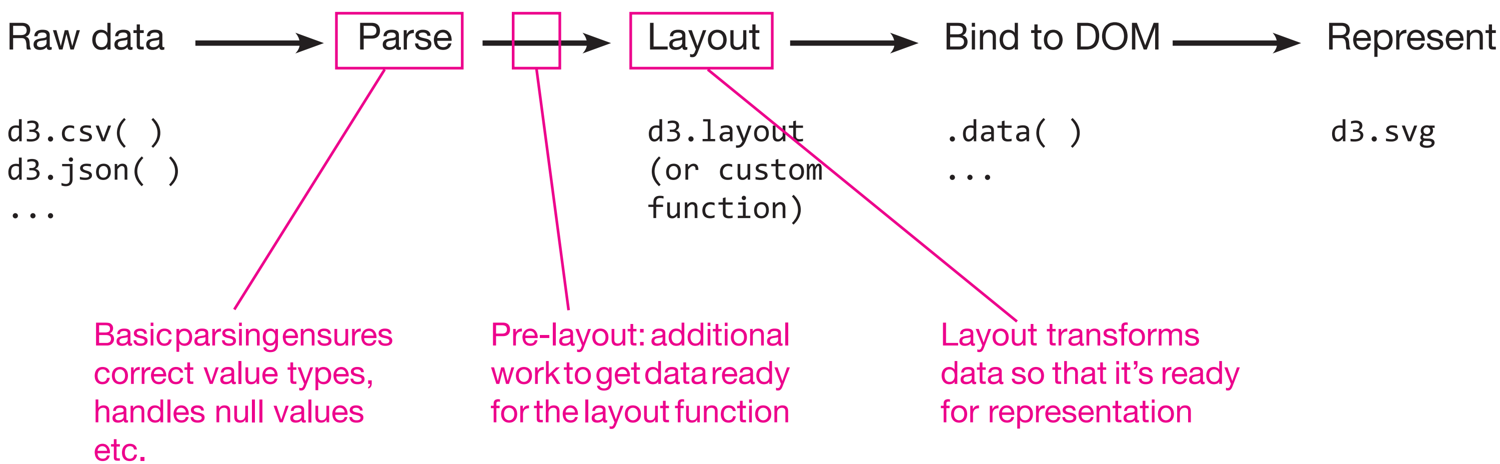
Collected under the `d3.layout` namespace

Does NOT literally "lay out" anything. It is about transforming the structure of data so that it can more readily be drawn

Frequently paired up with `d3.svg` generators. For example, `d3.layout.pie()` is frequently paired with `d3.svg.arc()`

How Does It All Fit Together?

Simple tabular data (such as from a .csv file) undergoes multiple transformations to its strucuture before it can be represented visually.



Example: Scatterplot

Raw data → Parse → Layout → Bind to DOM → Represent

```
d3.csv( )
d3.json( )
...
```

```
.data( )
...
```

```
var data = [
{x:100, y:300},
{x:400, y:230},
{x:500, y:800},
{x:600, y:932},
...
]
```

```
svg.selectAll('.point')
     .data(data)
     .enter()
     .append('circle')
     .attr('class', 'point')
     .attr('cx',function(d){
          return d.x;
     })
     .attr('cy',function(d){
          return d.y;
     })
     ...
```

Example: Line

1 Raw data → 2 Parse → Layout → 3 Bind to DOM → 4 Represent

```
d3.csv( )
d3.json( )
...
```

```
.data( )
...
```

```
d3.svg
```

3

```
svg.append('path')
     .datum(data)
     .attr('d',function(d){
          return lineGenerator(d);
     });
```

2

```
var data = [
{yr:100, v:300},
{yr:400, v:230},
{yr:500, v:800},
{yr:600, v:null},
...
]
```

4

```
var lineGenerator = d3.svg.line()
     .x(function(d){ return d.yr})
     .y(function(d){ return d.v })
     .defined(function(d){
          return d.yr && d.v;
     })
//returns description of path geometry
```

Example: Pie Chart

1 Raw data → 2 Parse → 3 Layout → 4 Bind to DOM → 5 Represent

```
d3.csv( )
d3.json( )
...
```

```
.data( )
...
```

```
d3.svg
```

2 Post-parse, pre-layout

```
var data = [
{yr:100, v:300},
{yr:400, v:230},
{yr:500, v:800},
{yr:600, v:null},
...
]
```

3 Run it through layout

```
var pieLayout = d3.layout.pie()
     .value(function(d){
     return d.yr;
     //based on what attribute should
pie layout calculate slice sizes
     });

var newData = pieLayout(data);
```

Post-layout data

```
console.log(newData);
[
{value:100, startAngle:0, endAngle:1.33443},
{value:400, startAngle:1.33443, endAngle:6.2333}
...
]
```

Example: Pie Chart

1 Raw data → 2 Parse → 3 Layout → 4 Bind to DOM → 5 Represent

Raw data:
```
d3.csv( )
d3.json( )
...
```

Bind to DOM:
```
.data( )
...
```

Represent:
```
d3.svg
```

4 Bind to DOM

```
svg.selectAll('path')
     .data(newData)
     .enter()
     .append('path')
     .attr('d', function(d){
          return arc(d);
     });
```

5 SVG generator

```
var arc = d3.svg.arc()
     .innerRadius(100)
     .outerRadius(500)
     .startAngle(function(d){
          return d.startAngle;
     })
     .endAngle(function(d){
          return d.endAngle
     });
```

Post-layout data

```
console.log(newData);
[
{value:100, startAngle:0, endAngle:1.33443},
{value:400, startAngle:1.33443, endAngle:6.2333}
...
]
```

An important challenge is to keep track of data structure through multiple transformations.

Let’s look at Exercise 1.

Problem: Teasing Structure Out of .csv Data

By definition, tabular .csv data is based on a linear, one-dimensional format. It's difficult to represent multi-dimensional, particularly hierarchical relationships.

A practical example: apple production in `n` countries over the course of `m` years.

We have `n x m` data points.

Problem: Teasing Structure Out of .csv Data

This can be represented as a `n` rows times `m` columns:

country	1990	1991	...
Albania	1000	1005	...
Algeria	2000	3000	...
...	...	...	...

Problem: Teasing Structure Out of .csv Data

Or frequently, `n x m` rows and 1 column:

country	year	value
Albania	1990	1000
Albania	1991	1005
...	...	...
Algeria	1990	2000
Algeria	1991	3000
...	...	...

Problem: Teasing Structure Out of .csv Data

In the latter case, the imported data doesn't lend itself to a serial representation (as a line).

```
[
    {country:Albania, year:1990, value:1000},
    {country:Albania, year:1991, value:1005},
    ...

]
```

flat array with nxm elements

How can we gather up all the data points belonging to one country into the same array easily?

d3.nest()

`d3.nest()` produces a nesting function. The nesting function takes in a large, flat array, and group array elements based on a common key.

```
[
    {country:Albania, year:1990, value:1000},
    {country:Albania, year:1991, value:1005},
    ...
    {country:Algeria, year:1990, value:2000},
    {country:Algeria, year:1991, value:3000}
    ...
]
```

d3.nest()

“key” to nesting

```
[
     {country:Albania, year:1990, value:1000},
     {country:Albania, year:1991, value:1005},
     ...
     {country:Algeria, year:1990, value:2000},
     {country:Algeria, year:1991, value:3000}
     ...
]
```

flat array with nxm elements

becomes:

outer array with n elements

```
[
     {key:Albania,
     values:[
          {country:Albania, year:1990, value:1000},
          {country:Albania, year:1991, value:1005},
          ...]
     }
]
```

Inner array with m elements

d3.nest()

```
var nest = d3.nest()
    .key(function(d){
        return d.country;
    });

var nestedData = nest.entries(data);
```

Exercise 2: Using the same data, how can we visualize the composition of apple production by country in any given year?

Problem: Reducing the Dimensionality of Data

In the previous example, we teased a flat array into a `n x m` structure. Now, we have to reduce `n x m` down to `n`

For this, we can use the `.rollup()` method of `d3.nest()`

d3.nest().rollup()

```
var nest = d3.nest()
    .key(function(d){
        return d.country;
    })
    .rollup(function(leaves){
        //take all the nested elements,
        //Combine them in some way
        //return a single value
    });

var nestedData = nest.entries(data);
```

Aside: d3.map() as a Lookup Table

We can also take advantage of the d3.map() data structure, which is similiar to a look-up dictionary.

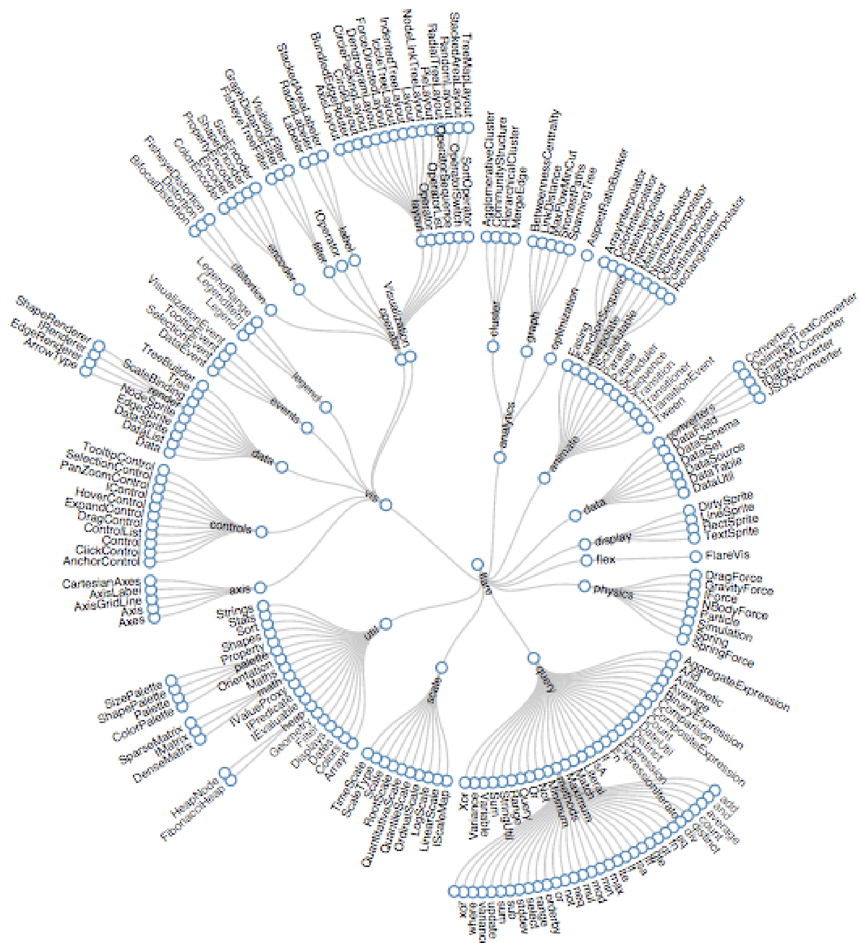
```
var lookup = d3.map();
lookup.set(“dog”, “chien”);
lookup.get(“dog”); //chien;
```

d3.nest().rollup()

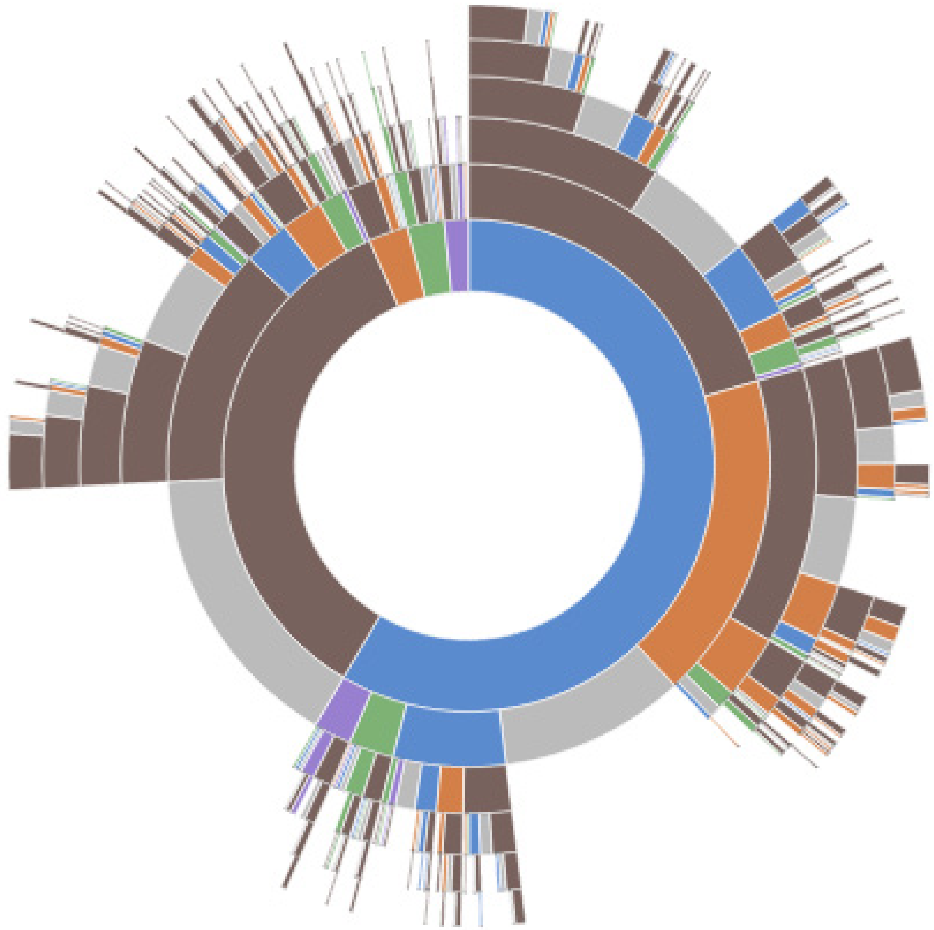
```
var nest = d3.nest()
    .key(function(d){
        return d.country;
    })
    .rollup(function(leaves){
        var dataSeries = d3.map();

        //...
        return dataSeries;
    });

var nestedData = nest.entries(data);
```

Exercise 3: Creating Hierarchy Out of Data

HIERARCHICAL LAYOUTS

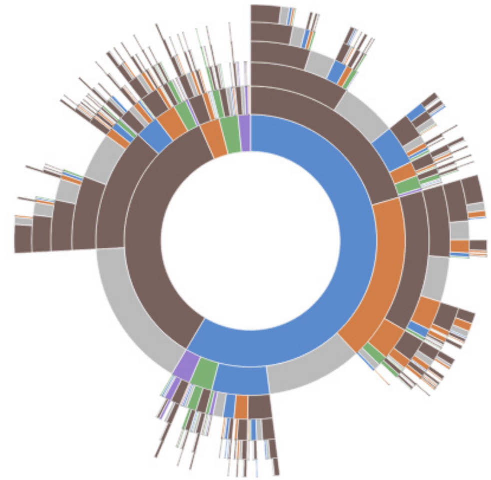


Tree



Parition

Pack

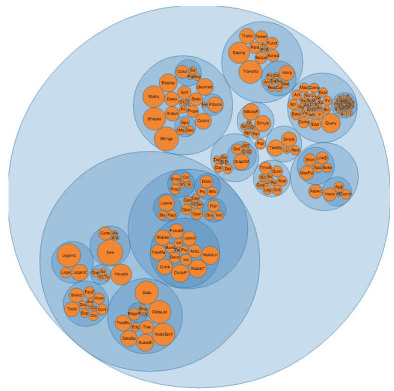


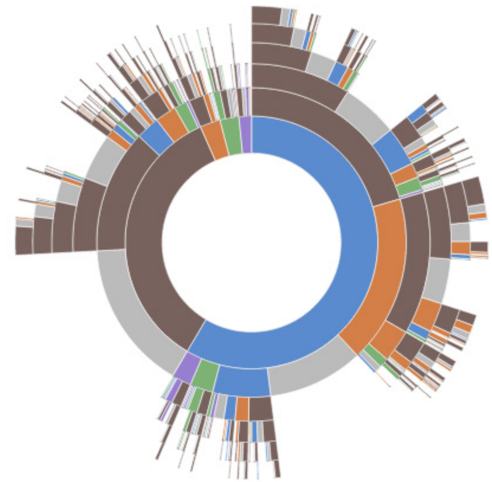
<u>How are they the same?</u>

They assume that the input, pre-layout data has a recurrent, hierarchical structure;

Both layout functions need to know how to navigate the hierarchical strucuture: i.e. starting from the top, how to find children nodes;

Both layout functions need to know how to size each node i.e. size by raw count, or some other attribute?





How are they different?

One is visually represented with svg <circle> elements; the other with <path> elements in conjunction with `d3.svg.arc( )`

1 Raw data → 2 Parse → 3 Layout → 4 Bind to DOM → 5 Represent

2 Post-parse, pre-layout

```
var root = {
  name: “parent”,
  children: [
    {name: “sibling1”, v:23},
    {
     name: “sibling2”,
     children:[
      {name: “pet1”,v:3},
      {name: “pet1”,v:3}
     ]
    }
  ]

}
```

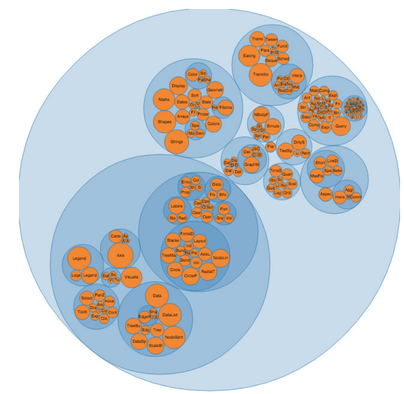
3 Layout function

```
var pack = d3.layout.pack()
     .children(function(d){
          return d.children;
     })
     .value(function(d){
          return d.v;
     })
     .size([width,height]);
```

1 Raw data → 2 Parse → 3 Layout → 4 Bind to DOM → 5 Represent

2 Post-parse, pre-layout

```
var root = {
  name: “parent”,
  children: [
    {name: “sibling1”, v:23},
    {
     name: “sibling2”,
     children:[
      {name: “pet1”,v:3},
      {name: “pet1”,v:3}
     ]
    }
  ]

}
```



3 Layout function

```
var pack = d3.layout.pack()
      .children(function(d){
            return d.children;
      })
      .value(function(d){
            return d.v;
      })
      .size([width,height]);
```

1 Raw data → 2 Parse → 3 Layout → 4 Bind to DOM → 5 Represent

What does post-layout data look like?

Try it out and inspect it in console

How do we represent this data?