## Week 9

# Drawing Complex Shapes

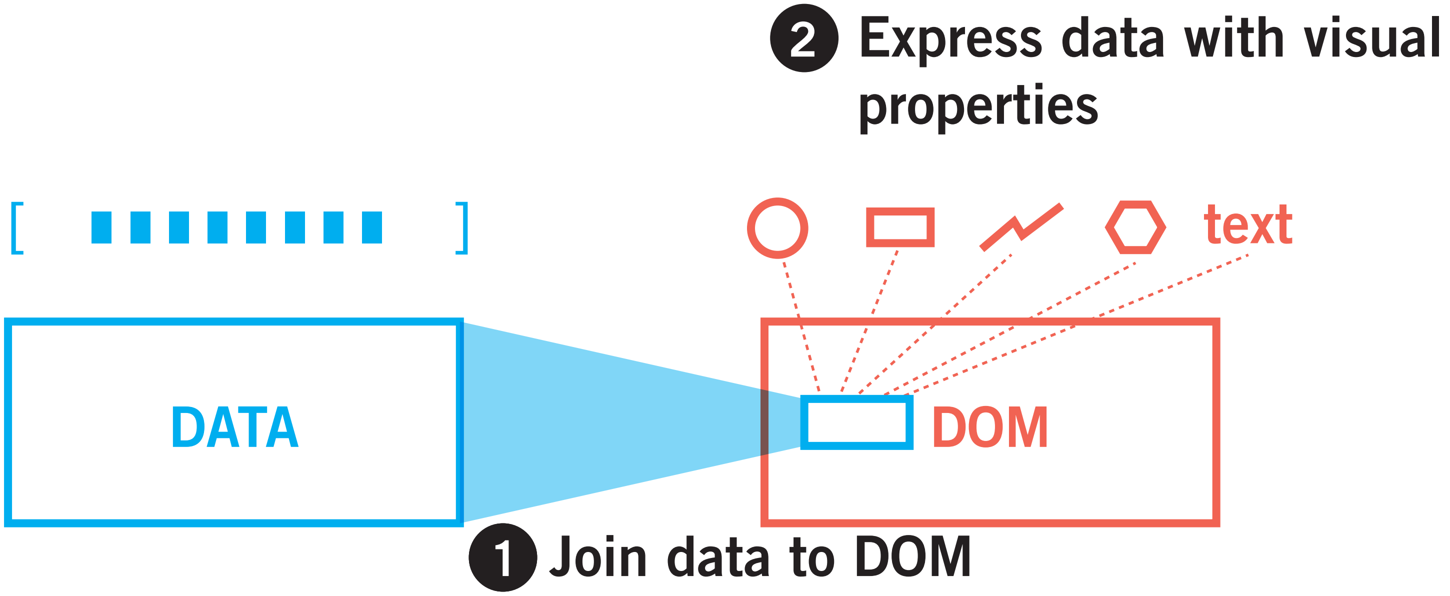
## Layout & Generator Functions

# Generalizing the Data Viz Process

Acquire
Parse
Filter
Mine
Represent
Refine
Interact

# "REPRESENT" IN d3 - DESIGN INTENTION

The basic idea is to "join" a piece of data to a DOM element, and then use the visual attribute of the DOM element to express the data



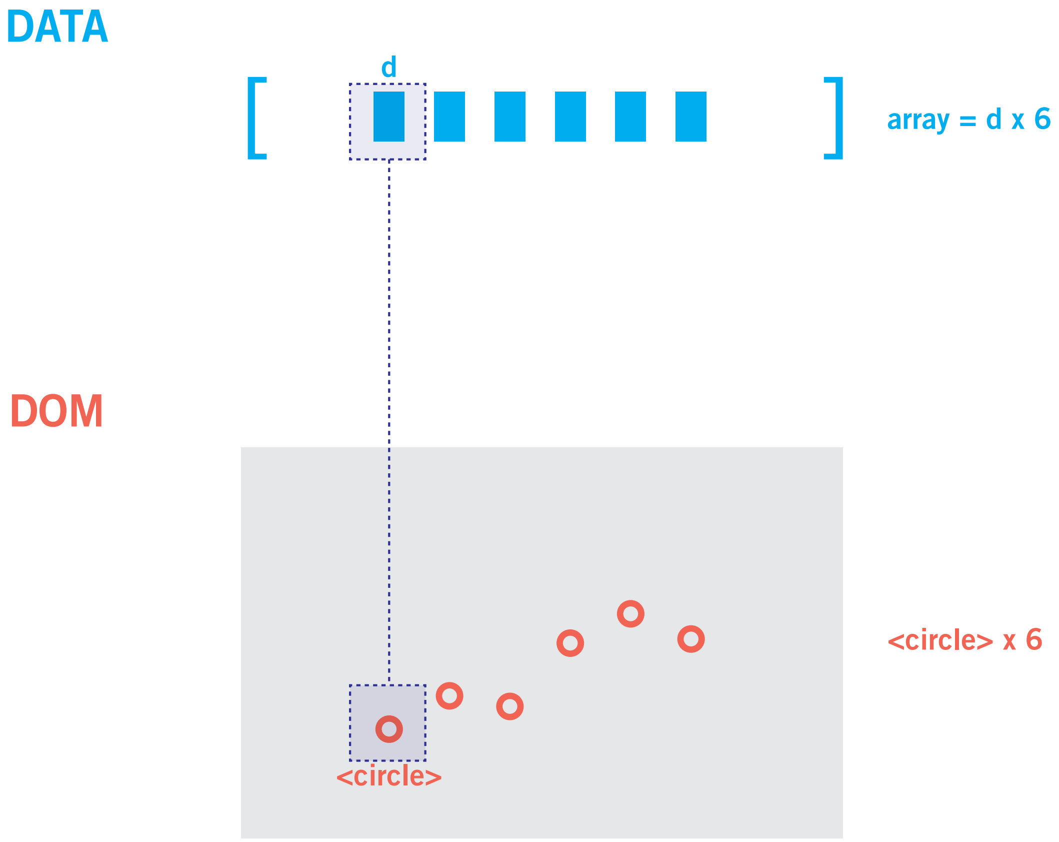
# DIFFERENT IMPLEMENTATION, SAME INTENTION

In Weeks 6 and 7, we used the enter/exit/update pattern to compute a "**many-to-many**" join between data and `<circle>` elements in a scatterplot.

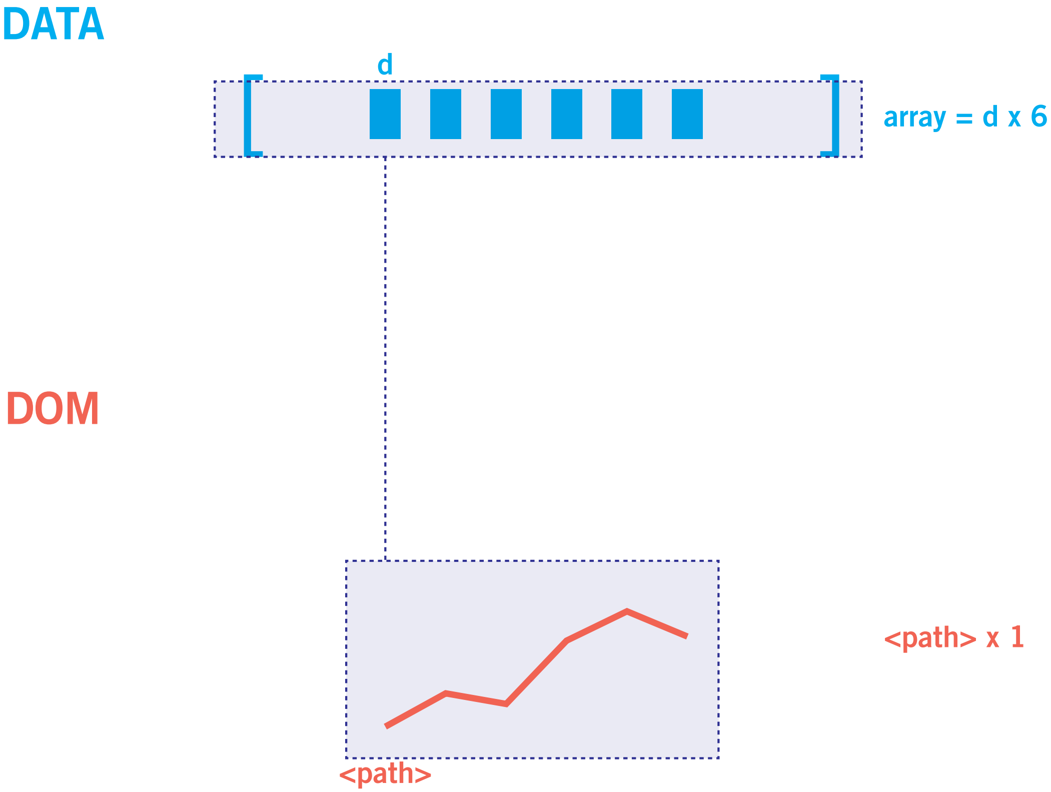
Last week, we performed a "**one-to-one**" join between a data array and a `<path>` element, and then used a generator function to generate a geometry for the path.

In both cases, the fundamental design intention of joining data to DOM is the same.

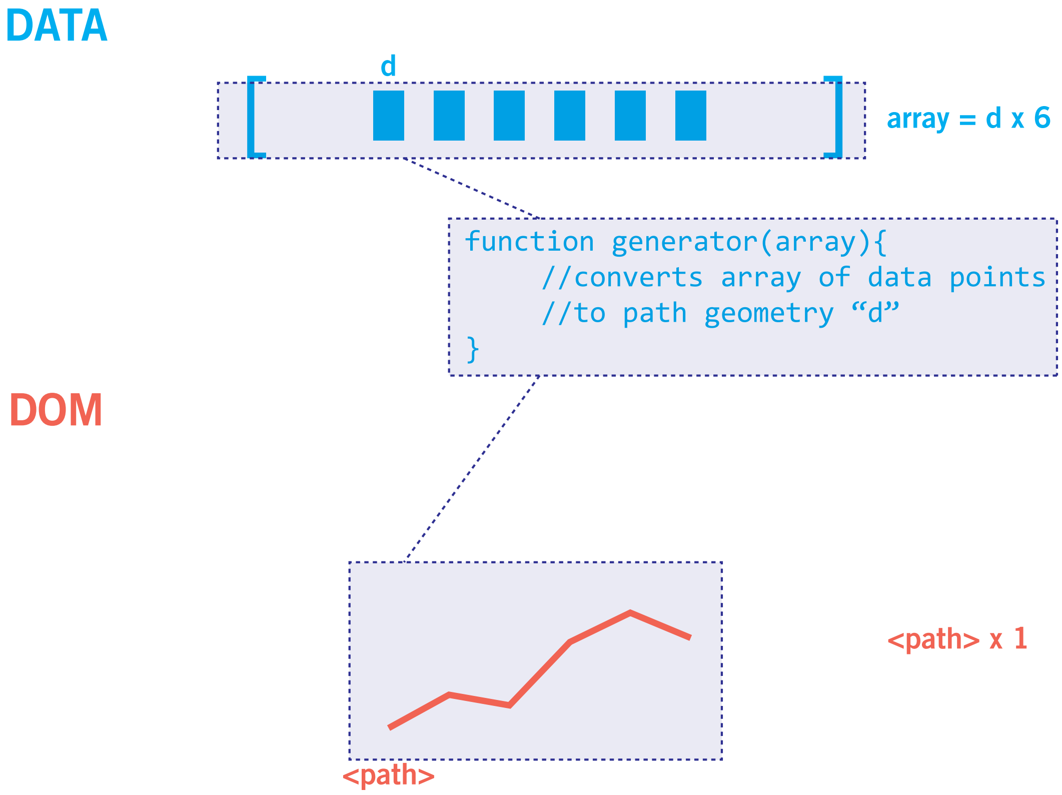
# “Many to Many” Relationship in a Scatterplot



# “One-to-One” Relationship in a Line Graph



# d3 Generators Functions



```
function generator(array){
     //converts array of data points
     //to path geometry “d”
}
```

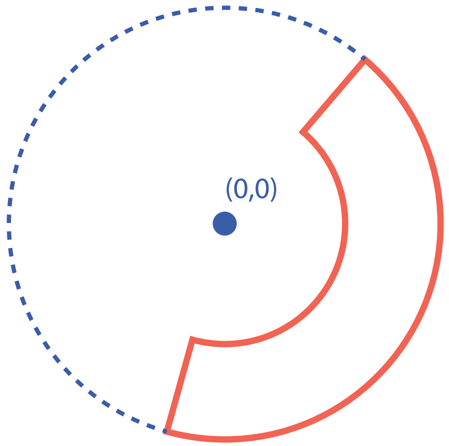
We rely on d3 generator functions to generate the geometry attribute “d” of `<path>` elements from the data joined to them.

# List of Built-in d3 Generators

```
d3.svg.line()
d3.svg.area()
d3.svg.arc()
d3.svg.chord()
d3.svg.diagonal()
```

# Drawing an Arc

An arc is an SVG `<path>` element, and forms part of a circle.

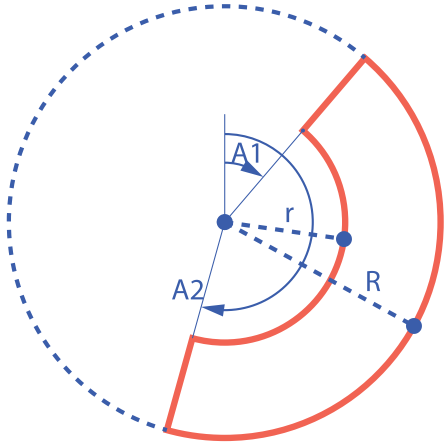


# Drawing an Arc

An arc is an SVG `<path>` element, and forms part of a circle.

To completely describe and generate the geometry for the arc, we need to know
1. **A1** start angle (from 12 o'clock)
2. **A2** end angle
3. **r** inner radius
4. **R** outer radius

*360 degree = 2*Math.PI radians



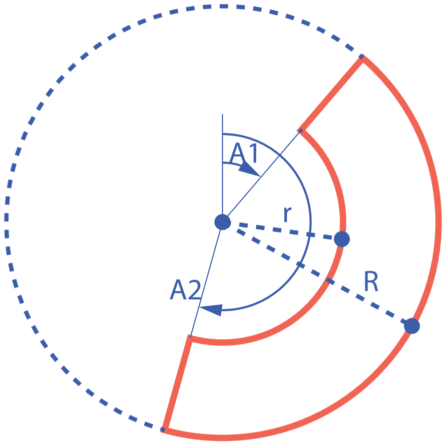
# Drawing an Arc

An arc is an SVG `<path>` element, and forms part of a circle.

To completely describe and generate the geometry for the arc, we need to know

1. **A1** start angle (from 12 o'clock)
2. **A2** end angle
3. **r** inner radius
4. **R** outer radius

*360 degree = 2*Math.PI radians



# d3.svg.arc()

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

- function(d){...} are accessor functions i.e. given a data object, how to extract the attribute defiining startAngle, endAngle etc.

This returns a function.

# d3.svg.arc()

```
var arc = d3.svg.arc()
    .startAngle(function(d){ return d.a1;})
    .endAngle(function(d){return d.a2})
    .innerRadius(function(d){return d.r1})
    .outerRadius(function(d){return d.r2});
```

```
{
    a1:0,
    a2:Math.PI,
    r1:100,
    r2:300
}
```

# d3.svg.arc()

```
var arc = d3.svg.arc()
    .startAngle(function(d){ return d.a1;})
    .endAngle(function(d){return d.a2})
    .innerRadius(0)
    .outerRadius(function(d){return d.r});
```

Any one of the properties can be defined as a constan, independent of data.

```
{
        startAngle:0,
        endAngle:Math.PI,
        r:500
}
```

# d3.svg.arc()

```
var arc = d3.svg.arc()
    .startAngle(function(d){ return d.a1;})
    .endAngle(function(d){return d.a2})
    .innerRadius(0)
    .outerRadius(function(d){return d.r});
...

plot.append(‘path’)
    .datum(dataObject)
    .attr(‘d’,arc);
```

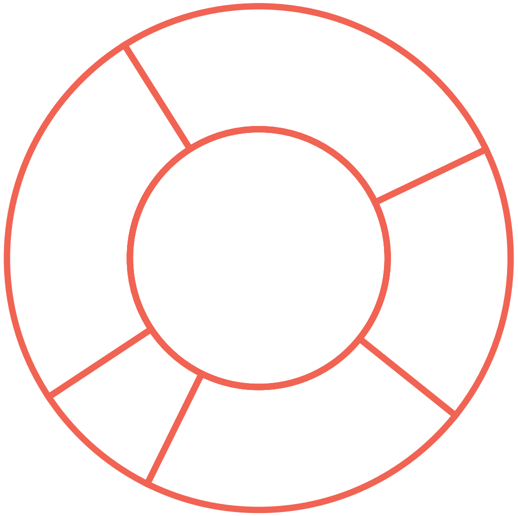
```
{
        startAngle:0,
        endAngle:Math.PI,
        r:500
}
```

# Exercise 1

Drawing arc-shaped <`path`> elements with the `d3.svg.arc()` generator

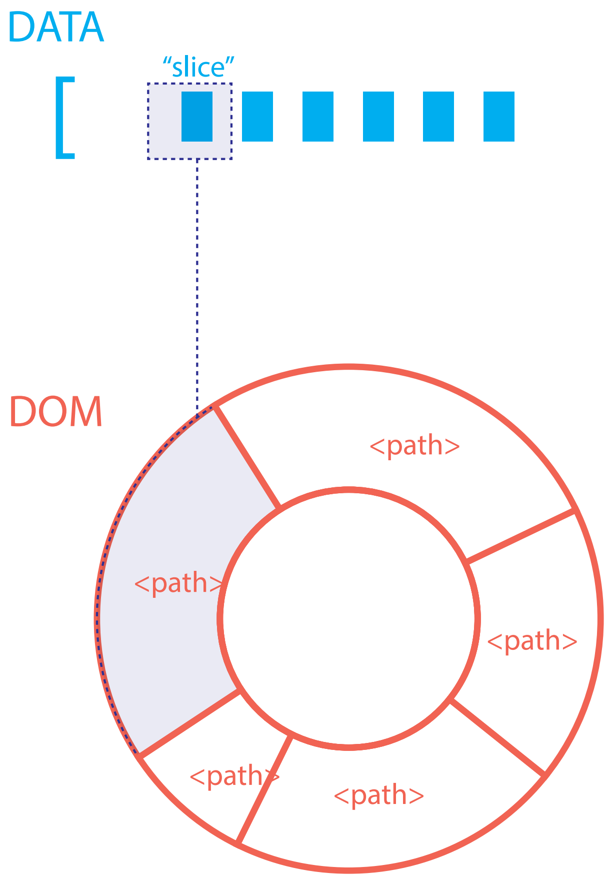
# Pie Chart

Visualizes the proportion of "slices" that collectively make up 100%

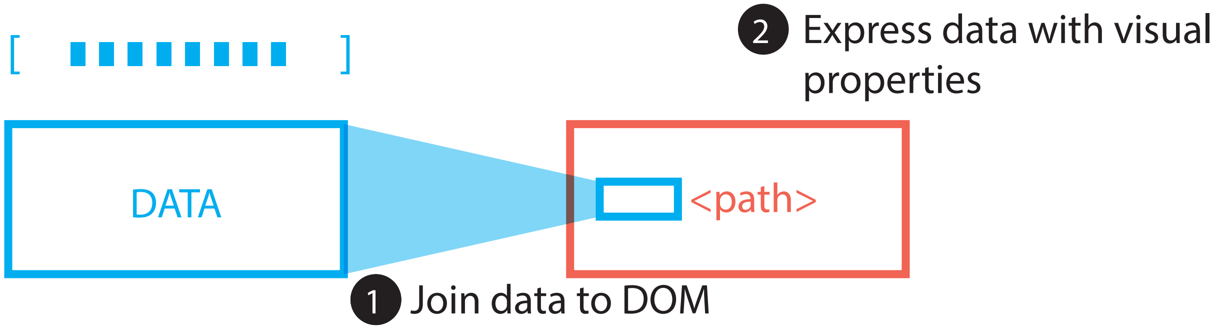


# Pie Chart

In terms of d3 implemention, pie charts consist of <`path`> elements drawn as arcs, and joined to an array of data ("many to many").



One complication: how can our dataset embed information about start angle, end angle etc. relative to each other?



# Layout Functions

Conceptually, before being joined to DOM elements, our array of data needs to be transformed, so that they contain attributes for startAngle, endAngle etc.

That’s the job of layout functions.

# `d3.layout.pie()` **Layout**

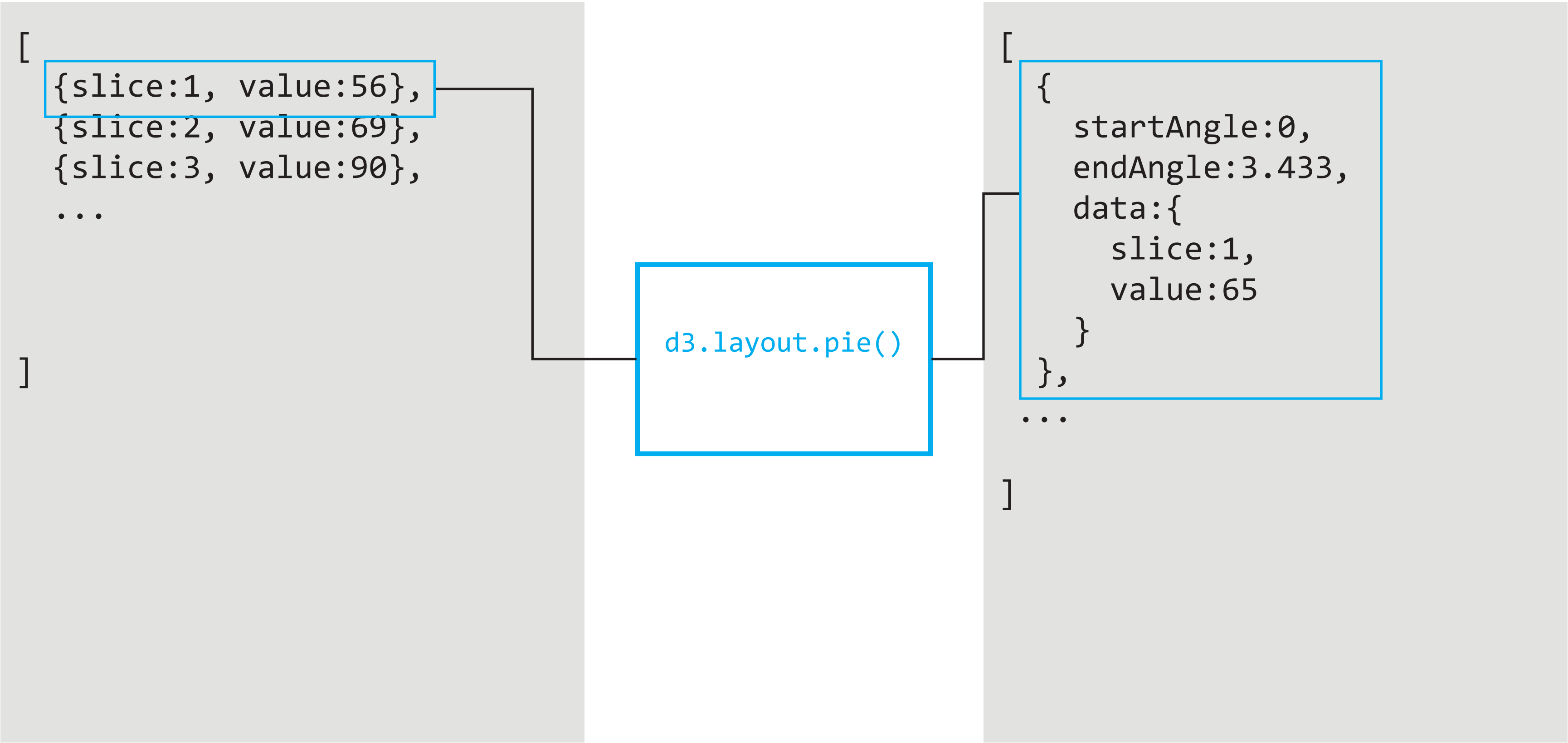
```
var pie = d3.layout.pie()
    .value( function(d){ ... });
```

This returns a function.

Given an array, the function transforms it so that each array element will have attributes

1. `startAngle`
2. `endAngle`
3. `data` --> which encapsulates the original array element, pre-transformation

# `d3.layout.pie()` **Layout**

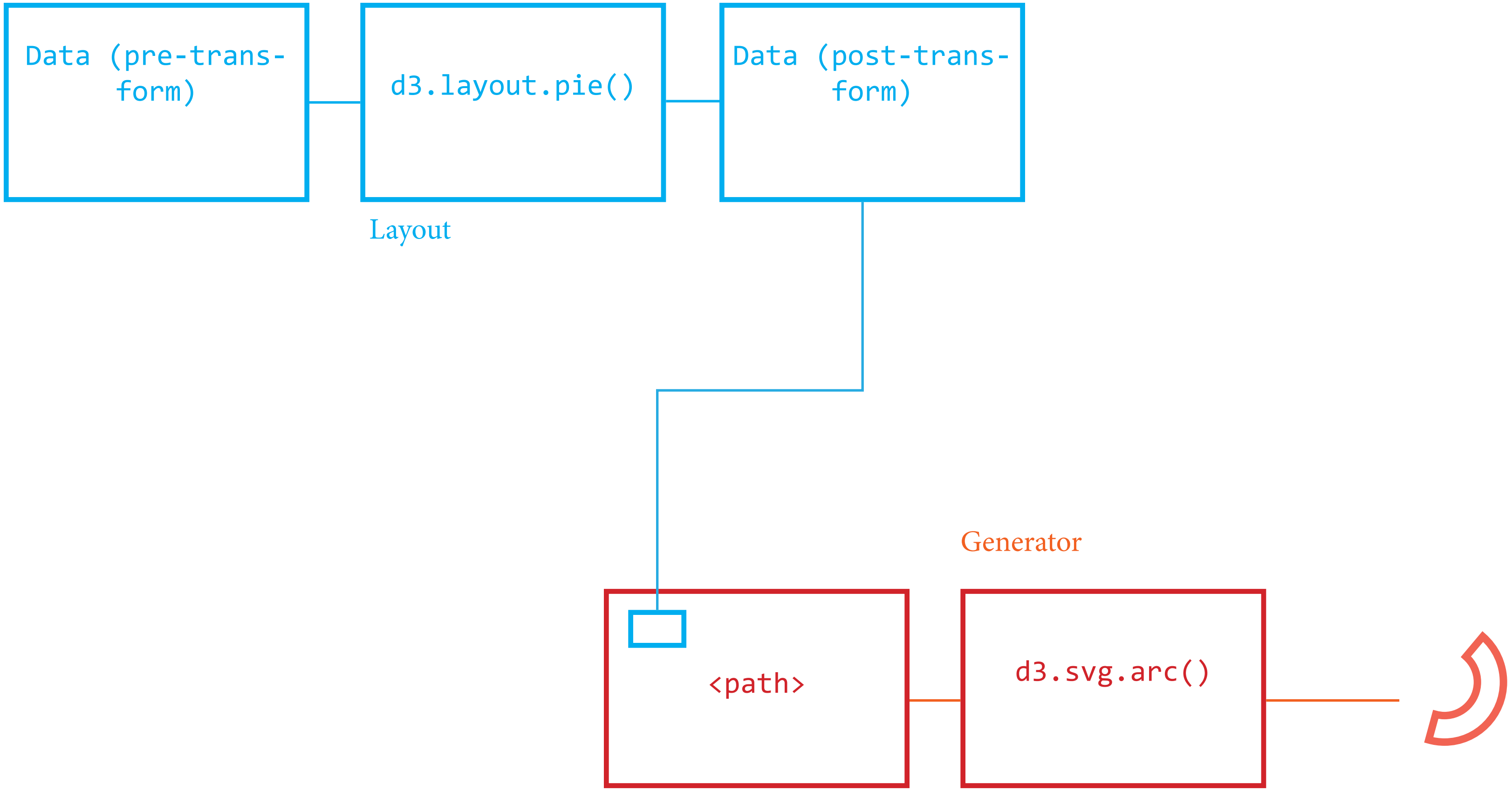


```
[
  {slice:1, value:56},
  {slice:2, value:69},
  {slice:3, value:90},
  ...


]
```

```
d3.layout.pie()
```

```
[
 {
   startAngle:0,
   endAngle:3.433,
   data:{
     slice:1,
     value:65
   }
 },
...

]
```

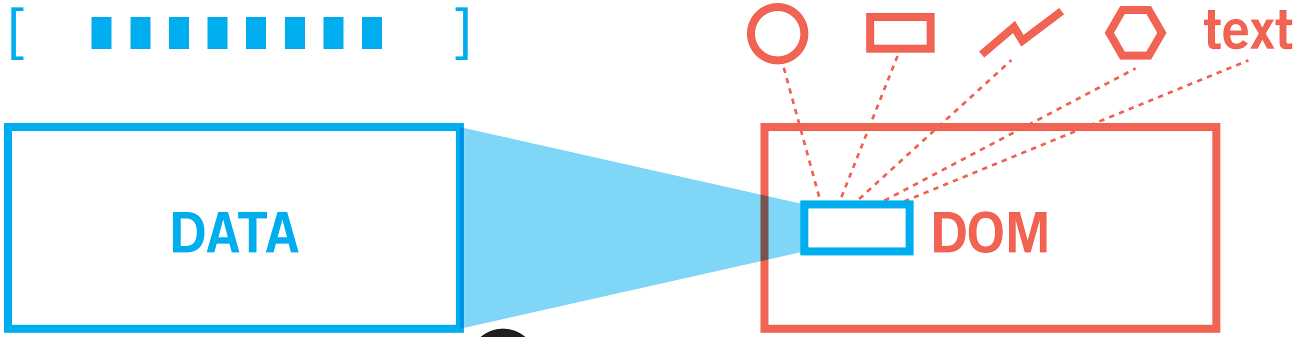
# Layout to Generator



# Exercise 2

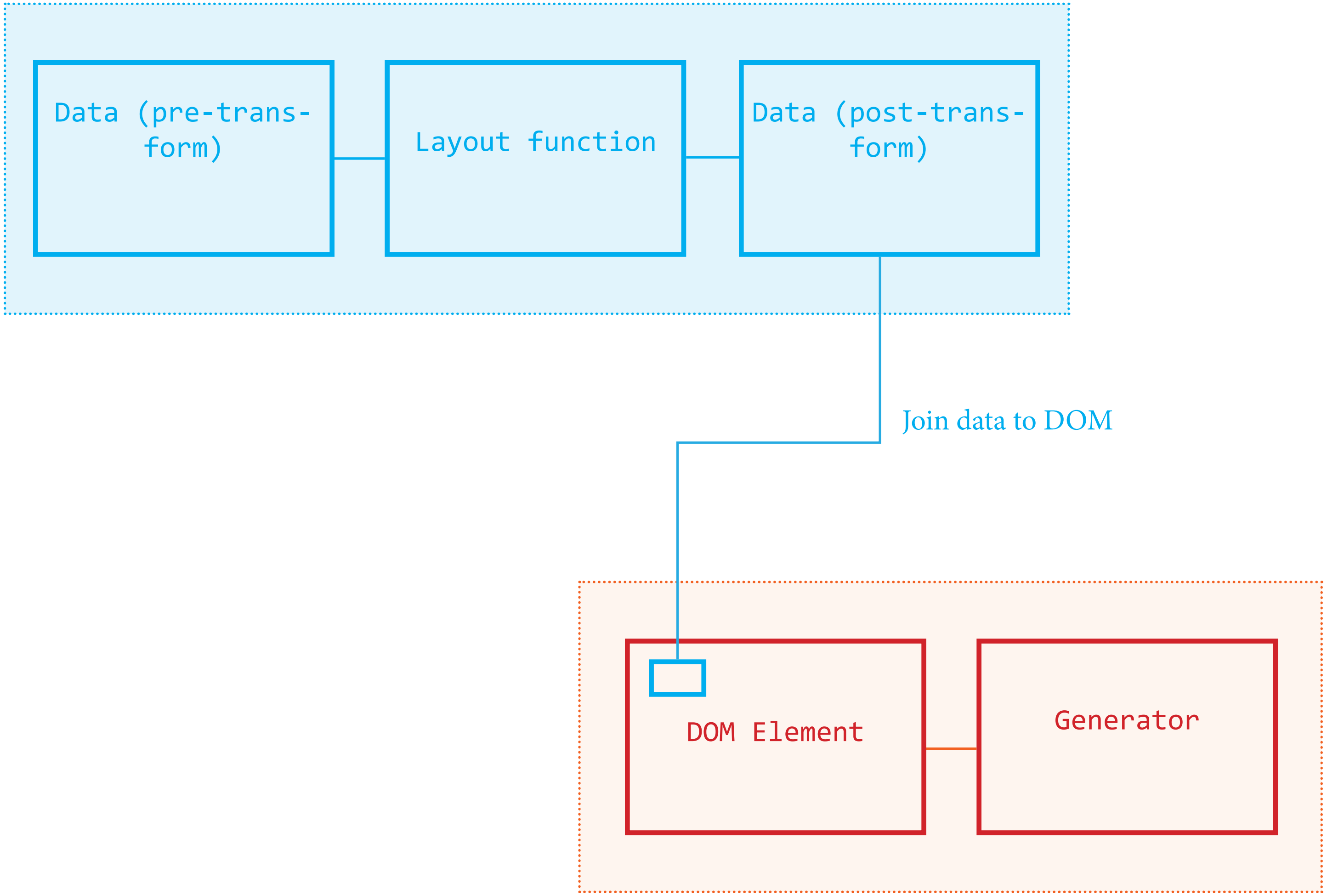
Let’s draw a pie chart!

2 Express data with visual properties



1 Join data to DOM

Data:
Transform data to have the right structure



Data (pre-transform)

Layout function

Data (post-transform)

Join data to DOM

DOM Element

Generator

DOM:use the visual attributes of DOM elements to express attributes of the data, including their shape (generator function)

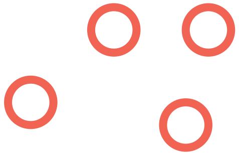
Both layout and generator functions anticipate a certain defined data structure.

More crucially, data sets express certain fundamental relationships, and anticipate certain fundamental visualization types:

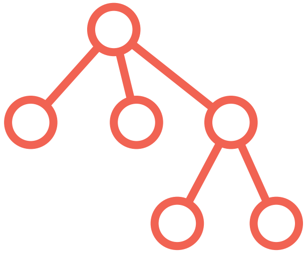
1. Examine the data;
2. What kind of fundamental relationship does it express? What aspect of that relationship should we highlight?
3. What steps for data transformation and DOM manipulation should I take?

# Common Types of Data

Point

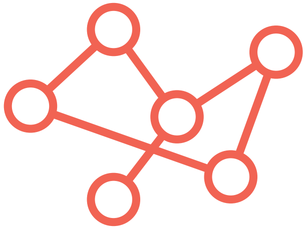


Hierarchy

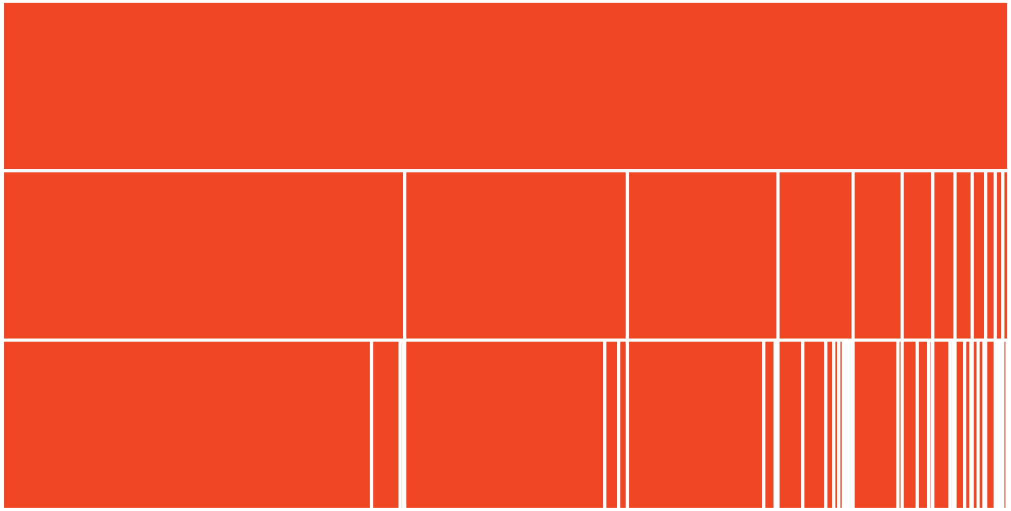


Line/Serial

Graph



# Partition Layout: A Type of Hierarchy Layout



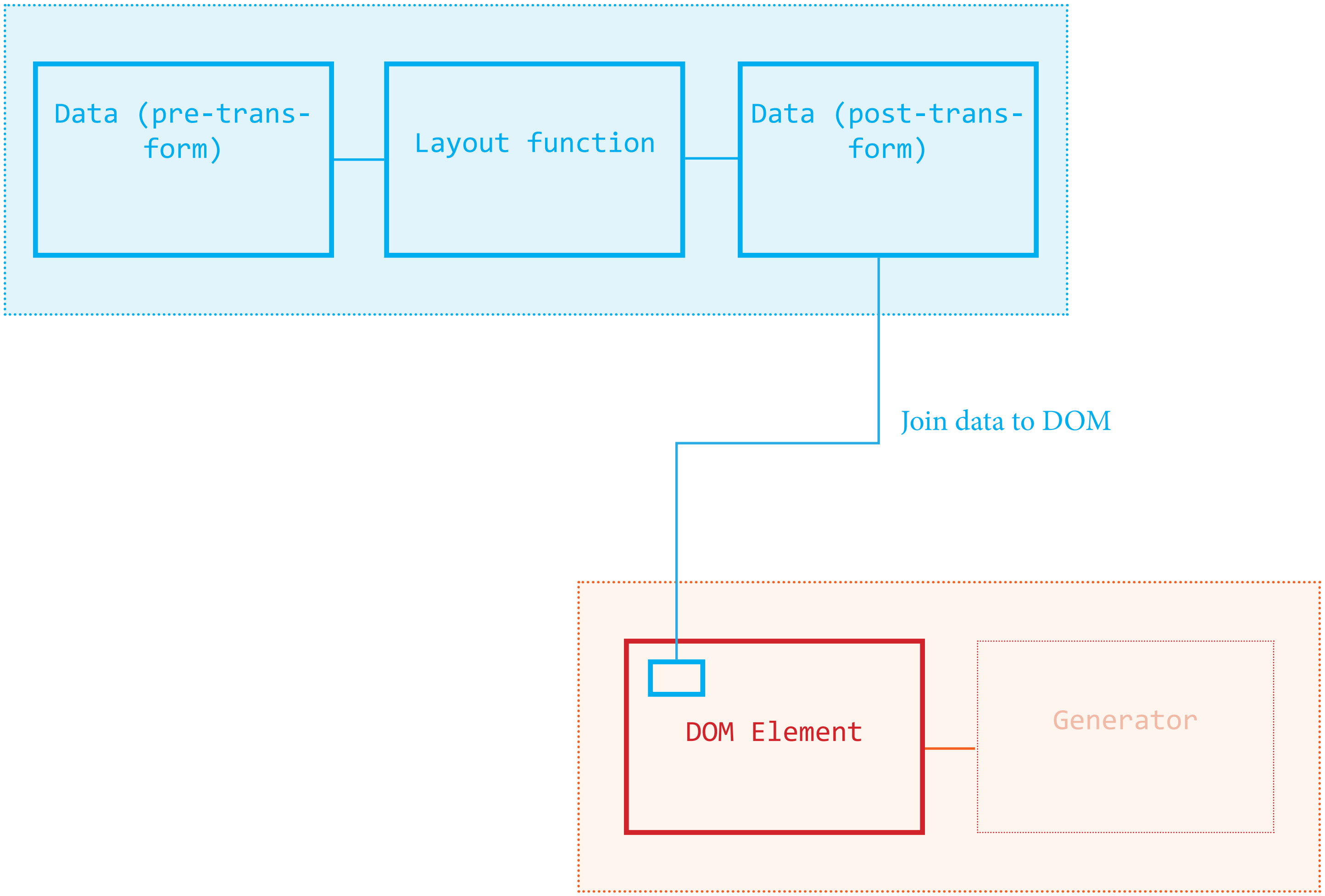
# Partition Layout: A Type of Hierarchy Layout

```
var partition = d3.layout.partition()
    .size([width,height])
    .children( function(d){...})
    .values( function(d){...} )
```

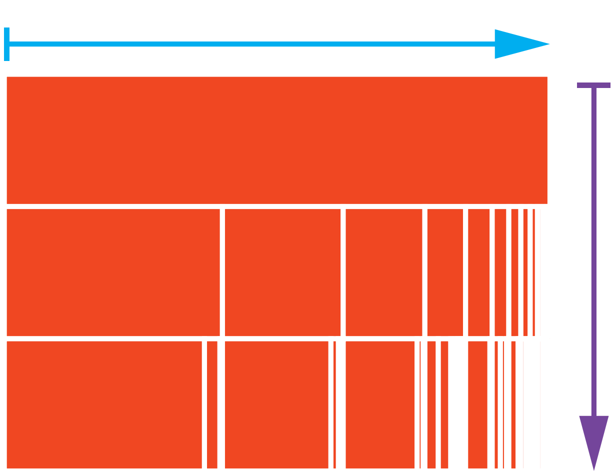
# Exercise 3

Let’s Draw a Partition Diagram

Data:
Transform data to have the right structure

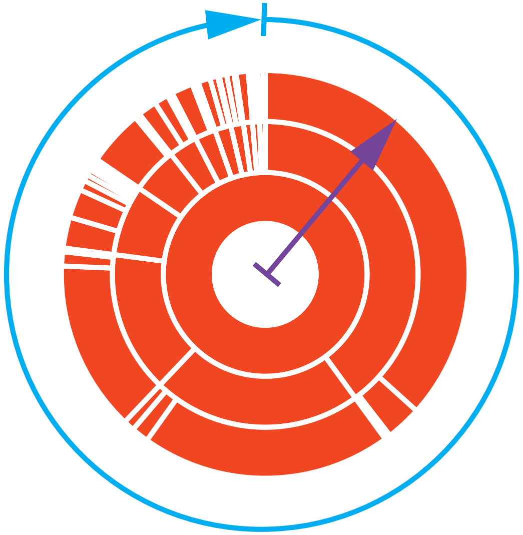


DOM:use the visual attributes of DOM elements to express attributes of the data, including their shape (generator function)



x: 0 -> width

y: 0 -> height



angle : 0 -> 2*Math.PI

radius