Making a pie chart : Cleaning up

Pie charts also use generator functions to draw the wedges

But first, we need to clean up some things.

Get rid of axes and titles

Get rid of scaleY, turn scaleX into an ordinal color scale:

```
var scaleColor = d3 .scaleOrdinal ( )
        .domain ( [ "16-19“ ,  "20-24“ ,  "25-34“ , "35-44“ ,  "45-54“ , "55-64",
                “65+“ ] )
        .range ( [ "red“ , "orange“ , “yellow“ , "green“ , "blue“ , "purple“ ,
                "magenta“ ] );
```

Making a pie chart: Add generator functions

Now, create the arc generator function.

First, define a variable to set the radius of the pie chart:

```
var pieRadius = 200;
```

Next, use that variable in the arc generator function:

```
var makeArc = d3 .arc ( )
   .outerRadius ( pieRadius )
   .innerRadius ( 0 );
```

Then, make the pie generator function (notice that there are two generator functions working together to make a pie chart!):

```
var makePie = d3.pie ( )
    .sort ( null )
    .value ( function ( d ) {  return d.total;  } );
```

Making a pie chart: Data binding

Think about the data bind:

We only want one pie chart, but that pie chart should have several arcs. All of the data should be bound to the pie chart, and each arc should be bound to a single data object.

makePie will take care of making the arcs work together in a single pie chart, and makeArc will draw each arc individually.

First, make a variable for each arc to live in. Bind the selection to loadData, using makePie to create a group for each arc.

```
var g = svg.selectAll('.arc')
      .data(makePie(loadData))
      .enter()
      .append('g')
      .attr('class','arc');
```

Making a pie chart: Data binding

The variable we saved above has one group for each arc. In each group, we want to append a path, and give it a d attribute using the arc generator function makeArc.

```
 g .append ( 'path' )
    .attr ( 'd' , makeArc )
    .attr ( 'fill' ,  function ( d ) { return scaleColor ( d.data.age ) } )
});
```

Notice that I'm using d.data.age, not d.age to set the color! This is because makePie adds a new layer of hierarchy to my data, putting all of my original data objects inside a d.data object. To get back to the original data items, I need to look in d.data instead of just d.

(If this is confusing, put a console.log inside of the fill accessor function and before the return to see what the data looks like)

Positioning the pie chart

Make a group for the pie chart to live in, and give it a position based on the width and height of the screen:

```
var pieX = width/2;
var pieY = height/2;

var pieGroup = svg .append ( 'g‘ )
   .attr ( 'transform‘ ,  'translate ( ' + pieX + ‘  ,  '  +  pieY + ‘ ) ‘ );
```

Then, substitute this group for svg when drawing the pie:

```
g = pieGroup .selectAll (  '.arc‘  )
```

This doesn’t end up centered - adjust the pieX and pieY, taking margins into account.

Add labels

Define an arc to hold the labels, just like we did for the wedges

```
var labelArc = d3 .arc ( )
   .outerRadius ( pieRadius – 30 )
   .innerRadius ( pieRadius – 30 ) ;
```

Append labels in the draw function

```
g.append("text")
      .attr("transform", function ( d ) {
                        return "translate ( " + labelArc.centroid ( d ) + " ) ";
       })
      .attr( 'text-anchor' , 'middle' )
      .attr ( "dy" ,  ".35em" )
      .text ( function ( d ) { return d.data.age; } );
```

From pie to donut: one easy switch

If you prefer donut charts instead of pies, just adjust the innerRadius value

Using generator functions to make symbols

Go back to scatterplot example, add point data explaining how to draw symbols (could be added from a .csv, but just hard-coded it here to keep things simple)

Define a makeSymbol generator function, and a symbolData function to call it and pass it the right data:

```
var makeSymbol = d3 .radialArea ( )
   .angle ( function ( d ) {
      return d.angle;
   })
   .innerRadius ( function ( d ) {
      return d.r0;
   })
   .outerRadius ( function ( d ) {
      return d.r1;
   });

var symbolData  =  makeSymbol ( symbolPoints ) ;
```

Setting up the data bind

Before, each data object was used to draw a circle, so we combined the .data( ) and .append( ) steps in the same piece of code.

This time, we want to do something more like the pie chart. For each data object, create an empty group that we will append symbols into.

```
var symbol = svg .selectAll (‘.symbolGroups‘ )
      .data ( loadData )
      .enter ( )
      .append ( "g“ )
      .attr ( 'class‘ , 'symbolGroups‘ );
```

Check the DOM to make sure that they’re showing up.

Appending the symbols to groups

For each group stored in the symbol variable, we now need to add a path that will use makeSymbol to draw the symbolPoints.

Instead of calling makeSymbol directly, we call the symbolData function that we defined, which will call makeSymbol and give it the right list of points.

```
symbol .append ( 'path‘ )
      .attr ( 'd‘ , symbolData )
      .attr ( 'fill‘ , "gray");
```

Move the symbol groups to the right places

When we append circles, they position themselves using the cx, cy coordinates that we give them. Groups need to be positioned using translate:

```
var symbol = svg .selectAll ( '.symbolGroups' )
      .data ( loadData )
      .enter ( )
      .append ( "g" )
      .attr ( 'class' , 'symbolGroups' )
      .attr( 'transform' ,  function ( d ) {
          return 'translate ( ' +  scaleX(d.age) +   ' , ' +  scaleY(d.women)  +  ' )'
      });
```

And make them a nicer size:

```
 symbol .append ( 'path' )
      .attr ( 'd' , symbolData )
      .attr ( 'fill' , "gray")
      .attr ( 'transform' , 'scale ( .1 ) ' );
```

Dorling Cartogram: Add scaled points to a map

Now, let's add a point for each state in the map, and scale it according to the population size (instead of using color, like we did in the chloropleth map).

Copy and paste the queue( ) code from the chloropleth_complete script (also update the map .data function and HTML code):

```
queue()
   .defer(d3.json, "./cb_2016_us_state_20m.json")
   .defer(d3.csv, "./statePop.csv")
   .await(function(err, mapData, populationData){
```

Also copy in stateLookup and color scale (change to size scale)

```
var stateLookup = d3.map( );

var sizeScale = d3.scaleLinear( ).range ( [ 0, 100 ] );
```

Dorling Cartogram: Setting up the lookup function

Copy in the stateLookup.set function, and the colorScale.domain:

```
populationData.forEach(function(d){
      stateLookup.set(d.name, d.population);
});

colorScale.domain([0, d3.max(populationData.map(function(d){return
+d.population}))]);
```

We want to plot a point for each state, placed in the center of the state. To do this, we need a list of the lat/long for each state center. We get this using d3's centroid function:

```
 var centroids = features.map(function (feature){
   return path.centroid(feature);
  });
```

Dorling Cartogram: Drawing circles

Update the circle drawing function:

```
.data(centroids)

.attr('cx', function (d){
        return d[0];
})
      .attr('cy', function (d){
        return d[1];
})
```

Dorling Cartogram: Link to state name

Next, we need to connect the circle size to the population data. Look at centroids function: doesn't contain state name! Edit the return part of the centroids function to return an object:

```
return {
        name: feature.properties.NAME,
        center: path.centroid(feature)
};
```

Update the drawing code to match:

```
d.center[0]
d.center[1]
```

Also, add an id with the state name to help identify the NaN point:

```
.attr ( 'id' , function ( d ) {  return d.name  } )
```

Dorling Cartogram: Cleaning up the data

Dealing with NaN for Puerto Rico:

Option 1: shove the circle off the map.

```
if ( isNaN ( d.center [ 0 ] ))  {
        return -100;
}
else {
        return d.center[0];
}
```

Beware: it's easy to miss errors in your code this way, and if you have too many circles drawn, it can slow your browser down significantly.

You could use a similar strategy in the 'r' attribute, setting the size to zero if the value is NaN, to hide the circles (same problems)

Dorling Cartogram: Cleaning up the data

A better way to deal with it is to cut that data out of the original centroid array before you bind it.

```
noPR = centroids.filter ( function ( d ) {
                                        return ! isNaN ( d.center [ 0 ] );
                                });
```

And then bind to that array instead.

Probably the best way is to go into the geoJSON and remove that data from the JSON file.

Go into the geoJSON, and use the Find command to locate the Puerto Rico feature. Select the whole feature (and only that feature), and delete.

Only do this if you are very sure, and make a backup of your data file first!

Dorling Cartogram: Apply size scaling

Now, let's actually connect the circle size to the data:

```
.attr ( 'r' ,  function ( d ) {
        return    sizeScale (  stateLookup.get ( d.name ) )
} )
```

Adjust size scaling and circle opacity to make the map more readable. Consider adding tooltips to show state name and properties (keep in mind that circles under others won't be accessible)

Symbol map: put symbols on the map

Maybe you just want to show geographic distributions of data points, using a symbol.

In the file symbolMap, I've replaced the population data with a list of latitude and longitude points and names.

Start by drawing a circle for each one:

```
svg.selectAll('circle')
      .data(dataPoints)
      .enter()
      .append('circle')
      .attr('cx', function (d){
         return albersProjection ( [ d.long, d.lat ] ) [ 0 ];
      })
      .attr('cy', function (d){
         return albersProjection ( [ d.long, d.lat ] ) [ 1 ];
      })
      .attr('r', 3)
```

Symbol map: replace circles with symbols

Now, draw a symbol for each point, rather than a circle.

Copy in symbolPoints, makeSymbol, and symbolData from the symbols example

Instead of circles, append groups, and give them a translate function based on the geographic projection. Save the groups in a variable

```
symbol = svg .selectAll ( '.symbolGroups‘ )
      .data ( dataPoints )
      .enter( )
      .append( ‘ g‘ )
      .attr('class‘ , 'symbolGroups‘ )
      .attr('transform', function( d ) {
            return 'translate('+
                  albersProjection ( [ d.long, d.lat ] ) [ 0 ] +','+
                        albersProjection ( [ d.long, d.lat ] ) [ 1 ] +')‘
            });
```

Symbol map: replace circles with symbols

Draw the symbols inside the groups we just made.

```
symbol .append ( 'path‘ )
      .attr ( 'd‘ , symbolData )
      .attr ( 'fill‘ , "gray“ )
      .attr ( 'transform‘ , 'scale(.1)‘ );
```

Building a timeline

First, set up the HTML structure. One div with a width of 100%.

SVG inside of the div is 300%.

Style for the parent div includes:

```
{
  overflow-y: scroll;
}
```

Formatting time

D3 uses a standard time object to deal with dates. To create this object, you need to parse your date data into the correct format.

The setup is similar to generator functions; you build a parse function at the top of the document, load your data, and then run the parse function on all of the elements of the data to create a date object for each one.

First, define the parse function, which tells d3 how your date is currently stored:

```
var parser = d3.timeParse ( "%m/%d/%Y“ );
```

(Note that there are lots of options for input formats: https://github.com/d3/d3-time-format#locale_format)

Parsing dates

Next, load the data. Then, run through every element in the dataset and parse its date information. Save the result to a new attribute:

```
dataIn .forEach ( function ( d ) {
      d.date  =  parser ( d.start_date );
});
```

Set up a scale to use these dates:

```
var scaleX  =  d3 .scaleTime ( ) .range ( [ 0 ,  width-2*marginLeft ] );
```

After the data is parsed, use the new date objects to set the domain:

```
scaleX .domain (
            [ d3 .min ( dataIn .map ( function ( d ) { return d.date } ) ),
              d3 .max ( dataIn .map ( function ( d ) {return d.date } ) ) ] )
```

Adding a time axis

Add an axis to the graph, and attach it to the time scale:

```
svg .append ( "g“ )
      .attr ( 'class‘ , 'x-axis‘ )
      .attr ( 'transform‘ , ‘translate ( 0 , '  +  ( height – 2 * marginTop ) + ‘ ) ‘ )
      .call ( d3 .axisBottom ( scaleX ) );
```

Remember to call the axis again after the scale has updated! Also, tell it to display a tick for every other year in the dataset:

```
d3 .select ( ‘.x-axis‘ )
      .call ( d3 .axisBottom ( scaleX ) .ticks ( d3 .timeYear .every ( 2 ) ) );
```

Drawing lines on the timeline

Set up a y axis scale, and use it to draw a line for each data point:

```
var scaleY = d3 .scaleLinear ( )
                    .range ( [ 0 , height – 2* marginTop ] )
                    .domain ( [ 0 , 200 ] );

 svg .selectAll (‘.date-lines‘ )
      .data ( dataIn )
      .enter ( )
      .append ( 'line‘ )
      .attr (‘class’ ,  ‘date-lines’)
      .attr ( 'x1‘ , function ( d ) {  return scaleX ( d.date );   })
      .attr ( 'x2‘ , function ( d ) {  return scaleX ( d.date );   })
      .attr ( 'y1‘ , scaleY(0) )
      .attr ( 'y2', scaleY(200) )
      .attr ( 'stroke-width‘ , 1 )
      .attr ( 'stroke‘ , 'gray‘ );
```

Add text to each line

Split the .enter( ) from the line drawing

```
var lines = svg .selectAll ('.date-lines' )
      .data ( dataIn )
      .enter ( );

lines
      .append('line')…
```

And then add text items:

```
lines .append ( 'text' )
        .attr ( 'x' ,  function ( d ) {
             return scaleX ( d.date );
         })
        .attr ( 'y' , scaleY ( 200 ) )
       .text ( function ( d ) { return d.text } );
```

Alternative to scrolling div approach:

http://talkingdirt.info/page1.php#K

Based off of:
https://bl.ocks.org/mbostock/34f08d5e11952a80609169b7917d4172

Next week:

Brief proposal presentation of each of your projects (5 mins + 5 mins discussion)

Slide 1: Overview of your data, and the question that you want to answer (what do you want your viewers to know after seeing your project?)

Slide 2 (through 4, if needed): Present sketches or early versions of visualizations, and discuss how they support (or don't support) users in gaining these insights.

Slide 3: Summarize your current status, and where you plan to go from here. Also, think about how you plan to present your visualizations on a webpage, and discuss what other elements you might need.

First half of class is presentations, second half is individual/group work and individual consults. Come prepared to work on your final project code in class.