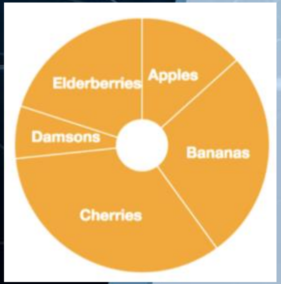
Layout

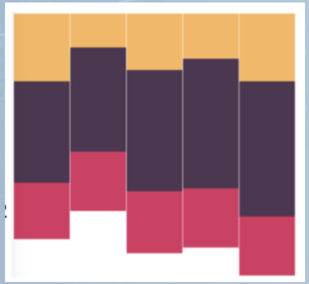
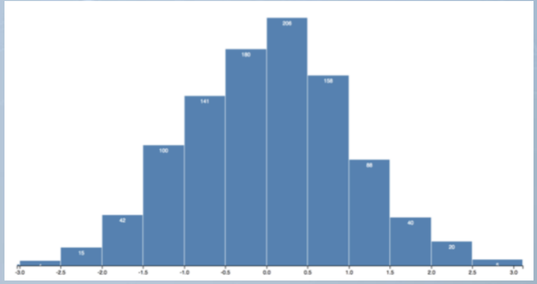
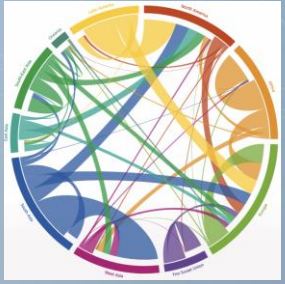
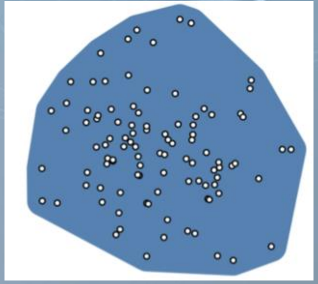
Data Visualization

Layouts

- In essence, a layout function in D3 is just a JavaScript function that
  - Takes your data as input
  - Computes visual variables such as position size to it so that we can visualize the data

- Pie, stack bar, histogram, chord, convex hull



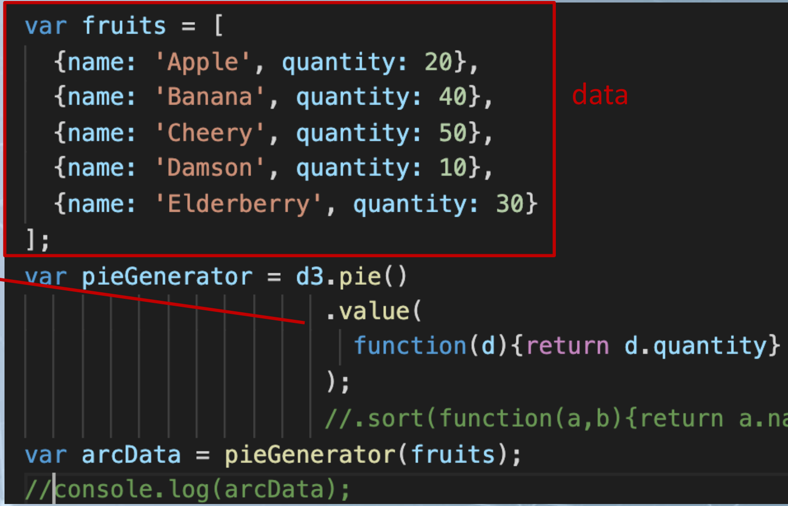





Pie Generator: d3.pie()

- Given an array of data, the pie generator computes the necessary angles to represent the data
  - Output an array of objects containing the original data augmented by **start** and **end angles** which can be used to draw a pie chart by d3.arc()

Pie Generator (Ex10-01)

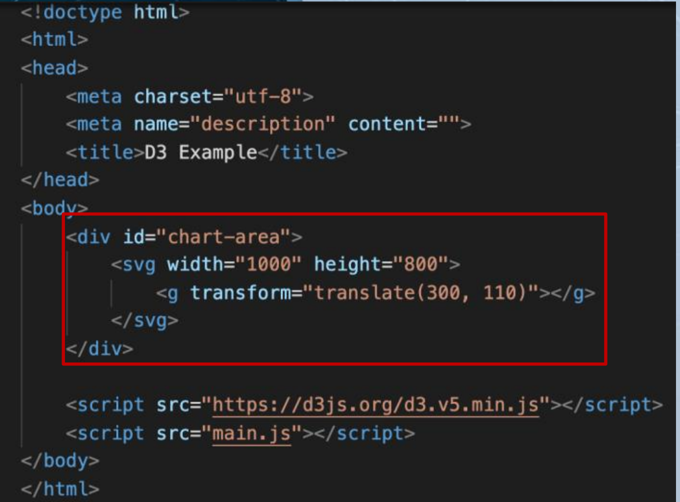
- Tell d3.pie() that to get data values from 'quantity' attribute of each elements in the data array



```
var fruits = [
  {name: 'Apple', quantity: 20},
  {name: 'Banana', quantity: 40},
  {name: 'Cheery', quantity: 50},
  {name: 'Damson', quantity: 10},
  {name: 'Elderberry', quantity: 30}
];
var pieGenerator = d3.pie()
                          .value(
                            function(d){return d.quantity}
                          );
                          //.sort(function(a,b){return a.na
var arcData = pieGenerator(fruits);
//console.log(arcData);
```

data

Index.html

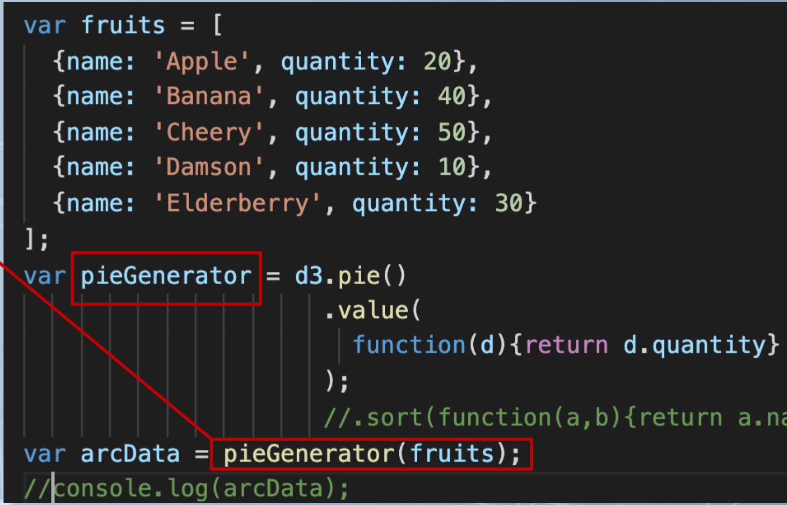


```
<!doctype html>
<html>
<head>
    <meta charset="utf-8">
    <meta name="description" content="">
    <title>D3 Example</title>
</head>
<body>
    <div id="chart-area">
        <svg width="1000" height="800">
            <g transform="translate(300, 110)"></g>
        </svg>
    </div>

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

I have created a 'svg' and a 'g' in index.html

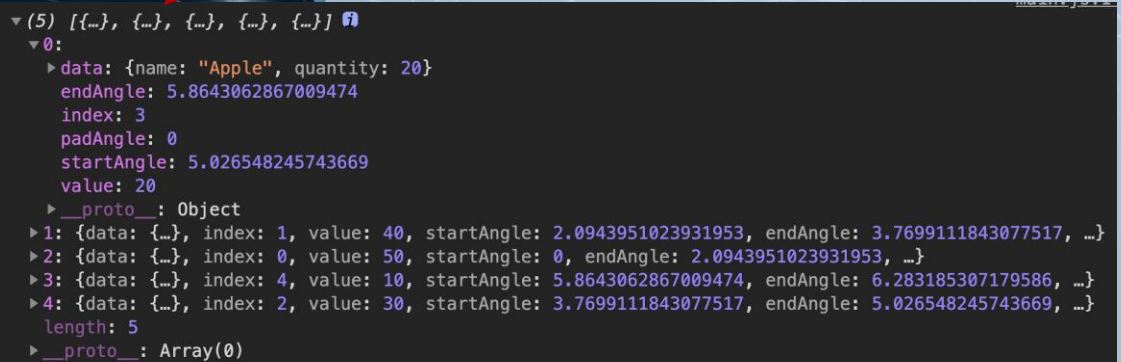
Pie Generator (Ex10-01)

- Send our data array to the pie generator
- ‘arcData’ stores the information for d3.arc() to draw the pie chart



```
var fruits = [
  {name: 'Apple', quantity: 20},
  {name: 'Banana', quantity: 40},
  {name: 'Cheery', quantity: 50},
  {name: 'Damson', quantity: 10},
  {name: 'Elderberry', quantity: 30}
];
var pieGenerator = d3.pie()
                     .value(
                       function(d){return d.quantity}
                     );
                     //.sort(function(a,b){return a.na
var arcData = pieGenerator(fruits);
//console.log(arcData);
```

If we print it out



```
▼(5) [{…}, {…}, {…}, {…}, {…}]
 ▼0:
  ▶data: {name: "Apple", quantity: 20}
   endAngle: 5.8643062867009474
   index: 3
   padAngle: 0
   startAngle: 5.026548245743669
   value: 20
  ▶__proto__: Object
 ▶1: {data: {…}, index: 1, value: 40, startAngle: 2.0943951023931953, endAngle: 3.7699111843077517, …}
 ▶2: {data: {…}, index: 0, value: 50, startAngle: 0, endAngle: 2.0943951023931953, …}
 ▶3: {data: {…}, index: 4, value: 10, startAngle: 5.8643062867009474, endAngle: 6.283185307179586, …}
 ▶4: {data: {…}, index: 2, value: 30, startAngle: 3.7699111843077517, endAngle: 5.026548245743669, …}
  length: 5
 ▶__proto__: Array(0)
```

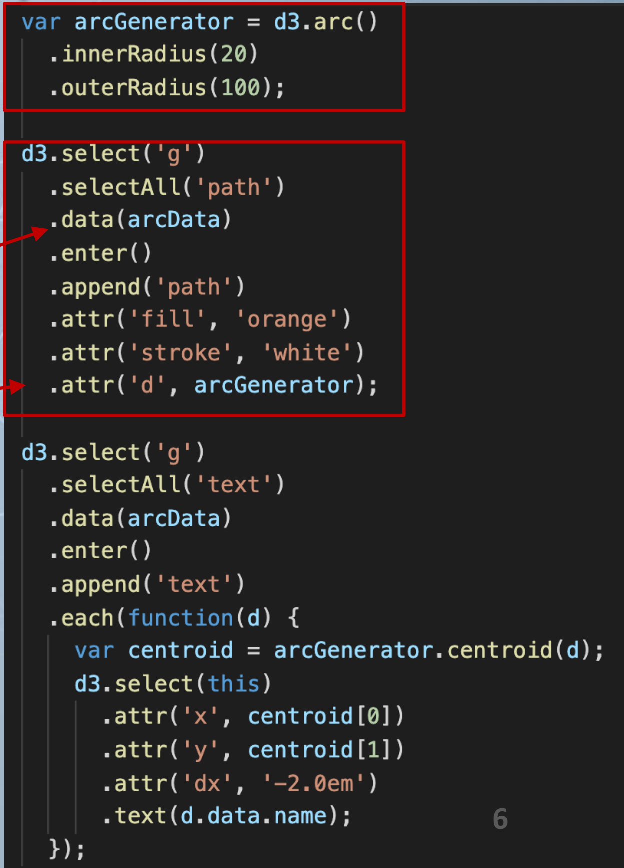
Pie Generator (Ex10-01)

- After you get the 'arcs' information from pie chart generator, the rest of work is the same as Ex05-07 (draw multiple arcs)

We also need the arc generator here to draw arcs

Bind the arcs information calculated from pie chart generator

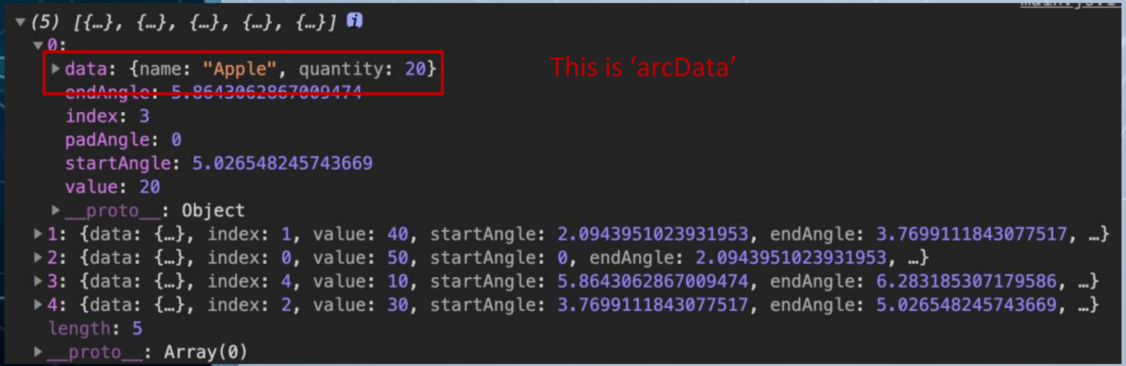
Data element is fed to arcGenerator to generate path information



```
var arcGenerator = d3.arc()
  .innerRadius(20)
  .outerRadius(100);

d3.select('g')
  .selectAll('path')
  .data(arcData)
  .enter()
  .append('path')
  .attr('fill', 'orange')
  .attr('stroke', 'white')
  .attr('d', arcGenerator);

d3.select('g')
  .selectAll('text')
  .data(arcData)
  .enter()
  .append('text')
  .each(function(d) {
    var centroid = arcGenerator.centroid(d);
    d3.select(this)
      .attr('x', centroid[0])
      .attr('y', centroid[1])
      .attr('dx', '-2.0em')
      .text(d.data.name);
  });
```

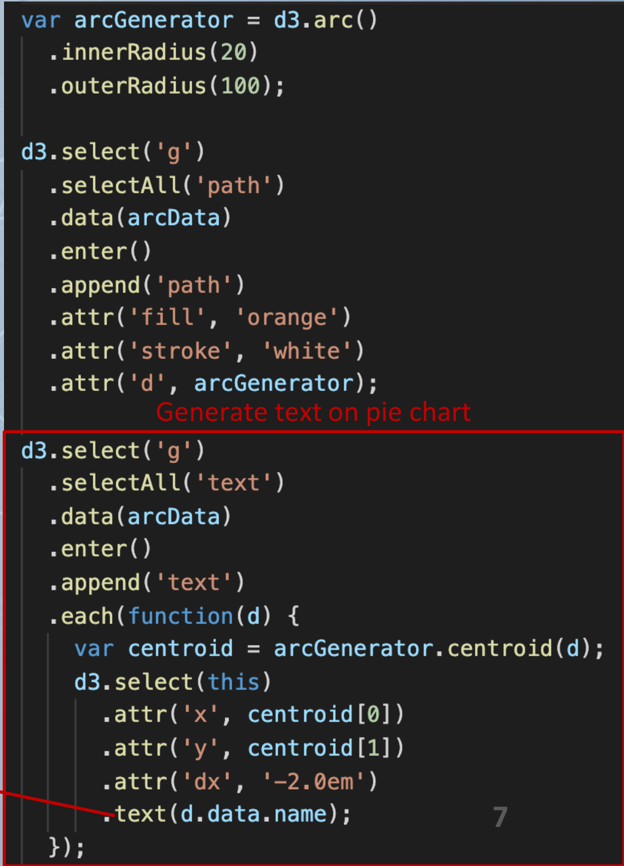
Pie Generator (Ex10-01)

- After you get the 'arcs' information from pie chart generator, the rest of work is the same as Ex05-07 (draw multiple arcs)



```
▼(5) [{…}, {…}, {…}, {…}, {…}]
  ▼0:
    ▶data: {name: "Apple", quantity: 20}
     endAngle: 5.8643062867009474
     index: 3
     padAngle: 0
     startAngle: 5.026548245743669
     value: 20
    ▶__proto__: Object
  ▶1: {data: {…}, index: 1, value: 40, startAngle: 2.0943951023931953, endAngle: 3.7699111843077517, …}
  ▶2: {data: {…}, index: 0, value: 50, startAngle: 0, endAngle: 2.0943951023931953, …}
  ▶3: {data: {…}, index: 4, value: 10, startAngle: 5.8643062867009474, endAngle: 6.283185307179586, …}
  ▶4: {data: {…}, index: 2, value: 30, startAngle: 3.7699111843077517, endAngle: 5.026548245743669, …}
   length: 5
  ▶__proto__: Array(0)
```

This is 'arcData'



```
var arcGenerator = d3.arc()
  .innerRadius(20)
  .outerRadius(100);

d3.select('g')
  .selectAll('path')
  .data(arcData)
  .enter()
  .append('path')
  .attr('fill', 'orange')
  .attr('stroke', 'white')
  .attr('d', arcGenerator);
```

Generate text on pie chart

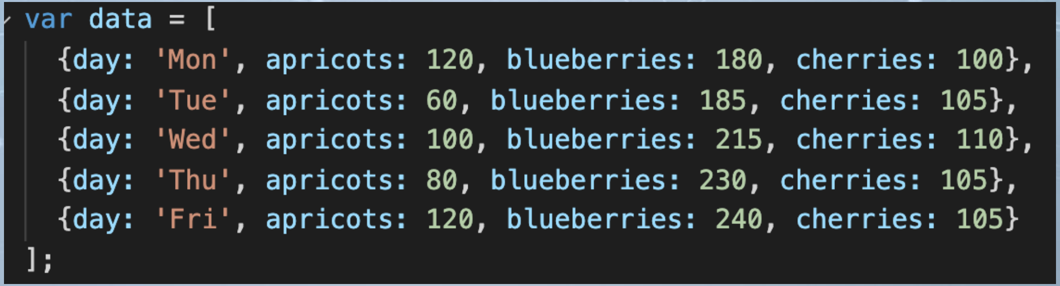
```
d3.select('g')
  .selectAll('text')
  .data(arcData)
  .enter()
  .append('text')
  .each(function(d) {
    var centroid = arcGenerator.centroid(d);
    d3.select(this)
      .attr('x', centroid[0])
      .attr('y', centroid[1])
      .attr('dx', '-2.0em')
      .text(d.data.name);
  });
```

The way to get 'name' from arcData

Stack Generator

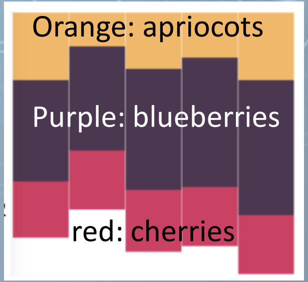
- A classic visualization from stack generator is stack bars
- They are used to show how a larger categories is divided into smaller categories and what the relationship of each part has on the total amount

different fruits sales volume from Mon. to Fri. (Ex10-02)



```
var data = [
  {day: 'Mon', apricots: 120, blueberries: 180, cherries: 100},
  {day: 'Tue', apricots: 60, blueberries: 185, cherries: 105},
  {day: 'Wed', apricots: 100, blueberries: 215, cherries: 110},
  {day: 'Thu', apricots: 80, blueberries: 230, cherries: 105},
  {day: 'Fri', apricots: 120, blueberries: 240, cherries: 105}
];
```

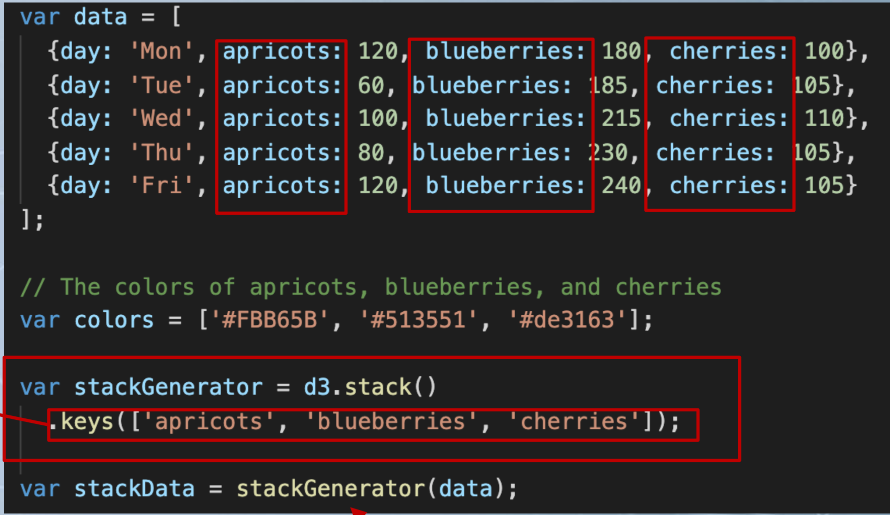
Column
Mon. ~ Fri.



d3.stack (Ex10-02)

- In order to send the data to d3.stack(), the data should be array of dictionaries.
  - Each element in the order will be stacked together

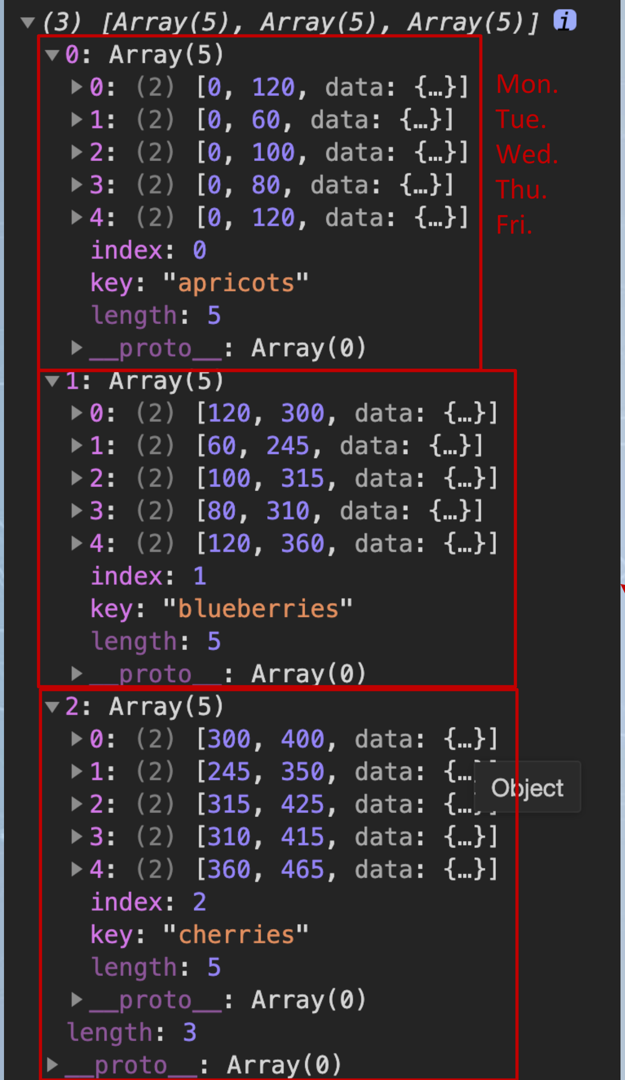
- Send the array of ‘key’ to d3.stack().keys()
  - Let d3.stack() know what values should be used to create the stacks



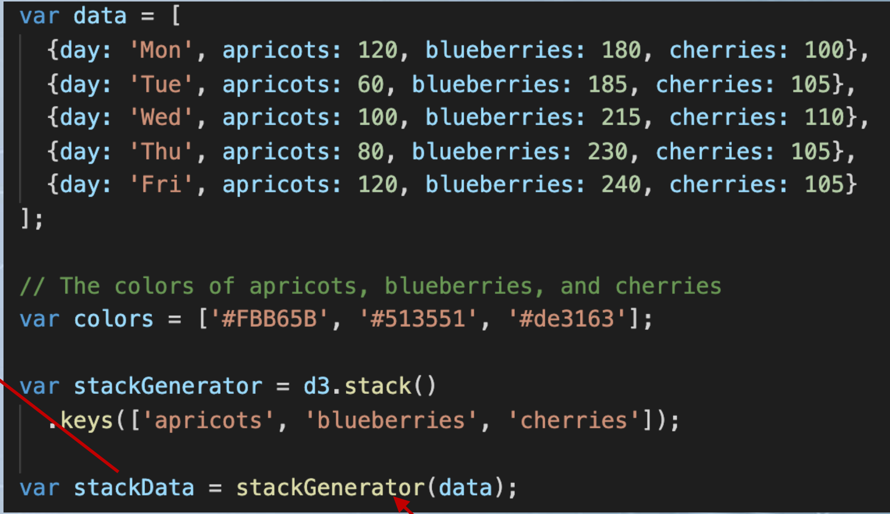
```
var data = [
  {day: 'Mon', apricots: 120, blueberries: 180, cherries: 100},
  {day: 'Tue', apricots: 60, blueberries: 185, cherries: 105},
  {day: 'Wed', apricots: 100, blueberries: 215, cherries: 110},
  {day: 'Thu', apricots: 80, blueberries: 230, cherries: 105},
  {day: 'Fri', apricots: 120, blueberries: 240, cherries: 105}
];

// The colors of apricots, blueberries, and cherries
var colors = ['#FBB65B', '#513551', '#de3163'];

var stackGenerator = d3.stack()
  .keys(['apricots', 'blueberries', 'cherries']);

var stackData = stackGenerator(data);
```

Send the data to the stack generator. It will return the information about how to layout the stacks

d3.stack (Ex10-02)



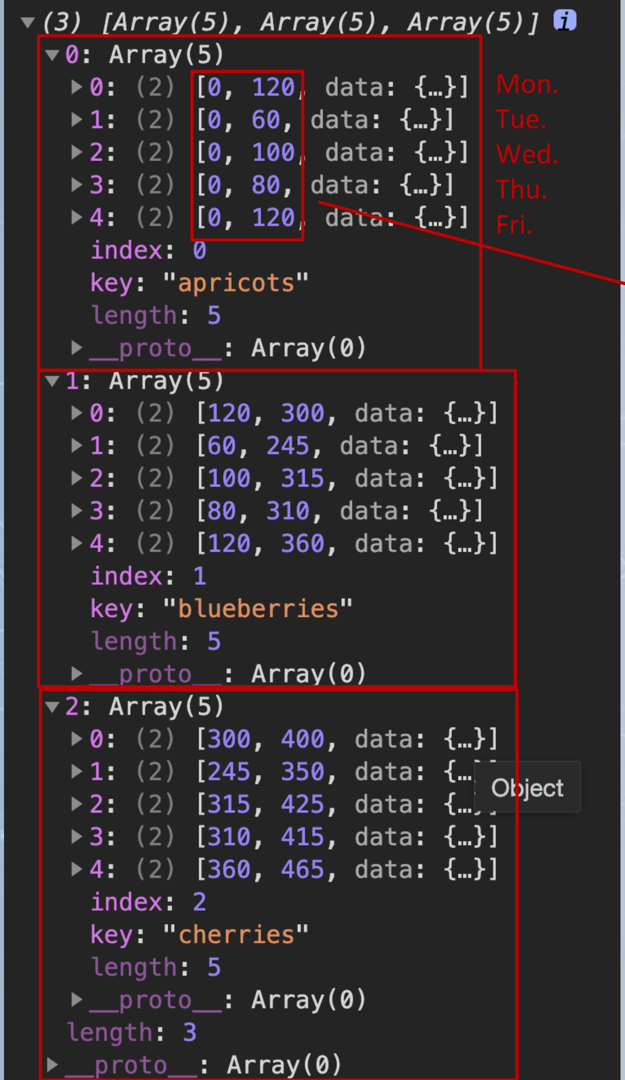
```
▼(3) [Array(5), Array(5), Array(5)]
  ▼0: Array(5)
    ▶0: (2) [0, 120, data: {…}]
    ▶1: (2) [0, 60, data: {…}]
    ▶2: (2) [0, 100, data: {…}]
    ▶3: (2) [0, 80, data: {…}]
    ▶4: (2) [0, 120, data: {…}]
     index: 0
     key: "apricots"
     length: 5
    ▶__proto__: Array(0)
  ▼1: Array(5)
    ▶0: (2) [120, 300, data: {…}]
    ▶1: (2) [60, 245, data: {…}]
    ▶2: (2) [100, 315, data: {…}]
    ▶3: (2) [80, 310, data: {…}]
    ▶4: (2) [120, 360, data: {…}]
     index: 1
     key: "blueberries"
     length: 5
    ▶__proto__: Array(0)
  ▼2: Array(5)
    ▶0: (2) [300, 400, data: {…}]
    ▶1: (2) [245, 350, data: {…}]
    ▶2: (2) [315, 425, data: {…}]
    ▶3: (2) [310, 415, data: {…}]
    ▶4: (2) [360, 465, data: {…}]
     index: 2
     key: "cherries"
     length: 5
    ▶__proto__: Array(0)
   length: 3
  ▶__proto__: Array(0)
```



```
var data = [
  {day: 'Mon', apricots: 120, blueberries: 180, cherries: 100},
  {day: 'Tue', apricots: 60, blueberries: 185, cherries: 105},
  {day: 'Wed', apricots: 100, blueberries: 215, cherries: 110},
  {day: 'Thu', apricots: 80, blueberries: 230, cherries: 105},
  {day: 'Fri', apricots: 120, blueberries: 240, cherries: 105}
];

// The colors of apricots, blueberries, and cherries
var colors = ['#FBB65B', '#513551', '#de3163'];

var stackGenerator = d3.stack()
  .keys(['apricots', 'blueberries', 'cherries']);

var stackData = stackGenerator(data);
```

Send the data to the stack generator. It will return the information about how to layout the stacks

d3.stack (Ex10-02)



```
▼(3) [Array(5), Array(5), Array(5)]
 ▼0: Array(5)
   ▶0: (2) [0, 120, data: {…}]
   ▶1: (2) [0, 60, data: {…}]
   ▶2: (2) [0, 100, data: {…}]
   ▶3: (2) [0, 80, data: {…}]
   ▶4: (2) [0, 120, data: {…}]
    index: 0
    key: "apricots"
    length: 5
   ▶__proto__: Array(0)
 ▼1: Array(5)
   ▶0: (2) [120, 300, data: {…}]
   ▶1: (2) [60, 245, data: {…}]
   ▶2: (2) [100, 315, data: {…}]
   ▶3: (2) [80, 310, data: {…}]
   ▶4: (2) [120, 360, data: {…}]
    index: 1
    key: "blueberries"
    length: 5
   ▶__proto__: Array(0)
 ▼2: Array(5)
   ▶0: (2) [300, 400, data: {…}]
   ▶1: (2) [245, 350, data: {…}]
   ▶2: (2) [315, 425, data: {…}]
   ▶3: (2) [310, 415, data: {…}]
   ▶4: (2) [360, 465, data: {…}]
    index: 2
    key: "cherries"
    length: 5
   ▶__proto__: Array(0)
  length: 3
 ▶__proto__: Array(0)
```

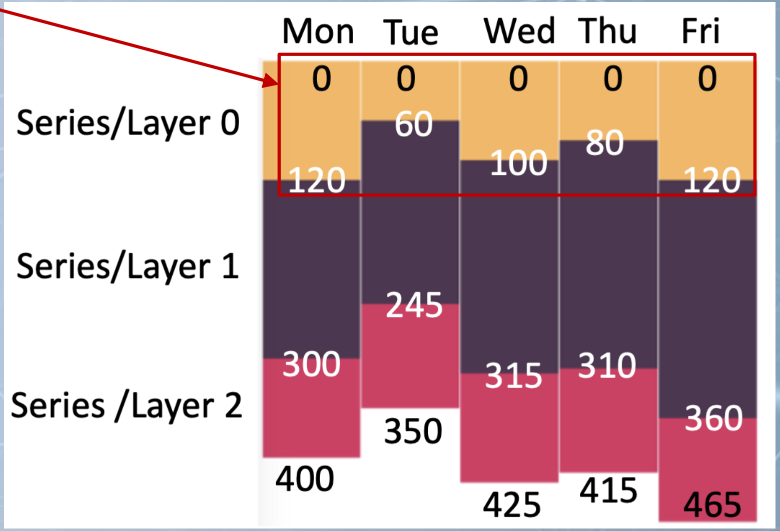
apricots

blueberries

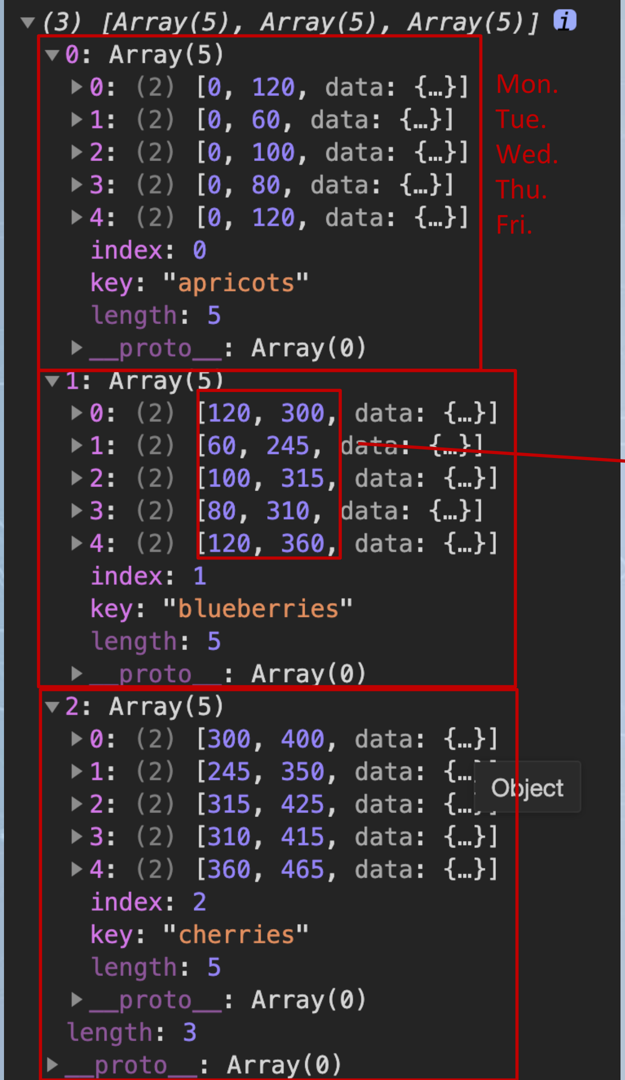
chrries

Mon.
Tue.
Wed.
Thu.
Fri.

Object



d3.stack (Ex10-02)



```
▼(3) [Array(5), Array(5), Array(5)]
  ▼0: Array(5)
    ▶0: (2) [0, 120, data: {…}]
    ▶1: (2) [0, 60, data: {…}]
    ▶2: (2) [0, 100, data: {…}]
    ▶3: (2) [0, 80, data: {…}]
    ▶4: (2) [0, 120, data: {…}]
     index: 0
     key: "apricots"
     length: 5
    ▶__proto__: Array(0)
  ▼1: Array(5)
    ▶0: (2) [120, 300, data: {…}]
    ▶1: (2) [60, 245, data: {…}]
    ▶2: (2) [100, 315, data: {…}]
    ▶3: (2) [80, 310, data: {…}]
    ▶4: (2) [120, 360, data: {…}]
     index: 1
     key: "blueberries"
     length: 5
    ▶__proto__: Array(0)
  ▼2: Array(5)
    ▶0: (2) [300, 400, data: {…}]
    ▶1: (2) [245, 350, data: {…}]
    ▶2: (2) [315, 425, data: {…}]
    ▶3: (2) [310, 415, data: {…}]
    ▶4: (2) [360, 465, data: {…}]
     index: 2
     key: "cherries"
     length: 5
    ▶__proto__: Array(0)
   length: 3
  ▶__proto__: Array(0)
```

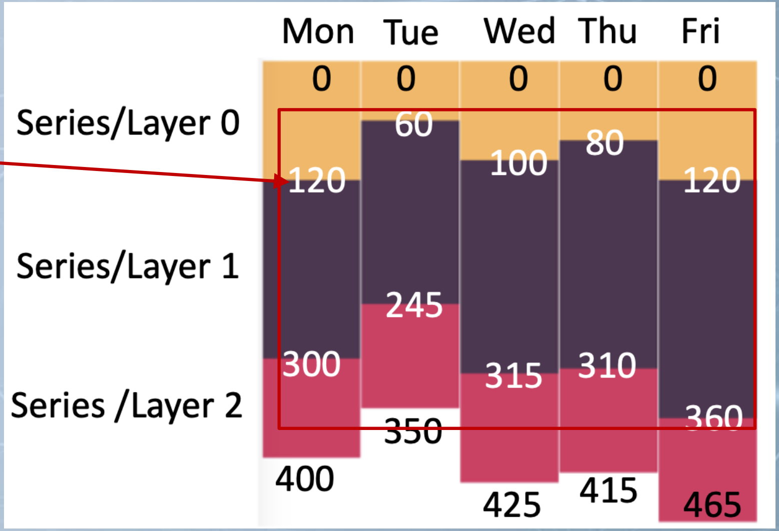
apricots

Mon.
Tue.
Wed.
Thu.
Fri.

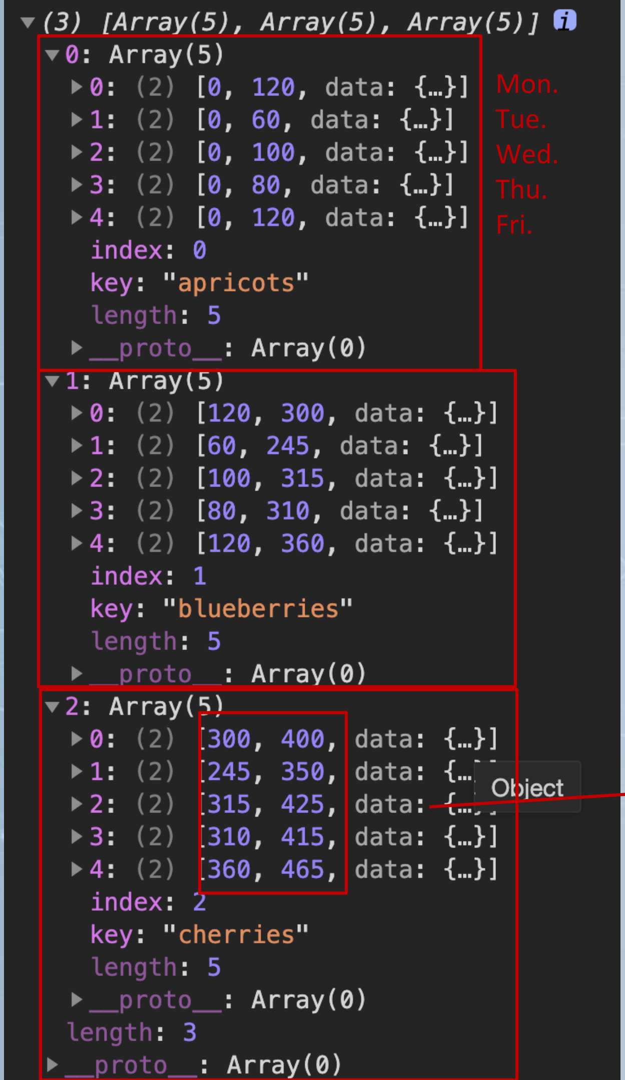
blueberries

chrries

Object



d3.stack (Ex10-02)



```
▼(3) [Array(5), Array(5), Array(5)]
  ▼0: Array(5)
    ▶0: (2) [0, 120, data: {…}]
    ▶1: (2) [0, 60, data: {…}]
    ▶2: (2) [0, 100, data: {…}]
    ▶3: (2) [0, 80, data: {…}]
    ▶4: (2) [0, 120, data: {…}]
     index: 0
     key: "apricots"
     length: 5
    ▶__proto__: Array(0)
  ▼1: Array(5)
    ▶0: (2) [120, 300, data: {…}]
    ▶1: (2) [60, 245, data: {…}]
    ▶2: (2) [100, 315, data: {…}]
    ▶3: (2) [80, 310, data: {…}]
    ▶4: (2) [120, 360, data: {…}]
     index: 1
     key: "blueberries"
     length: 5
    ▶__proto__: Array(0)
  ▼2: Array(5)
    ▶0: (2) [300, 400, data: {…}]
    ▶1: (2) [245, 350, data: {…
    ▶2: (2) [315, 425, data: {…}]
    ▶3: (2) [310, 415, data: {…}]
    ▶4: (2) [360, 465, data: {…}]
     index: 2
     key: "cherries"
     length: 5
    ▶__proto__: Array(0)
   length: 3
  ▶__proto__: Array(0)
```

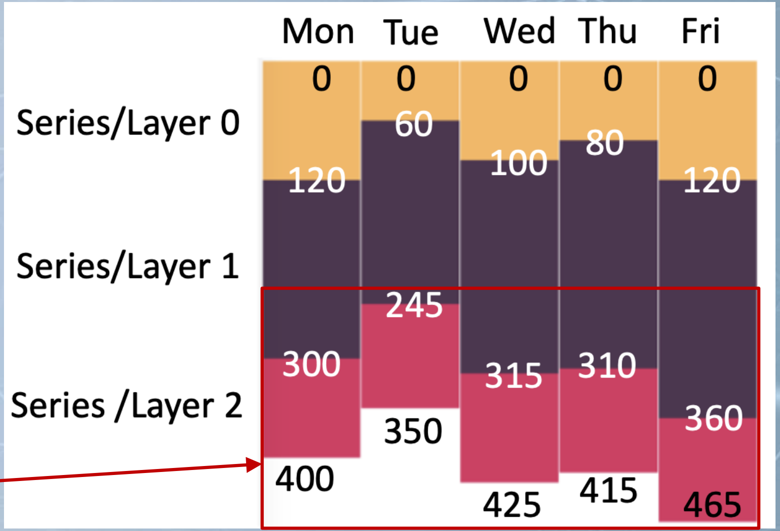
apricots

blueberries

chrries

Mon.
Tue.
Wed.
Thu.
Fri.

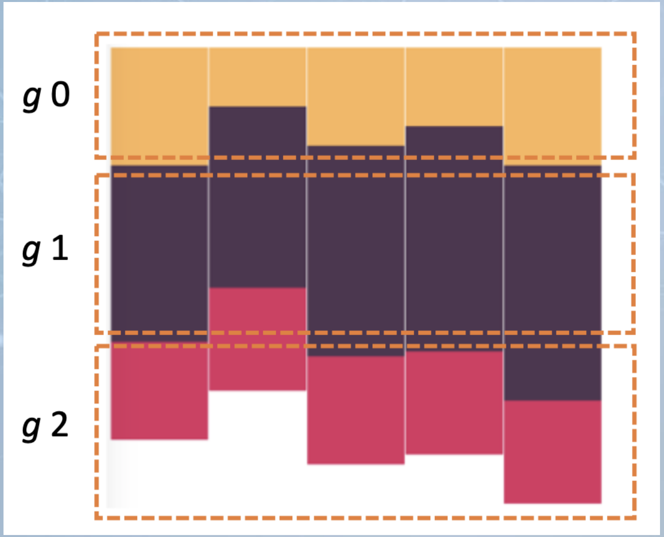
Object

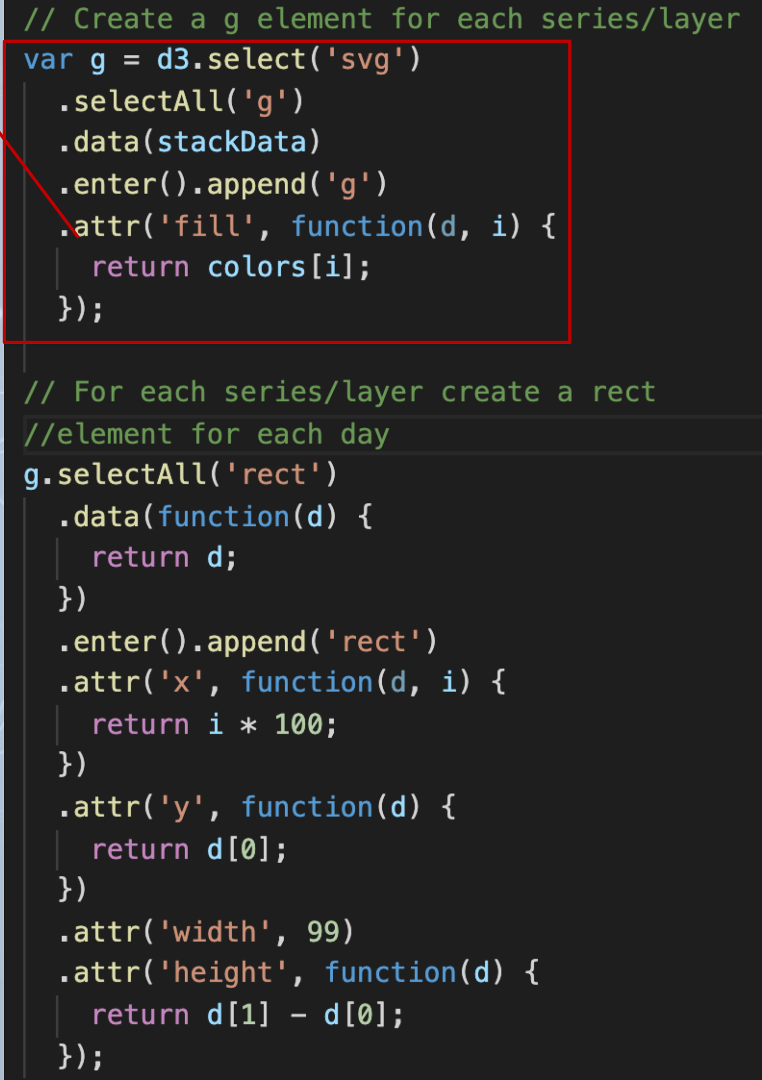


d3.stack (Ex10-02)

Which color to fill the shape in a <g>

- Create ‘g’ for each fruits
- So, the variable ‘var g’ contains three groups

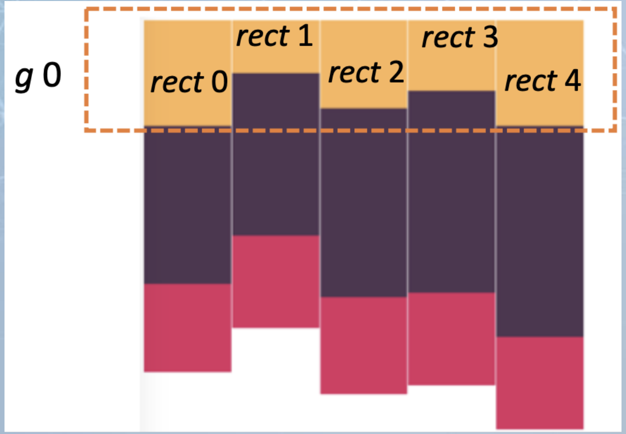




```
// Create a g element for each series/layer
var g = d3.select('svg')
  .selectAll('g')
  .data(stackData)
  .enter().append('g')
  .attr('fill', function(d, i) {
    return colors[i];
  });

// For each series/layer create a rect
//element for each day
g.selectAll('rect')
  .data(function(d) {
    return d;
  })
  .enter().append('rect')
  .attr('x', function(d, i) {
    return i * 100;
  })
  .attr('y', function(d) {
    return d[0];
  })
  .attr('width', 99)
  .attr('height', function(d) {
    return d[1] - d[0];
  });
```

d3.stack (Ex10-02)

- For each <g> tag, the data is an array computed by stackGenerator
  - E.g. the data [[0,120], [0,60], [0,100], [0,80], [0,120]] is attached to the first <g> in "var g"

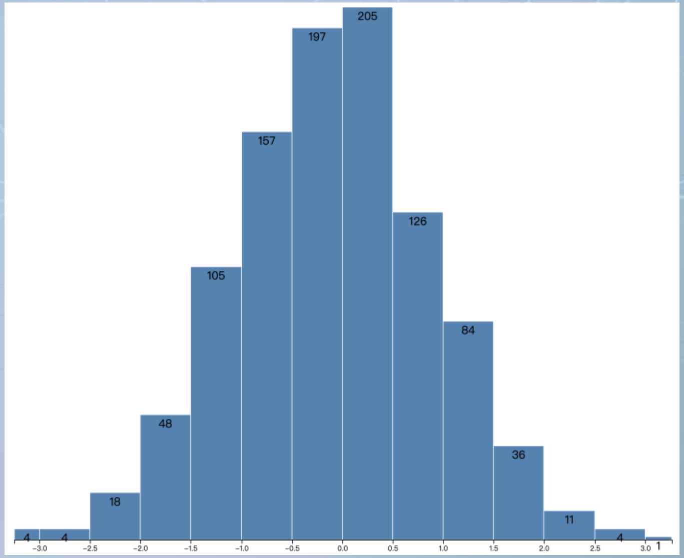


```
// Create a g element for each series/layer
var g = d3.select('svg')
  .selectAll('g')
  .data(stackData)
  .enter().append('g')
  .attr('fill', function(d, i) {
    return colors[i];
  });

// For each series/layer create a rect
//element for each day
g.selectAll('rect')
  .data(function(d) {
    return d;
  })
  .enter().append('rect')
  .attr('x', function(d, i) {
    return i * 100;
  })
  .attr('y', function(d) {
    return d[0];
  })
  .attr('width', 99)
  .attr('height', function(d) {
    return d[1] - d[0];
  });
```

Because 'g' has three <g>, this statement will go through the three <g> and the corresponding data to create rectangles

Histogram

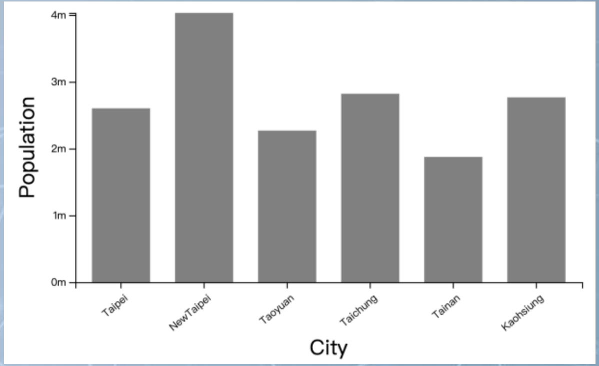
- Histograms bin many discrete samples into a smaller number of consecutive, non-overlapping intervals
  - They are often used to visualize the distribution of numerical data



Draw Histogram From Data Points

- Recall how to draw bars (Ex04-16)
- In Ex04-16, our data array directly store the height (simply map from the population) to draw the bars

- Now, our data array stores the data points. We have to divide the data domain into multiple bins and calculate how many data point fail into each bin
  - d3.histogram can help us for this calculation

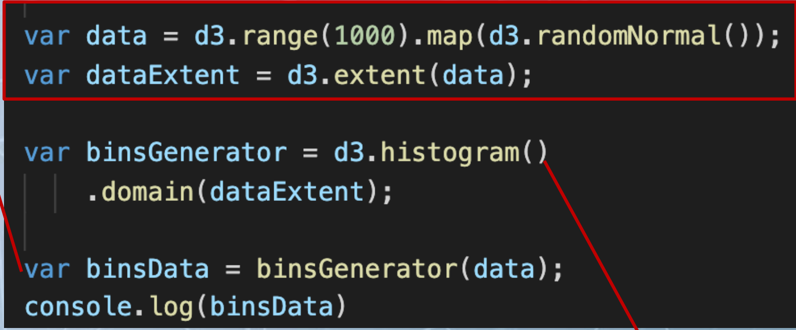


Ex04-16

Draw Histogram From Data Points (Ex10-3)

- Pass data to the histogram generator
- It will return you each bin 's lower bound (x0), upper bound(x1) and number of elements in that bin (length)

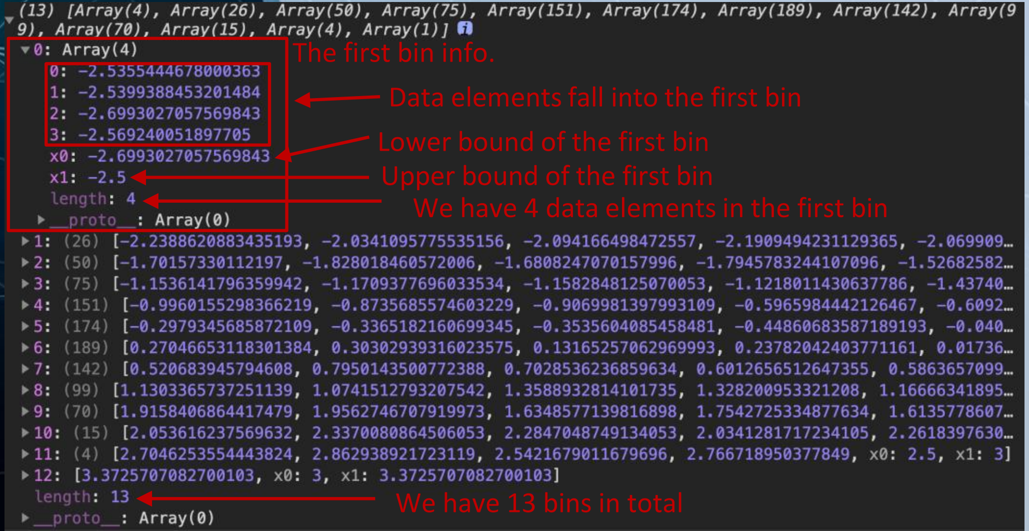
Generate data with 1000 data points and get the min and max values



```
var data = d3.range(1000).map(d3.randomNormal());
var dataExtent = d3.extent(data);

var binsGenerator = d3.histogram()
    .domain(dataExtent);

var binsData = binsGenerator(data);
console.log(binsData)
```

Create the histogram generator function and give it the domain range

binData



```
▼(13) [Array(4), Array(26), Array(50), Array(75), Array(151), Array(174), Array(189), Array(142), Array(9
9), Array(70), Array(15), Array(4), Array(1)]
 ▼0: Array(4)
    0: -2.5355444678000363
    1: -2.5399388453201484
    2: -2.6993027057569843
    3: -2.569240051897705
    x0: -2.6993027057569843
    x1: -2.5
    length: 4
  ▶__proto__: Array(0)
 ▶1: (26) [-2.2388620883435193, -2.0341095775535156, -2.094166498472557, -2.1909494231129365, -2.069909…
 ▶2: (50) [-1.70157330112197, -1.828018460572006, -1.6808247070157996, -1.7945783244107096, -1.52682582…
 ▶3: (75) [-1.1536141796359942, -1.1709377696033534, -1.1582848125070053, -1.1218011430637786, -1.43740…
 ▶4: (151) [-0.9960155298366219, -0.8735685574603229, -0.9069981397993109, -0.5965984442126467, -0.6092…
 ▶5: (174) [-0.2979345685872109, -0.3365182160699345, -0.3535604085458481, -0.44860683587189193, -0.040…
 ▶6: (189) [0.27046653118301384, 0.30302939316023575, 0.13165257062969993, 0.23782042403771161, 0.01736…
 ▶7: (142) [0.520683945794608, 0.7950143500772388, 0.7028536236859634, 0.6012656512647355, 0.5863657099…
 ▶8: (99) [1.1303365737251139, 1.0741512793207542, 1.3588932814101735, 1.328200953321208, 1.16666341895…
 ▶9: (70) [1.9158406864417479, 1.9562746707919973, 1.6348577139816898, 1.7542725334877634, 1.6135778607…
 ▶10: (15) [2.053616237569632, 2.3370080864506053, 2.2847048749134053, 2.0341281717234105, 2.2618397630…
 ▶11: (4) [2.7046253554443824, 2.862938921723119, 2.5421679011679696, 2.766718950377849, x0: 2.5, x1: 3]
 ▶12: [3.3725707082700103, x0: 3, x1: 3.3725707082700103]
  length: 13
 ▶__proto__: Array(0)
```

The first bin info.

Data elements fall into the first bin

Lower bound of the first bin

Upper bound of the first bin

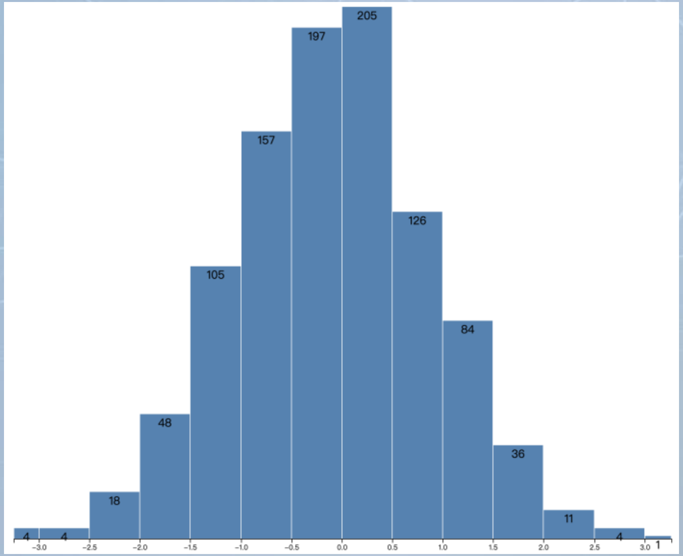
We have 4 data elements in the first bin

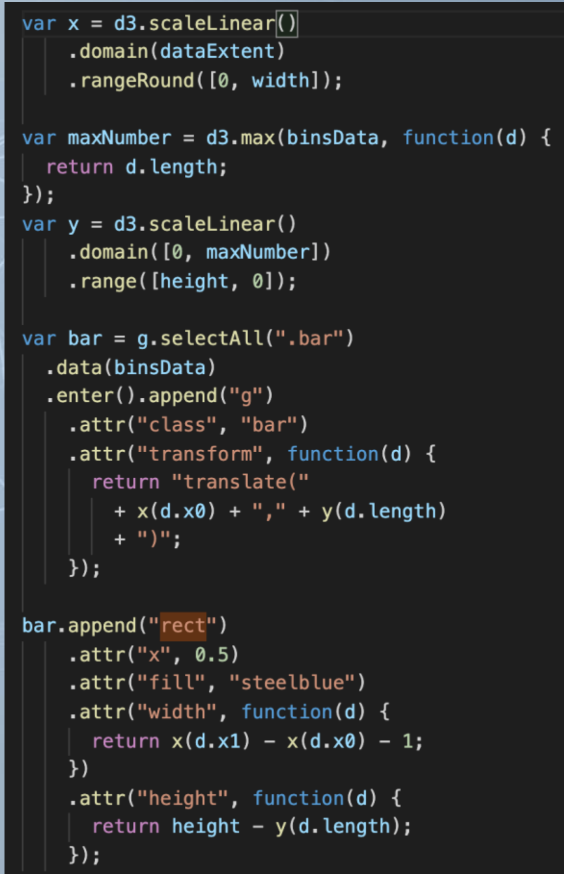
We have 13 bins in total

It can automatically determine a proper number of bins form you or you can specify it

Draw Histogram From Data Points (Ex10-3)

- After you know the lower bound, upper bound, counts of each bin, the following step is almost the same as draw a bar chart

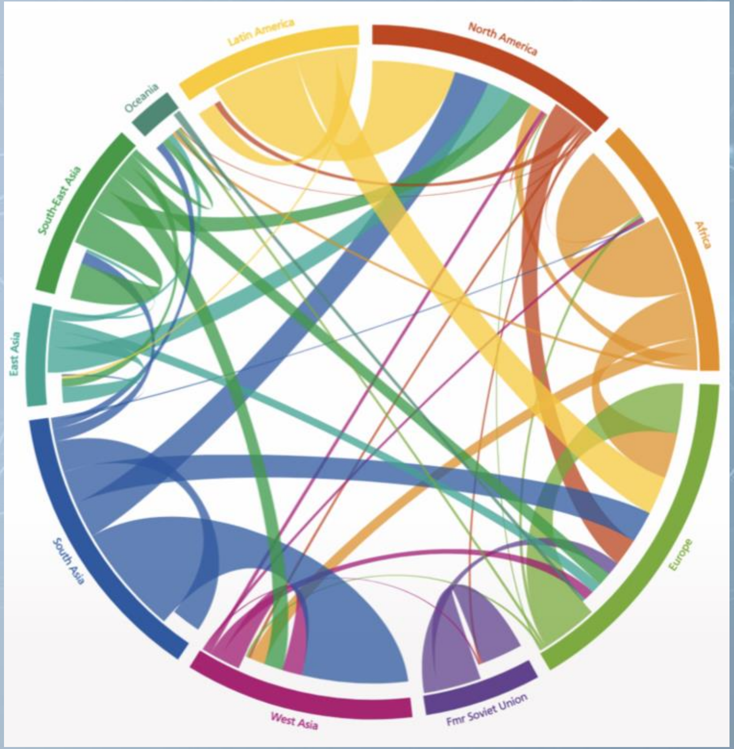




```
var x = d3.scaleLinear()
    .domain(dataExtent)
    .rangeRound([0, width]);

var maxNumber = d3.max(binsData, function(d) {
  return d.length;
});
var y = d3.scaleLinear()
    .domain([0, maxNumber])
    .range([height, 0]);

var bar = g.selectAll(".bar")
  .data(binsData)
  .enter().append("g")
    .attr("class", "bar")
    .attr("transform", function(d) {
      return "translate("
        + x(d.x0) + "," + y(d.length)
        + ")";
    });

bar.append("rect")
    .attr("x", 0.5)
    .attr("fill", "steelblue")
    .attr("width", function(d) {
      return x(d.x1) - x(d.x0) - 1;
    })
    .attr("height", function(d) {
      return height - y(d.length);
    });
```

Chord

- Chord diagrams visualize links (or flows) between a group of nodes, where each flow has a numeric value.
- Example
  - Migration flow between and within regions

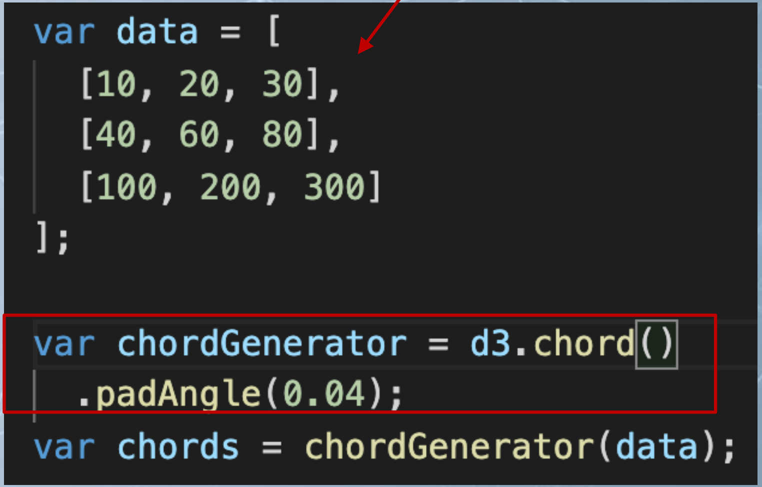


Not our code example!

Chord Generator: d3.chord()

- d3.chord()
  - Compute **startAngle** and **endAngle** of each data item
  - .padAngle(): set padding angle (gaps) between adjacent group

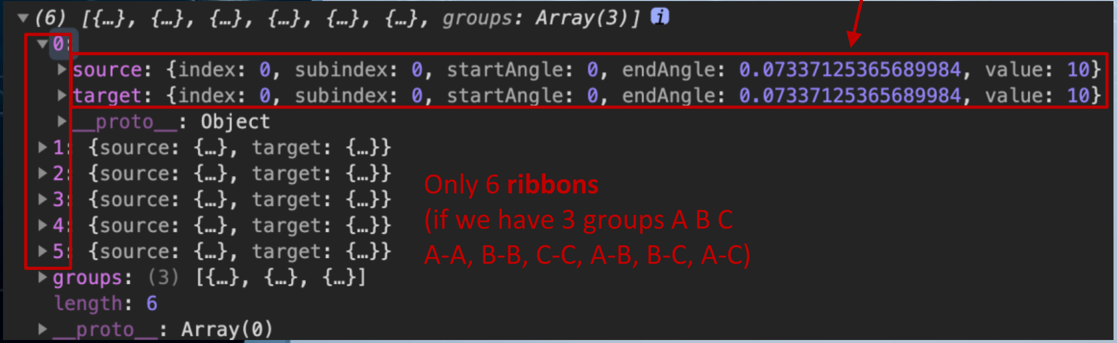
A 2D matrix to describe the transition among 3 groups



```
var data = [
    [10, 20, 30],
    [40, 60, 80],
    [100, 200, 300]
];

var chordGenerator = d3.chord()
    .padAngle(0.04);
var chords = chordGenerator(data);
```

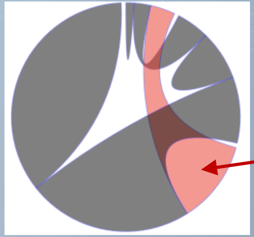
chords

Information of a ribbon



```
▼(6) [{…}, {…}, {…}, {…}, {…}, {…}, groups: Array(3)] i
  ▼0:
    ▶source: {index: 0, subindex: 0, startAngle: 0, endAngle: 0.07337125365689984, value: 10}
    ▶target: {index: 0, subindex: 0, startAngle: 0, endAngle: 0.07337125365689984, value: 10}
    ▶__proto__: Object
  ▶1: {source: {…}, target: {…}}
  ▶2: {source: {…}, target: {…}}
  ▶3: {source: {…}, target: {…}}
  ▶4: {source: {…}, target: {…}}
  ▶5: {source: {…}, target: {…}}
  ▶groups: (3) [{…}, {…}, {…}]
   length: 6
  ▶__proto__: Array(0)
```

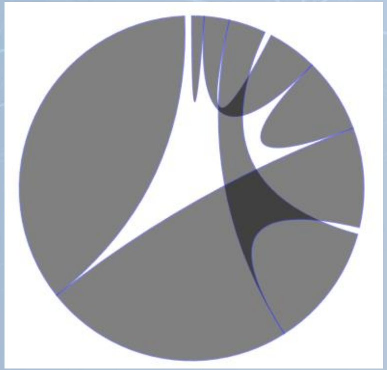
Only 6 **ribbons**
(if we have 3 groups A B C
A-A, B-B, C-C, A-B, B-C, A-C)

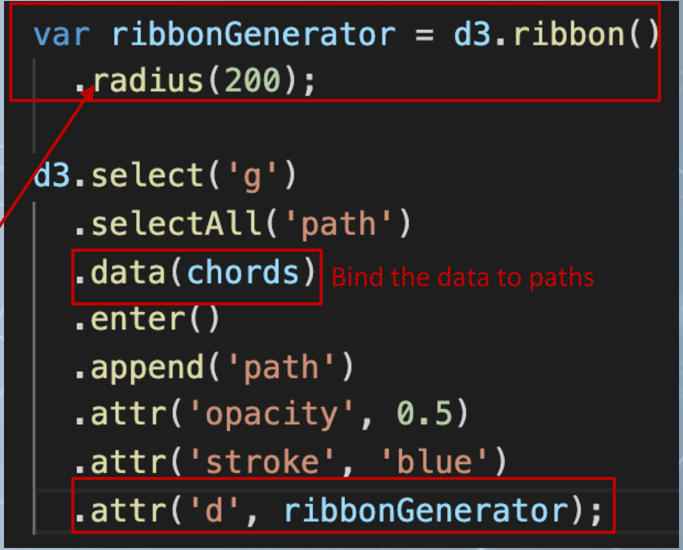


This is a ribbon

Ribbon Generator: d3.ribbon()

- After we have ribbons’ information, we can use ribbon generator (d3.ribbon()) to generate paths and draw
- d3.ribbon()
  - Converts the chord properties (startAngle and endAngle into path data so that we can draw chord by SVG
  - .radius(): control the radius of the final layout





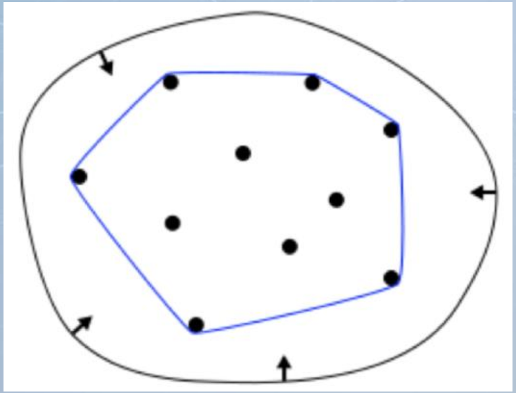
```
var ribbonGenerator = d3.ribbon()
  .radius(200);

d3.select('g')
  .selectAll('path')
  .data(chords)
  .enter()
  .append('path')
  .attr('opacity', 0.5)
  .attr('stroke', 'blue')
  .attr('d', ribbonGenerator);
```

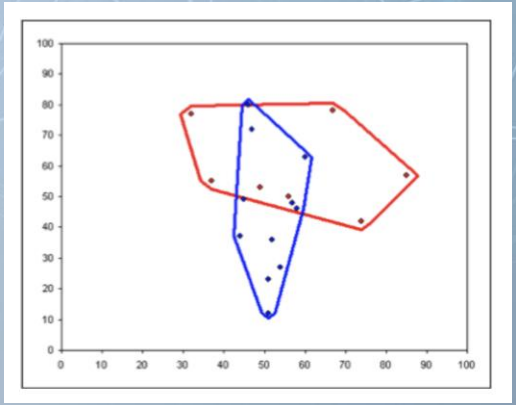
Bind the data to paths

‘d’ is the information to describe the path shape. ‘ribbonGenerator’ here indicates that the data bound to a path will pass to the ribbonGenerator to generator the path

Convex Hull

- In mathematics, the convex hull of a set of points in a Euclidean space is the smallest convex set that contain the points
  - Application: visualize different sets/clusters of points on the screen

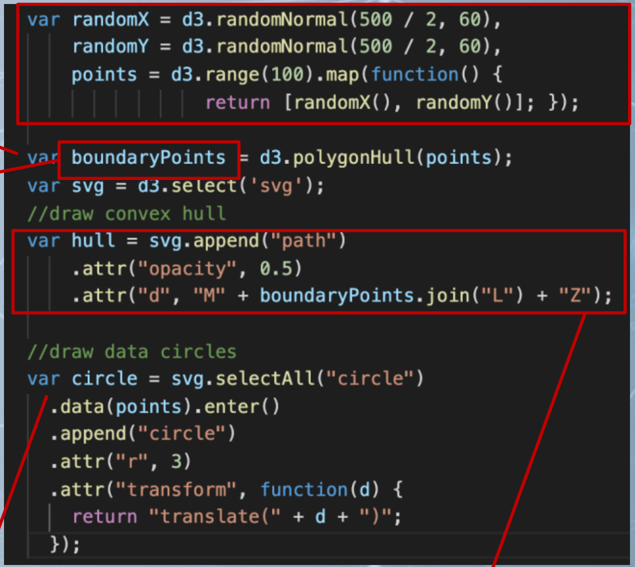




Convex Hull – d3.polygonHull()

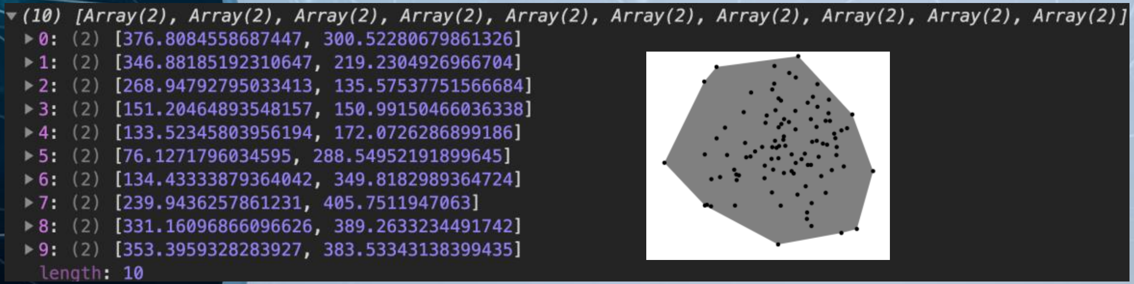
- d3.polygonHull()
  - Pass data point to it
  - it calculates boundary points for you

Generate 100 x-y pair and store them in ‘points’

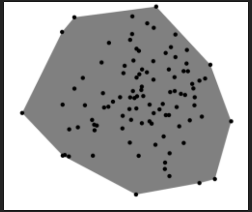


```
var randomX = d3.randomNormal(500 / 2, 60),
    randomY = d3.randomNormal(500 / 2, 60),
    points = d3.range(100).map(function() {
            return [randomX(), randomY()]; });

var boundaryPoints = d3.polygonHull(points);
var svg = d3.select('svg');
//draw convex hull
var hull = svg.append("path")
    .attr("opacity", 0.5)
    .attr("d", "M" + boundaryPoints.join("L") + "Z");

//draw data circles
var circle = svg.selectAll("circle")
  .data(points).enter()
  .append("circle")
  .attr("r", 3)
  .attr("transform", function(d) {
    return "translate(" + d + ")";
  });
```

They are the boundary points



```
▼(10) [Array(2), Array(2), Array(2), Array(2), Array(2), Array(2), Array(2), Array(2), Array(2), Array(2)]
 ▶0: (2) [376.8084558687447, 300.52280679861326]
 ▶1: (2) [346.88185192310647, 219.2304926966704]
 ▶2: (2) [268.94792795033413, 135.57537751566684]
 ▶3: (2) [151.20464893548157, 150.99150466036338]
 ▶4: (2) [133.52345803956194, 172.0726286899186]
 ▶5: (2) [76.1271796034595, 288.54952191899645]
 ▶6: (2) [134.43333879364042, 349.8182989364724]
 ▶7: (2) [239.9436257861231, 405.7511947063]
 ▶8: (2) [331.16096866096626, 389.2633234491742]
 ▶9: (2) [353.3959328283927, 383.53343138399435]
  length: 10
```



Draw all points by circles

Draw the polygon by the boundary points