D3

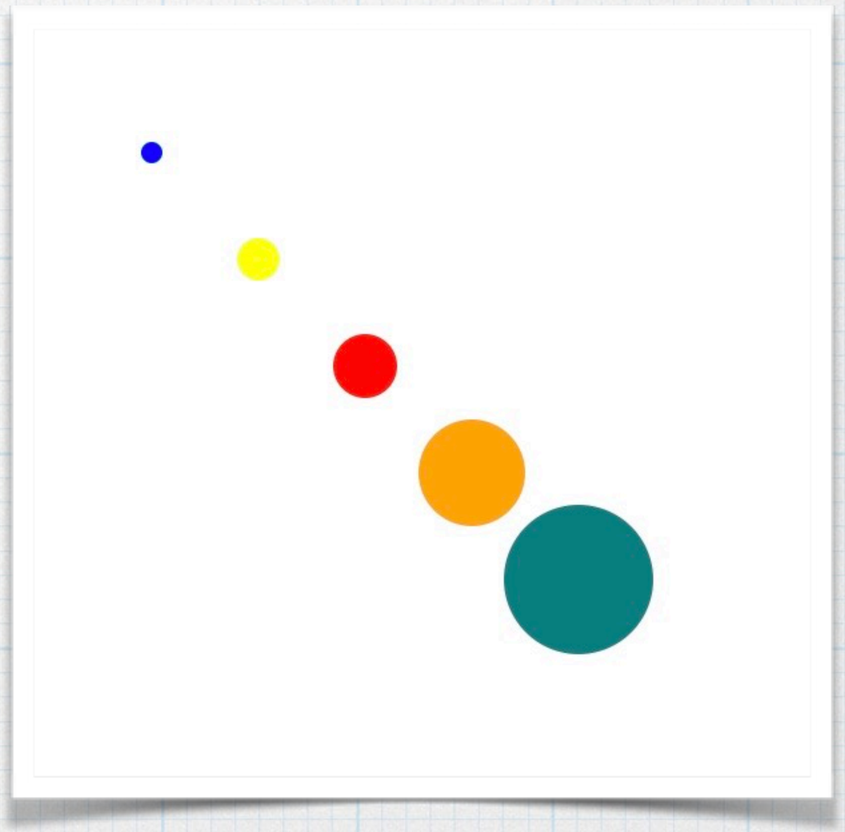
Data-Driven Documents Introduction

USING DATA TO CREATE SVG

The whole idea of D3 is to import some data and use that data to create SVG elements: circles, rectangles, etc. We will structure those SVG elements into useful data visualizations. For now, let's start by drawing some circles based on data.

```
var my_data = [1, 2, 3, 5, 7];
```

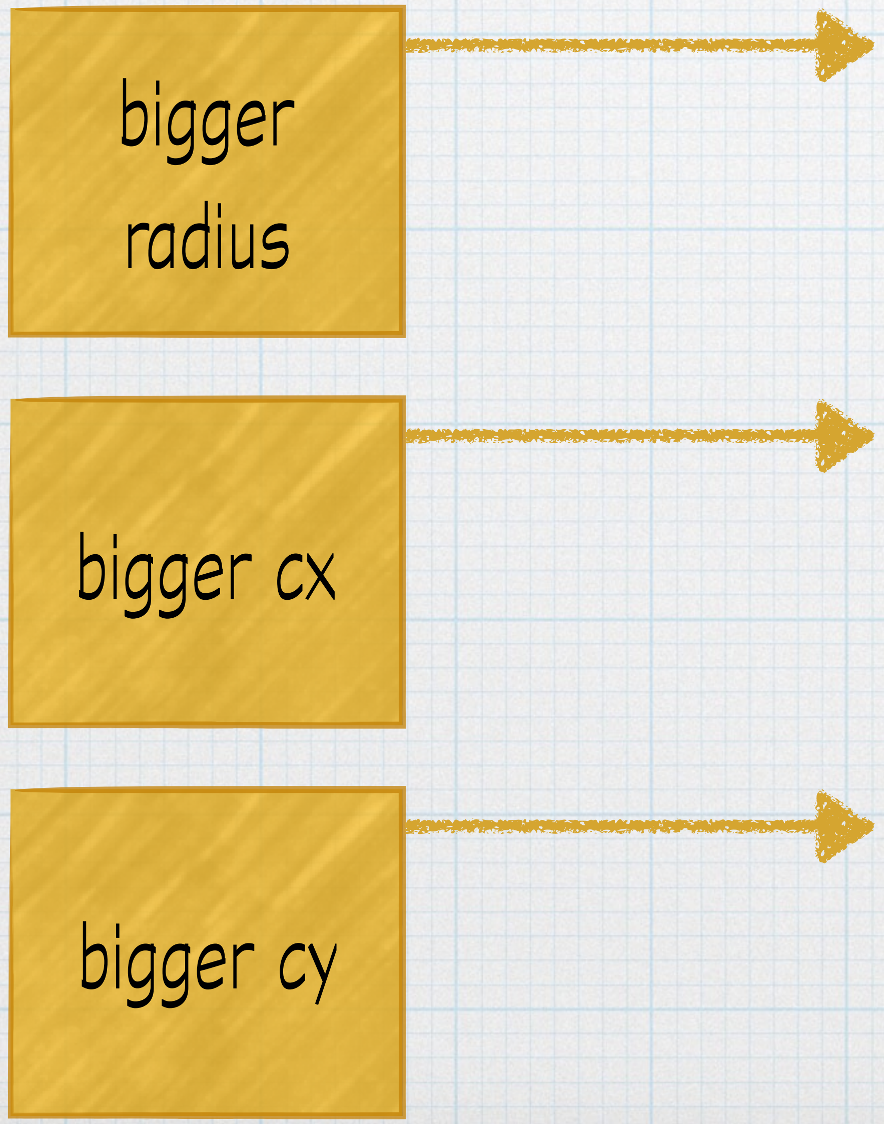
I want circles that represent the size of my data points.

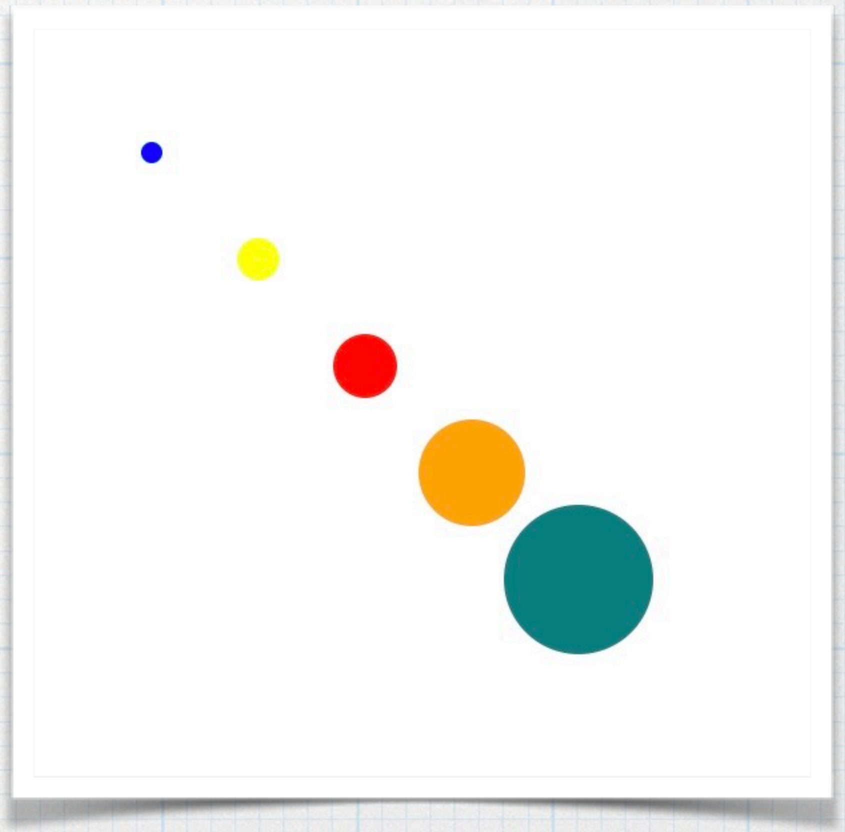


USING DATA TO CREATE SVG

The whole idea of D3 is to import some data and use that data to create SVG elements: circles, rectangles, etc. We will structure those SVG elements into useful data visualizations. For now, let's start by drawing some circles based on data.

```
var my_data = [1, 2, 3, 5, 7];
```

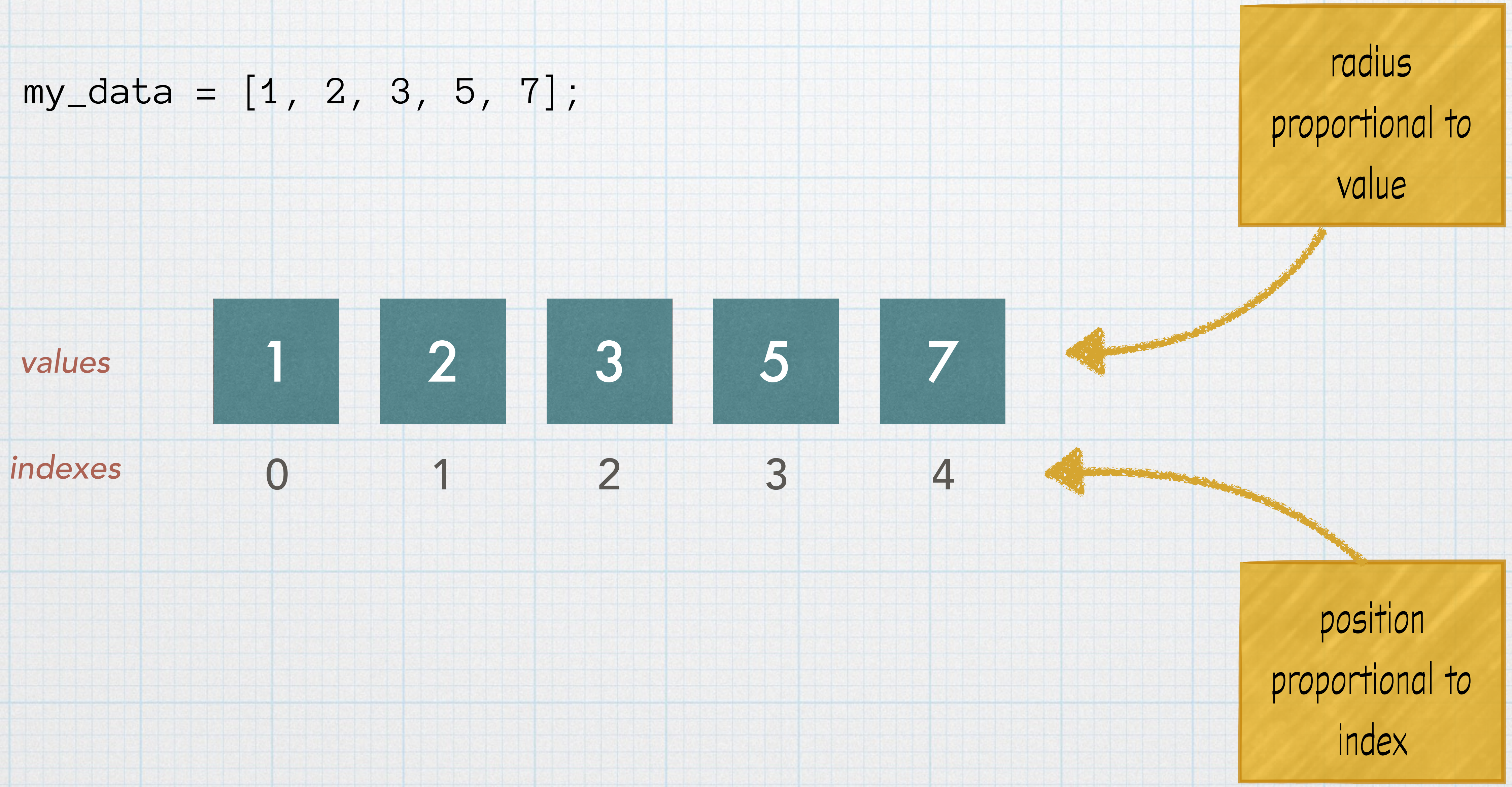




USING DATA TO CREATE SVG

The whole idea of D3 is to import some data and use that data to create SVG elements: circles, rectangles, etc. We will structure those SVG elements into useful data visualizations. For now, let's start by drawing some circles based on data.

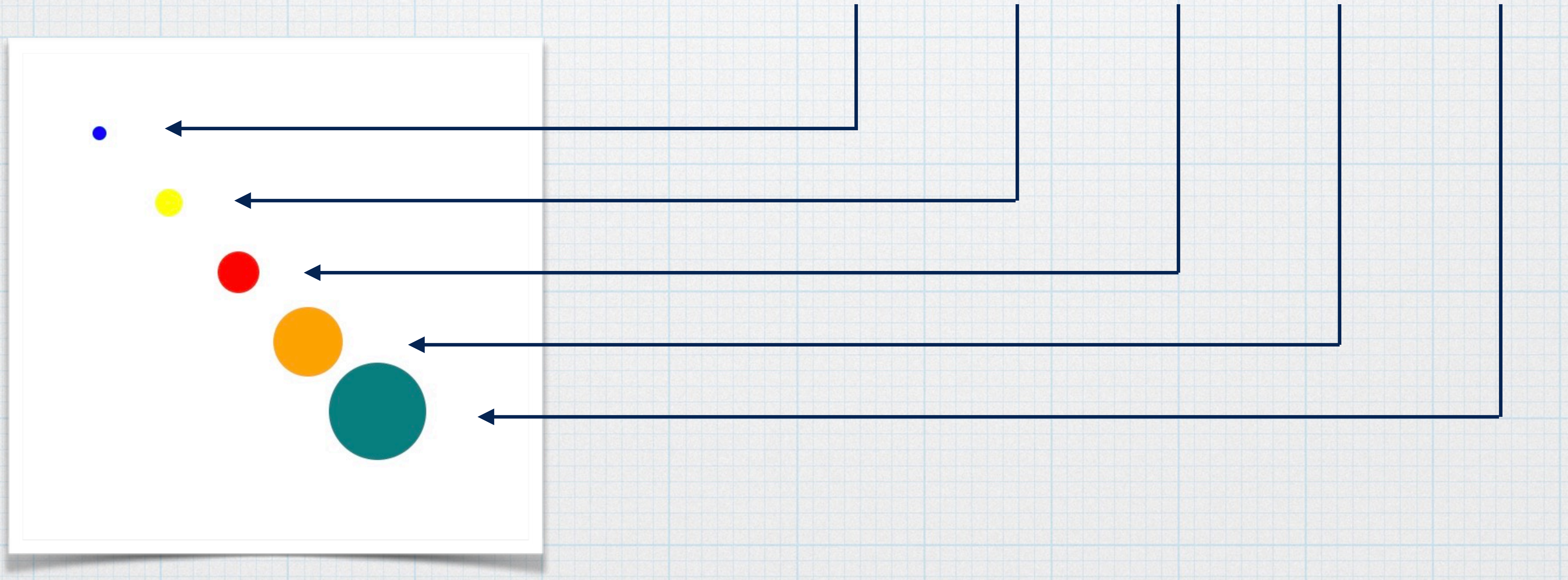
```
var my_data = [1, 2, 3, 5, 7];
```



CALCULATE POSITION

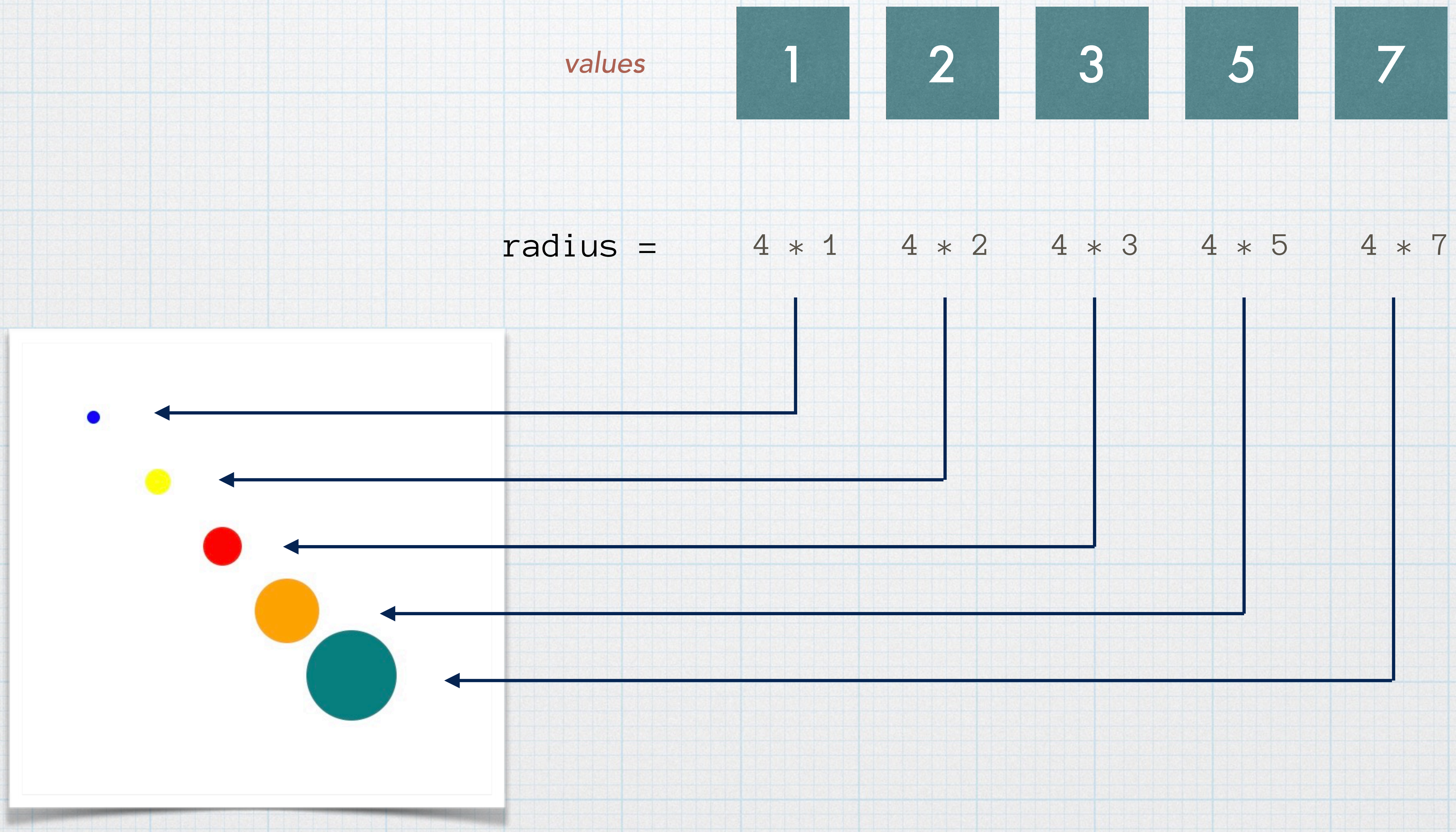
We can calculate both the cx and cy positions by using the array indexes. Push the next circle 40 pixels further right on the x-axis, 40 pixels further down on the y-axis.

values	1	2	3	5	7
indexes	0	1	2	3	4
`cx & cy =`	`0 * 40`	`1 * 40`	`2 * 40`	`3 * 40`	`4 * 40`



CALCULATE SIZE

The radius of each circle should be proportional to the array value. Let's start by making the circle's radius four times bigger than the value.



Append Circles

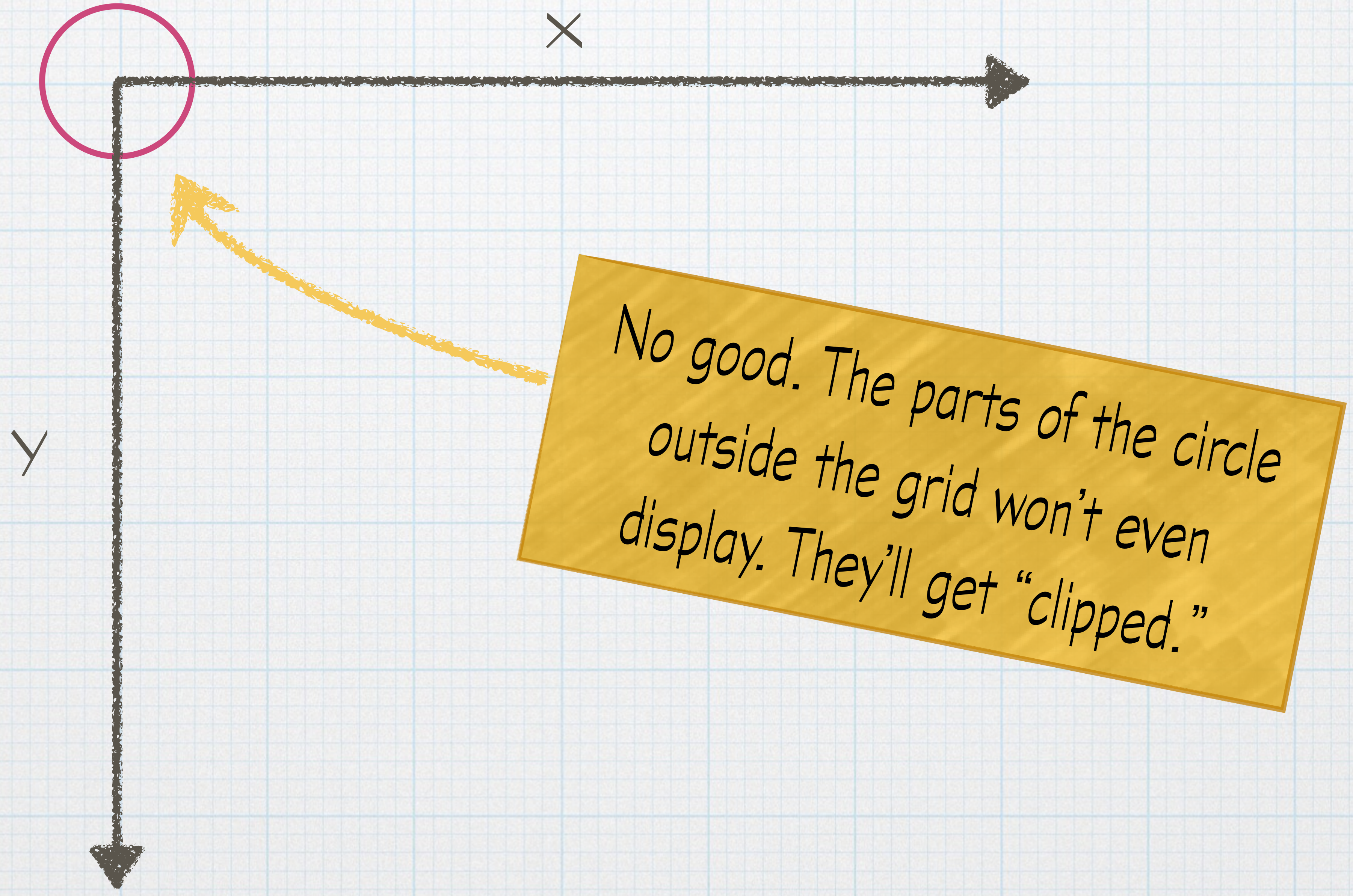
We want to append new circles to our SVG element, calculating both position and size for each new circle:

```
var my_data = [1, 2, 3, 5, 7];
var colors = ['blue', 'yellow', 'red', 'orange', 'teal'];

d3.select('svg')
    .selectAll('circle')
    .data(my_data)
    .enter()
    .append('circle')
    .attr('cx', function(data, index) {return 40 * index + 25})
    .attr('cy', function(data, index) {return 40 * index + 25})
    .attr('r', function(data) {return 4 * data})
    .attr('fill', function(data, index) {return colors[index]});
```

THAT PESKY ORIGIN

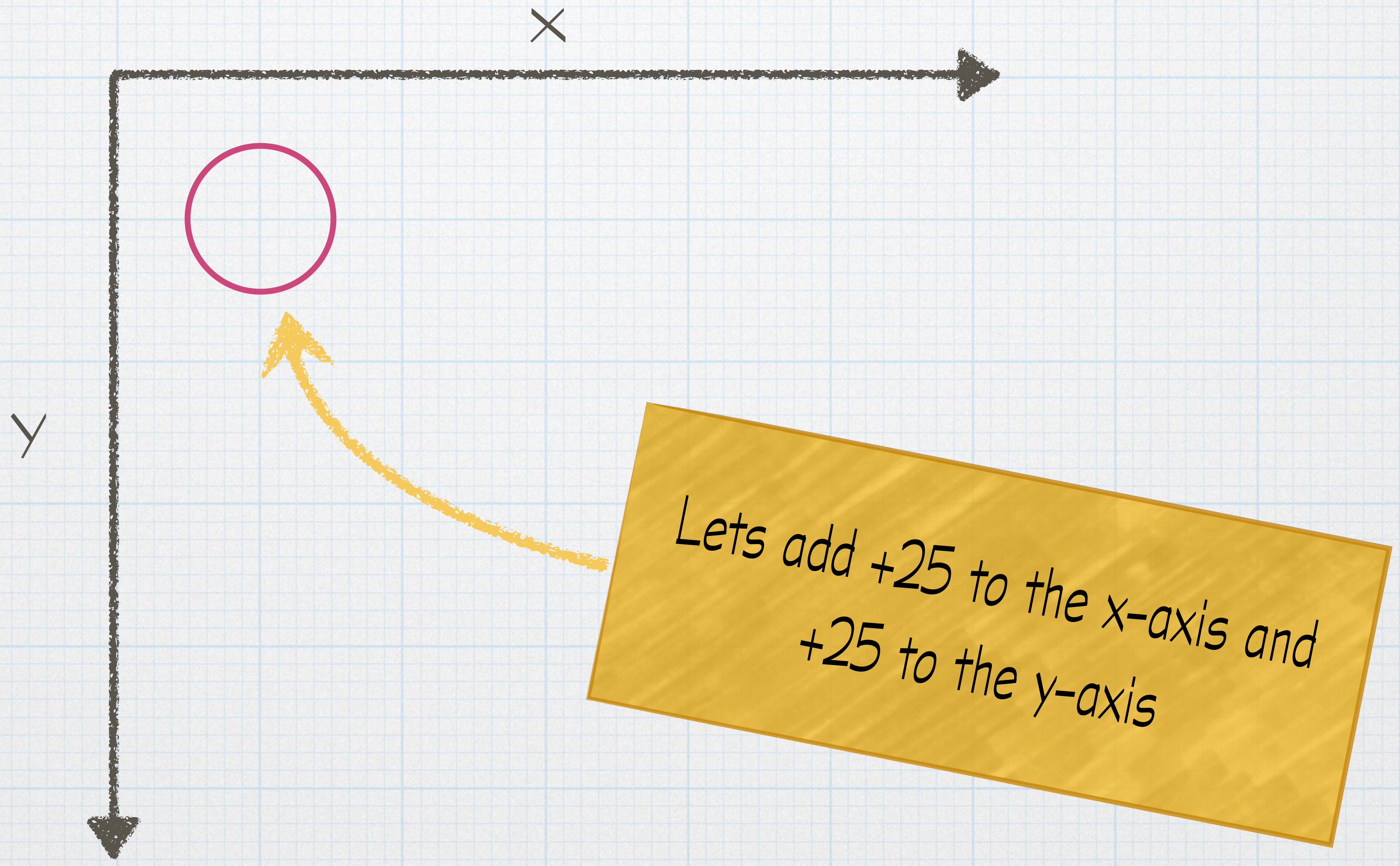
Because our zeroth circle's centre sits at (0, 0), we'll want to slide it over and down a bit:



No good. The parts of the circle outside the grid won't even display. They'll get "clipped."

THAT PESKY ORIGIN

Because our zeroth circle's centre sits at (0, 0), we'll want to slide it over and down a bit:



Append Circles

We want to append new circles to our SVG element, calculating both position and size for each new circle:

```
var my_data = [1, 2, 3, 5, 7];
var colors = ['blue', 'yellow', 'red', 'orange', 'teal'];

d3.select('svg')
    .selectAll('circle')
    .data(my_data)
    .enter()
    .append('circle')
    .attr('cx', function(data, index) {return 40 * index + 25})
    .attr('cy', function(data, index) {return 40 * index + 25})
    .attr('r', function(data) {return 4 * data})
    .attr('fill', function(data, index) {return colors[index]});
```

Slide over and down 25 pixels

Change Colours

We can create an array of colours (spelled however you wish!) and we can assign each new circle the next colour in the array. We can use the index to access the next colour:

```
var my_data = [1, 2, 3, 5, 7];
var colors = ['blue', 'yellow', 'red', 'orange', 'teal'];

d3.select('svg')
    .selectAll('circle')
    .data(my_data)
    .enter()
    .append('circle')
    .attr('cx', function(data, index) {return 40 * index + 25})
    .attr('cy', function(data, index) {return 40 * index + 25})
    .attr('r', function(data) {return 4 * data})
    .attr('fill', function(data, index) {return colors[index]});
```

DATA SELECTION AND BINDING

Let's take a closer look at what D3 needs to do:

```
d3.select('svg')

    .selectAll('circle')

    .data(my_data)

    .enter()
```

Note: we have an entire slideshow later detailing precisely how this works. For now, we'll just get the general overview!

DATA SELECTION AND BINDING

Let's take a closer look at what D3 needs to do:

```
d3.select('svg')

    .selectAll('circle')

    .data(my_data)

    .enter()
```

This statement selects the svg container so that we can do something with it. In typical Javascript fashion, we'll chain more functions, which means that everything that follows acts upon this container.

DATA SELECTION AND BINDING

Let's take a closer look at what D3 needs to do:

```
d3.select('svg')

    .selectAll('circle')

    .data(my_data)

    .enter()
```

Next, we will select all the existing `circle` elements that are already in the `svg` container.

There aren't any, but D3 doesn't know that until it tries to select them.

D3 will now be able to determine whether we have too few circles, too many circles, or just enough circles.

DATA SELECTION AND BINDING

Let's take a closer look at what D3 needs to do:

```
d3.select('svg')

    .selectAll('circle')

    .data(my_data)

    .enter()
```

Next, we correlate every data element to a `circle`. This is called ***binding*** our data. This step allows D3 to map every circle to a data element and every data element will have its very own circle.

Because we had too few circles, D3 will build a new circle for every unmatched data element so that everyone has a partner.

DATA SELECTION AND BINDING

Let's take a closer look at what D3 needs to do:

```
d3.select('svg')

    .selectAll('circle')

    .data(my_data)

    .enter()
```

Next, we define the characteristics of all the new circles that are ***entering*** our canvas. All the chained functions that follow will define the properties of these new circles.

If we had too many circles and wanted to delete them, we could define how they disappear with the `.exit()` function.

DATA SELECTION AND BINDING

This is everything that D3 needs to do:

```
var my_data = [1, 2, 3, 5, 7];
var colors = ['blue', 'yellow', 'red', 'orange', 'teal'];

d3.select('svg')
    .selectAll('circle')
    .data(my_data)
    .enter()
    .append('circle')
    .attr('cx', function(data, index) {return 40 * index + 25})
    .attr('cy', function(data, index) {return 40 * index + 25})
    .attr('r', function(data) {return 4 * data})
    .attr('fill', function(data, index) {return colors[index]});
```

QUICK NOTE ON FUNCTIONS

Notice the function signatures for our D3 anonymous functions. If the function takes only one parameter, it's always the `data` element that we get:

```
function(data) {return 4 * data})
```

Because it's always the same, D3 gurus often shorten that parameter to `d`:

```
function(d) {return 4 * d})
```

If the function takes two parameters, `data` is always first and `index` is always second:

```
function(data, index) {return 40 * index + 25})
```

Because it's always the same, D3 gurus often shorten these parameters to `d` and `i`:

```
function(d, i) {return 40 * i + 25})
```

Arrow Syntax

The newer versions of Javascript have included a new kind of syntax for anonymous functions. It's generally just called the *arrow syntax* or *arrow notation*. Sometimes these are called *arrow functions*. You're welcome to use either style. As we progress, we may lean more towards arrow notation. Here's what the new style looks like:

```
var my_data = [1, 2, 3, 5, 7];
var colors = ['blue', 'yellow', 'red', 'orange', 'teal'];

d3.select('svg')
    .selectAll('circle')
    .data(my_data)
    .enter()
    .append('circle')
    .attr('cx', (data, index) => 40 * index + 25)
    .attr('cy', (data, index) => 40 * index + 25)
    .attr('r', data => 4 * data)
    .attr('fill', (data, index) => colors[index]);
```