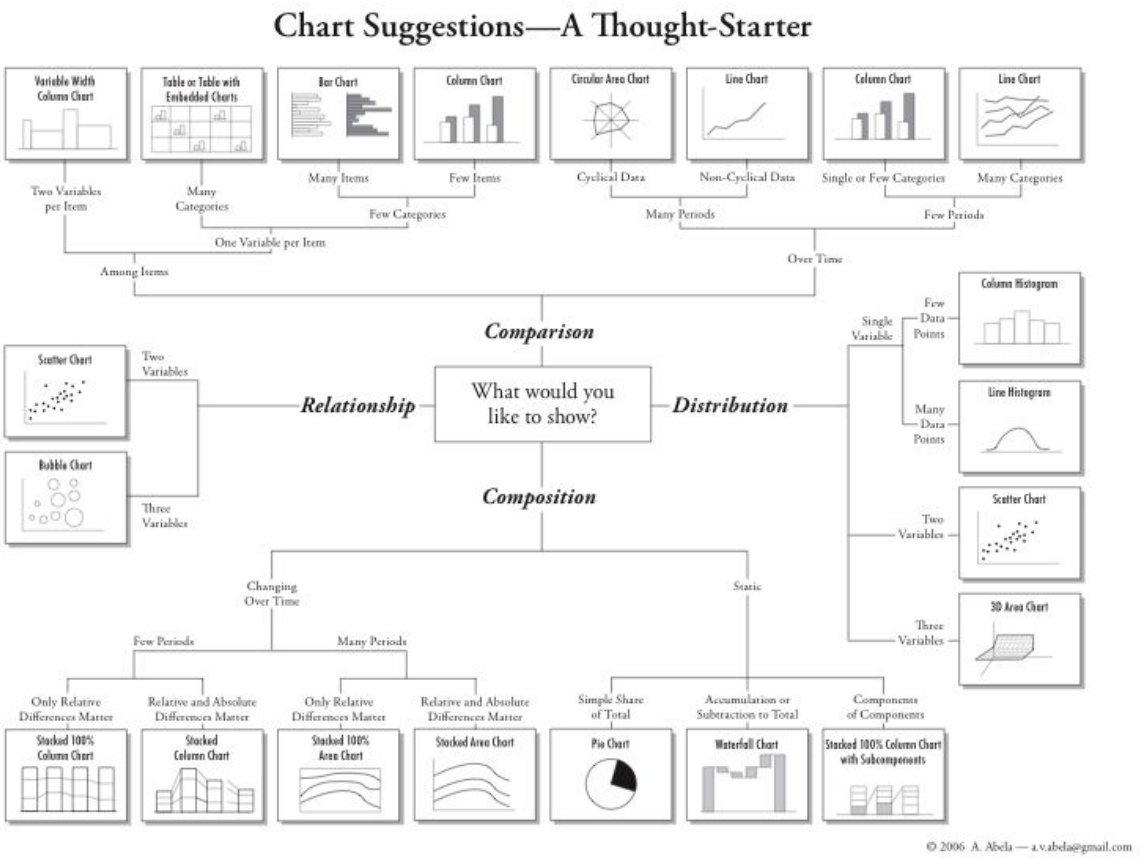
CTG D3 and Data Visualization Reference

Visualization Concepts

Choosing the right Chart

- What kind of insight do you want to convey to your audience?
- How does your data look like?
- How many dimensions you have?

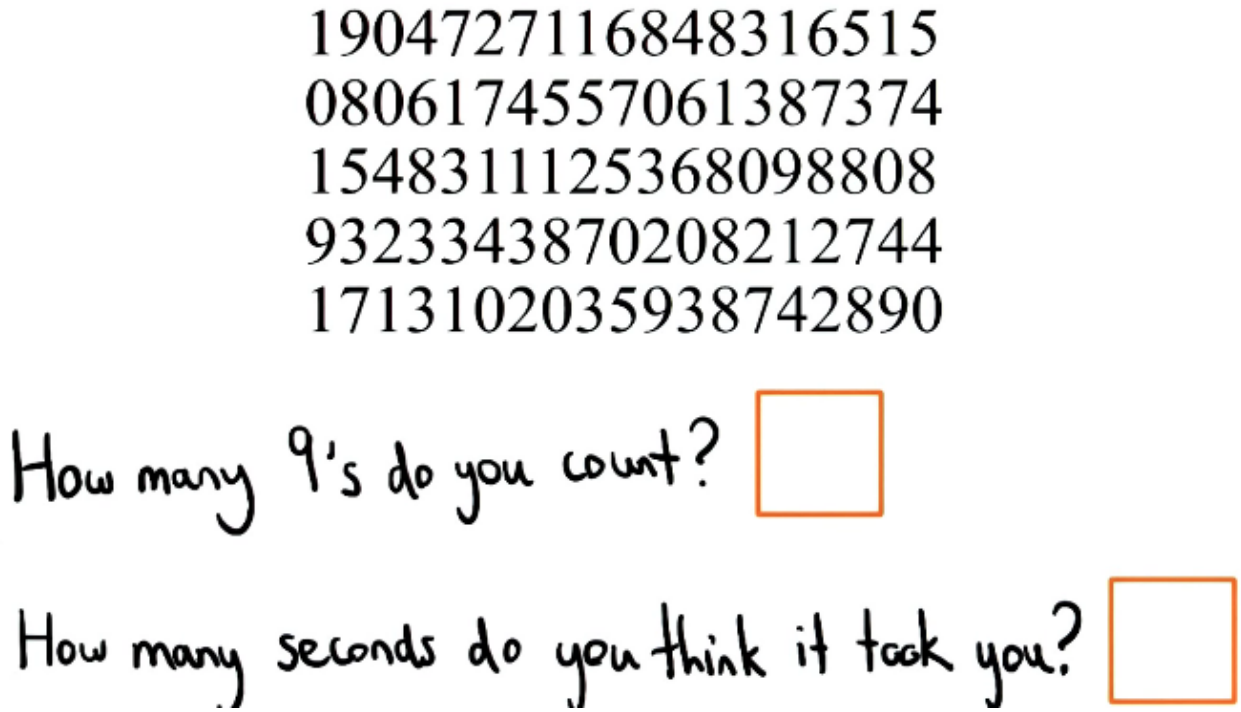
Here is a chart suggestion: (originally from http://extremepresentation.com/choosing_a_good-2/)



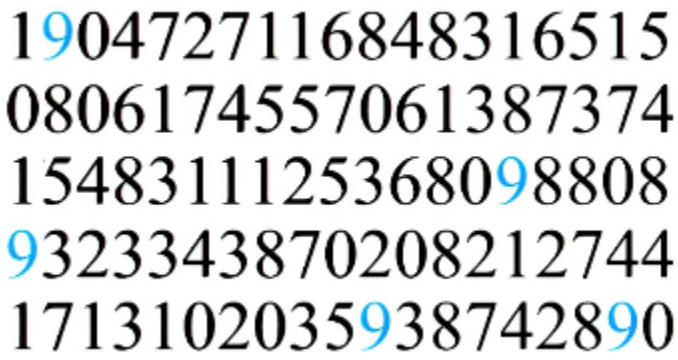
Pre-attentive Processing

In addition the choosing the right char type to present your data, you can leverage good design principles to communicate clearly and efficiently. Pre-attentive Processing allow us to instantly recognize parts of a visualization. Let's examine one aspect of perception that can aid us in the design process.

Take a look at these numbers and the questions.



Now take a look at these:



How long it took this time? Much faster right?

It's pretty incredible such a small change such as color can allow us to recognize and instantly process this digit 9 more quickly.

Here are some real world situations:

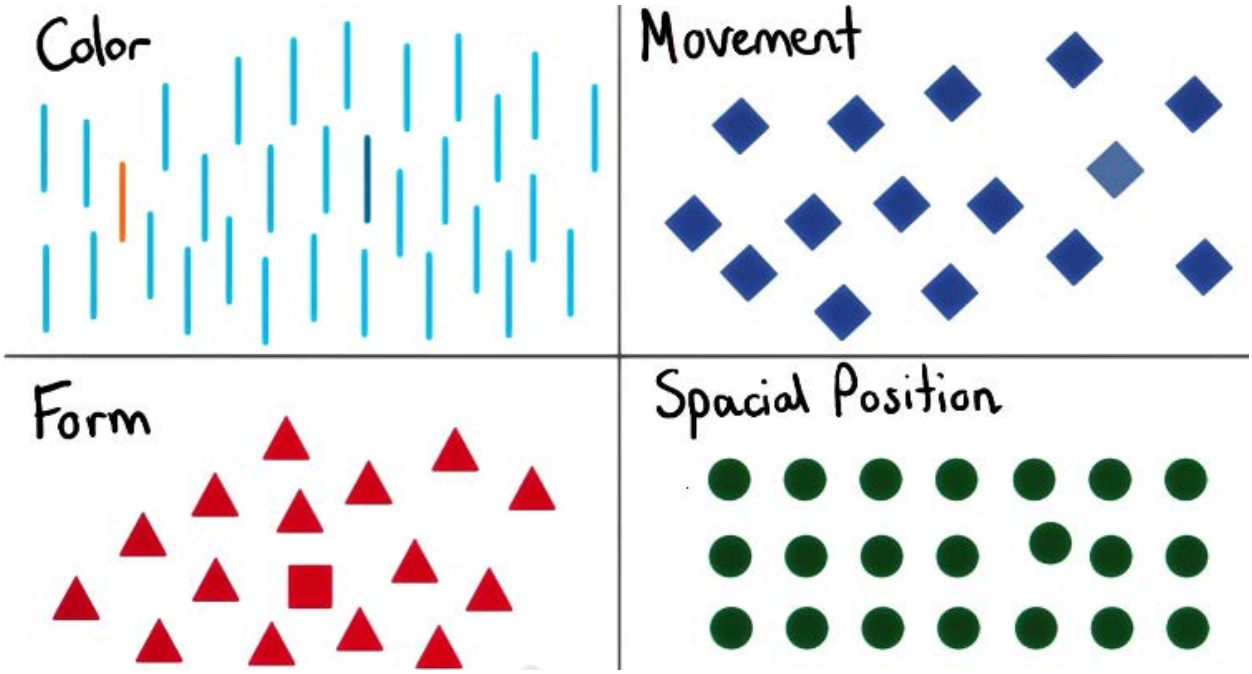
1. 1 millisecond is the duration of light for a typical photo flash strobe.
2. 100 milliseconds is the time interval between gear changes on a Ferrari FXX.
3. 200-250 milliseconds is about the same time that it takes the human brain to recognize emotion in facial expressions.
4. 300-400 milliseconds is about the time that it takes the human eye to blink.
5. 430-500 milliseconds are common modern dance music tempos (120-140 BPM).

What would you guess how fast Pre-attentive Processing happens for you?

The correct answer is 3 (200-250).

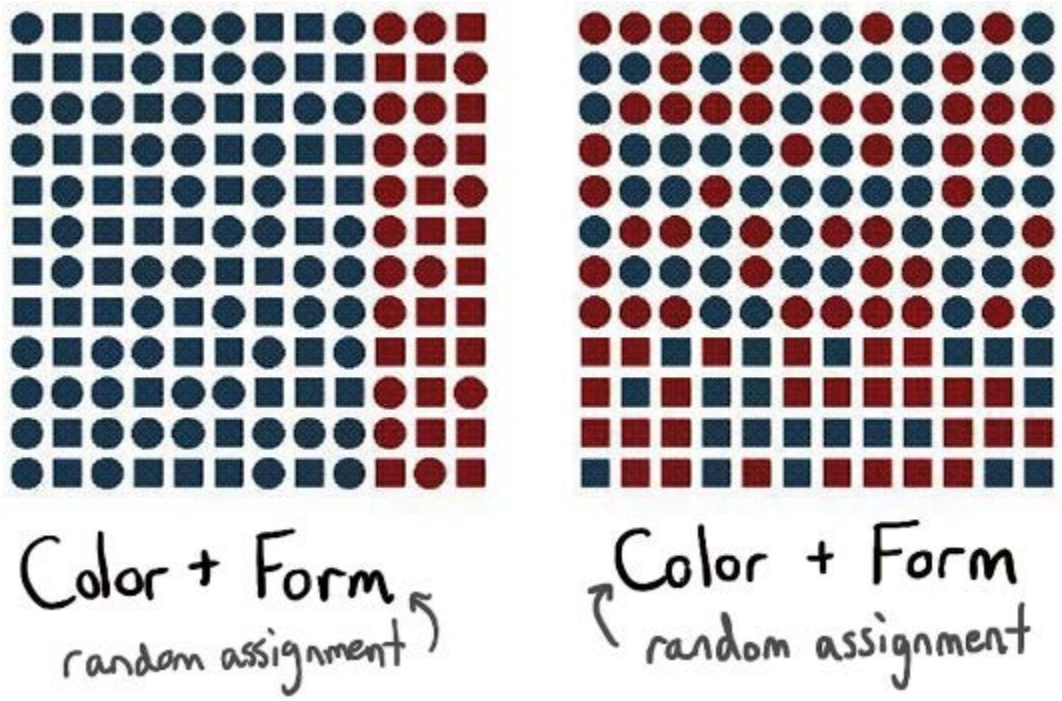
Here is some further information on the topic.

There are several different pre-attentive attributes that can be used to highlight information or to lead the eye to information.



Here and here are some other examples of attributes.

And last an example of them combined



Note how color or shapes were used to create a clear boundary between the information.

Colors

When choosing what color or color hue to use, there are several considerations to make. The following resources provide descriptive insight on the subject.

Stephen Few's, Practical Rules for Using Colors in Charts.

Choosing color pallets: Part 1, Part 2.

Important D3 Concepts

D3 Selectors

In vanilla JavaScript, you typically deal with elements one at a time. For example, to create a divelement, set its contents, and then append it to the body:

This code snippet is JavaScript and should be placed in ascript tag somewhere in the body, after loading D3.

```
var div = document.createElement("div");
div.innerHTML = "Hello, world!";
document.body.appendChild(div);
```

With D3 (as with jQuery and other libraries), you instead handle groups of related elements calledselections. Working with elements en masse gives selections their power; you can manipulate a single element or many of them without substantially restructuring your code. Although this may seem like a small change, eliminating loops and other control flow can make your code much cleaner.

A selection can be created in myriad ways. Most often you create one by querying a selector, which is a special string that identifies desired elements by property, say by name or class ("div" or ".foo", respectively). While you can create a selection for a single element:

```
var body = d3.select("body");
var div = body.append("div");
div.html("Hello, world!");
```

You can just as easily perform the same operation on many elements:

```
var section = d3.selectAll("section");
var div = section.append("div");
div.html("Hello, world!");
```

Chaining Methods

Another convenience of selections is method chaining: selection methods, such as selection.attr, return the current selection. This lets you easily apply multiple operations to the same elements. To set the text color and background color of the body without method chaining, you'd say:

```
var body = d3.select("body");
body.style("color", "black");
body.style("background-color", "white");
```

"Body, body, body!" Compare this to method chaining, where the repetition is eliminated:

```
d3.select("body")
    .style("color", "black")
    .style("background-color", "white");
```

Note we didn't even need a var for the selected body element. After applying any operations, the selection can be discarded. Method chaining lets you write shorter code (and waste less time fretting over variable names).

There is a gotcha with method chaining, however: while most operations return the same selection, some methods return a new one! For example, selection.append returns a new selection containing the new elements. This conveniently allows you to chain operations into the new elements.

The recommended indentation pattern for method chaining is four spaces for methods that preserve the current selection and two spaces for methods that change the selection.

```
d3.selectAll("section")
    .attr("class", "special")
  .append("div")
    .html("Hello, world!");
```

The recommended indentation pattern for method chaining is four spaces for methods that preserve the current selection and two spaces for methods that change the selection.

Since method chaining can only be used to descend into the document hierarchy, use var to keep references to selections and go back up.

```
var section = d3.selectAll("section");

section.append("div")
    .html("First!");

section.append("div")
    .html("Second.");
```

Thinking with Joins

Say you're making a basic scatterplot using D3, and you need to create some SVG circle elements to visualize your data. You may be surprised to discover that D3 has no primitive for creating multiple DOM elements. Wait, WHAT?

Sure, there's the append method, which you can use to create a single element.

Here svg refers to a single-element selection containing an<svg> element created previously (or selected from the current page, say).

```
svg.append("circle")
  .attr("cx", d.x)
  .attr("cy", d.y)
  .attr("r", 2.5);
```

Here svg refers to a single-element selection containing an<svg> element created previously (or selected from the current page, say).

But that's just a single circle, and you want many circles: one for each data point. Before you bust out a for loop and brute-force it, consider this mystifying sequence from one of D3's examples.

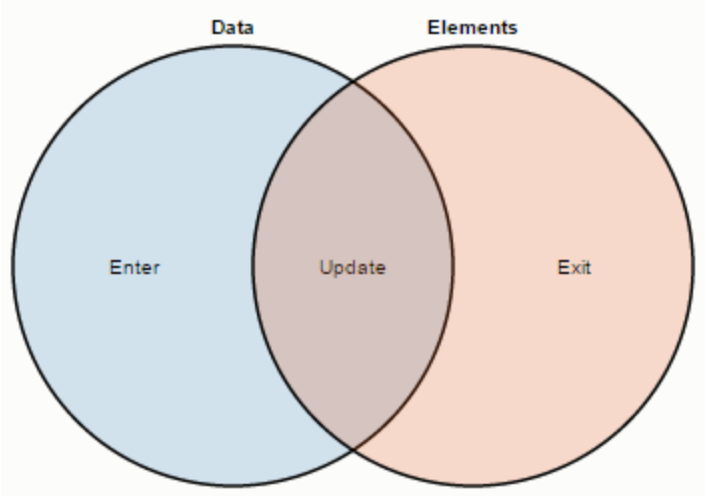
Here data is an array of JSON objects with x and y properties, such as: [{"x": 1.0, "y":1.1}, {"x": 2.0, "y":2.5}, …].

```
svg.selectAll("circle")
    .data(data)
  .enter().append("circle")
    .attr("cx", function(d) {
return d.x; })
    .attr("cy", function(d) {
return d.y; })
    .attr("r", 2.5);
```

Here data is an array of JSON objects with x and y properties, such as: [{"x": 1.0, "y":1.1}, {"x": 2.0, "y":2.5}, …].

This code does exactly what you need: it creates a circle element for each data point, using the x andy data properties for positioning. But what's with the selectAll("circle")? Why do you have to select elements that you know don't exist in order to create new ones? WAT.

Here's the deal. Instead of telling D3 how to do something, tell D3 what you want. You want the circle elements to correspond to data. You want one circle per datum. Instead of instructing D3 to create circles, then, tell D3 that the selection "circle" should correspond to data. This concept is called the data join:



Data points joined to existing elements produce the update (inner) selection. Leftover unbound data produce the enter selection (left), which represents missing elements. Likewise, any remaining unbound elements produce the exit selection (right), which represents elements to be removed.

Now we can unravel the mysterious enter-append sequence through the data join:

1. First, svg.selectAll("circle") returns a new empty selection, since the SVG container was empty. The parent node of this selection is the SVG container.
2. This selection is then joined to an array of data, resulting in three new selections that represent the three possible states: *enter*, *update*, and *exit*. Since the selection was empty, the update and exit selections are empty, while the enter selection contains a placeholder for each new datum.
3. The update selection is returned by selection.data, while the enter and exit selections hang off the update selection; selection.enter thus returns the enter selection.
4. The missing elements are added to the SVG container by calling selection.append on the enter selection. This appends a new circle for each data point to the SVG container.

Thinking with joins means declaring a relationship between a selection (such as "circle") and data, and then implementing this relationship through the three enter, update and exit states.

But why all the trouble? Why not just a primitive to create multiple elements? The beauty of the data join is that it generalizes. While the above code only handles the enter selection, which is sufficient for static visualizations, you can extend it to support dynamic visualizations with only minor modifications for update and exit. And that means you can visualize realtime data, allow interactive exploration, and transition smoothly between datasets!

Here's an example of handling all three states:

```
var circle = svg.selectAll("circle")
    .data(data);
```

```
circle.exit().remove();

circle.enter().append("circle")
    .attr("r", 2.5);

circle
    .attr("cx", function(d) { return d.x; })
    .attr("cy", function(d) { return d.y; });
```

Whenever this code is run, it recomputes the data join and maintains the desired correspondence between elements and data. If the new dataset is smaller than the old one, the surplus elements end up in the exit selection and get removed. If the new dataset is larger, the surplus data ends up in theenter selection and new nodes are added. If the new dataset is exactly the same size, then all the elements are simply updated with new positions, and no elements are added or removed.

Thinking with joins means your code is more *declarative*: you handle these three states without any branching (if) or iteration (for). Instead you describe how elements should correspond to data. If a given *enter*, *update* or *exit* selection happens to be empty, the corresponding code is a no-op.

Joins also let you target operations to specific states, if needed. For example, you can set constant attributes (such as the circle's radius, defined by the "r" attribute) on enter rather than update. By reselecting elements and minimizing DOM changes, you vastly improve rendering performance! Similarly, you can target animated transitions to specific states. For example, for entering circles to expand-in:

```
circle.enter().append("circle")
    .attr("r", 0)
  .transition()
    .attr("r", 2.5);
```

Likewise, to shrink-out:

```
circle.exit().transition()
    .attr("r", 0)
    .remove();
```

Core D3 Methods

Here is a list of the core D3 methods and a link to their documentation.

d3.json

loads a data file and returns an array of Javascript objects

d3.nest

groups data based on particular keys and returns an array of JSON

d3.scale

converts data to a pixel or color value that can be displayed

d3.layout

applies common transformations on predefined chart objects

d3.selection.append

inserts HTML or SVG elements into a web page

d3.selection.attr

changes a characteristic of an element such as position or fill