Information Visualization with Altair – part 1

LAB COURSE NOTES & EXERCISES

Pere-Pau Vázquez | Information Visualization | 2021 - 2022

Contents

1. Introduction

Altair is a declarative visualization library for Python. In its current version (4.1 as of April 2020), it only supports Python 3.6, due to the deprecation of previous versions of Python.

From the architectural point of view, Altair generates Vega code, which is then executed in a JavaScript environment.

There are other alternative libraries for visualization in Python, but most of them have important limitations such as:

- Too many programming required.
- Not enough support for interaction.
- Steep learning curve.
- Not enough power.

As a result, the developers of Altair focused on creating something that was simple, and powerful at the same time. If the goal is having something with the maximum flexibility, one should turn to something such as D3 over JavaScript.

1.1 DECLARATIVE LANGUAGE

From the point of view of programming, there are two main programming paradigms: imperative and declarative.

The **imperative** paradigm requires the user to specify exactly **how** the tasks accomplish the needs. The user must provide all the steps that accomplish the result.

The **declarative** paradigm focuses on specifying **what** needs to be done, and the way it is realized remains as internal details of the system.

Using a declarative language often has advantages in terms of usability since the amount of code required to write is less. But it has other advantages too, such as that the solutions are easier to specify in terms of *what* needs to be done instead of *how*.

1.2 ALTERNATIVES

There are other alternatives to get the same work done in Python. For instance, Seaborn is an easy-to-use library that is programmed using the imperative paradigm, while Bokeh is a declarative library that can also be used. However, both of them are much less powerful than other alternatives such as *Matplolib*, which, again is imperative, or *Plotly*.

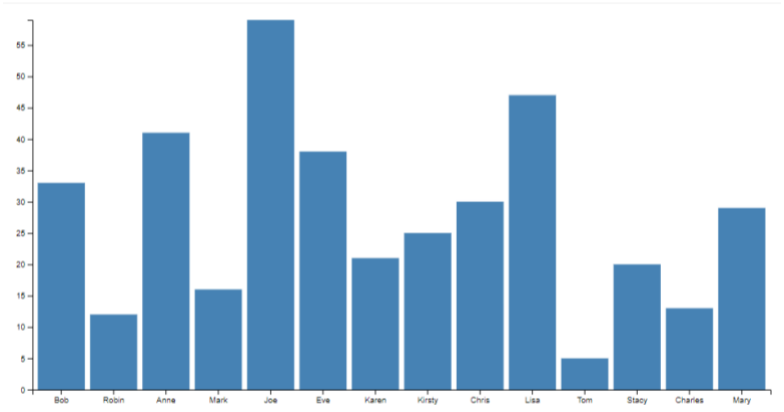
As already mentioned, there are other alternatives outside Python, such as D3, which is an extremely powerful library, over JavaScript. However, working with D3 to design visualizations from scratch, require a much larger number of lines than for doing the same work using Altair.

For example, to create a simple bar chart in D3, we would require a several dozens of lines…

```
.bar { fill: steelblue; }

</style>
<body>

<!-- load the d3.js library -->
<script src="//d3js.org/d3.v4.min.js"></script>
<script>

// set the dimensions and margins of the graph
var margin = {top: 20, right: 20, bottom: 30, left: 40},
    width = 960 - margin.left - margin.right,
    height = 500 - margin.top - margin.bottom;

// set the ranges
var x = d3.scaleBand()
          .range([0, width])
          .padding(0.1);
var y = d3.scaleLinear()
          .range([height, 0]);

// append the svg object to the body of the page
// append a 'group' element to 'svg'
// moves the 'group' element to the top left margin
var svg = d3.select("body").append("svg")
    .attr("width", width + margin.left + margin.right)
    .attr("height", height + margin.top + margin.bottom)
  .append("g")
    .attr("transform",
          "translate(" + margin.left + "," + margin.top + ")");

// get the data
d3.csv("sales.csv", function(error, data) {
  if (error) throw error;

  // format the data
  data.forEach(function(d) {
    d.sales = +d.sales;
  });

  // Scale the range of the data in the domains
  x.domain(data.map(function(d) { return d.salesperson; }));
  y.domain([0, d3.max(data, function(d) { return d.sales; })]);
```

```
  // append the rectangles for the bar chart
  svg.selectAll(".bar")
      .data(data)
    .enter().append("rect")
      .attr("class", "bar")
      .attr("x", function(d) { return x(d.salesperson); })
      .attr("width", x.bandwidth())
      .attr("y", function(d) { return y(d.sales); })
      .attr("height", function(d) { return height - y(d.sales); });

  // add the x Axis
  svg.append("g")
      .attr("transform", "translate(0," + height + ")")
      .call(d3.axisBottom(x));

  // add the y Axis
  svg.append("g")
      .call(d3.axisLeft(y));

});
```

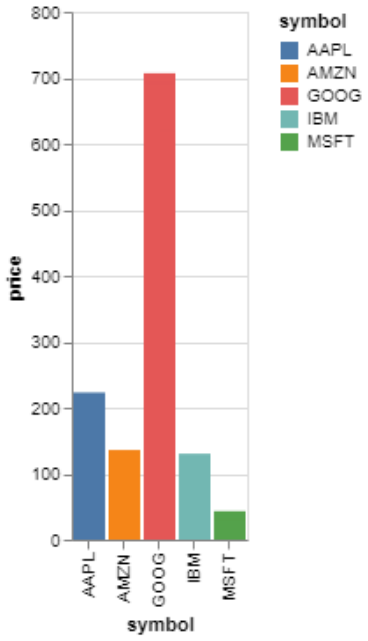
The code above generates the following chart:



On the other hand, building a simple bar chart with Altair requires only less than a dozen lines…

```
import altair as alt
from vega_datasets import data

source = data.stocks.url

alt.Chart(source).mark_bar().encode(
    x='symbol:N',
    y='price:Q',
    color = 'symbol:N'
)
```

The previous code generates the following chart:



2. Data specification

The data is specified to each top-level chart object using a dataset encoded in one of three ways:

- A Pandas DataFrame
- A Data or related object
- An URL string pointing to a *json* or *csv* formatted text file

Pandas Dataframe are two-dimensional size-mutable tabular data structure with labeled axes (rows and columns). See Pandas' documentation for more details on creating those datasets if required (https://pandas.pydata.org). For example, we can create a simple dataset using the following code:

```
import altair as alt
import pandas as pd

data = pd.DataFrame({'x': ['A', 'B', 'C', 'D', 'E'],
                     'y': [5, 3, 6, 7, 2]})
```

Data is a class that can be used to specify data using JSON-style records. To create the same dataset using Data, we should define it as follows:

```
import altair as alt

data = alt.Data(values=[{'x': 'A', 'y': 5},
                        {'x': 'B', 'y': 3},
                        {'x': 'C', 'y': 6},
                        {'x': 'D', 'y': 7},
                        {'x': 'E', 'y': 2}])
```

The main difference between the two encodings is the ability of Altair for extracting the data type from the DataFrame. For Data, we will need to specify the types of the different elements (though we can override the detected type from the DataFrame using the same procedure to specify them in the Chart construction.

We can also input data form an url provided that the data is stored in a JSON file. The following code will do the work:

```
import altair as alt
from vega_datasets import data
url = data.cars.url
```

Like in the previous case, we will need to specify the data types because those cannot be extracted from the file pointed by the URL string.

There are two common conventions for storing data in a DataFrame: long-form and wide-form.

– **Wide-form** data has one row per *independent variable*, with metadata recorded in the *row and column labels*

– **Long-form data** has one row per *observation*, with metadata recorded within the table as *values*

An example of wide-form is provided next:

```
wide_form = pd.DataFrame({'Date': ['2007-10-01', '2007-11-01', '2007-12-01'],
                          'AAPL': [189.95, 182.22, 198.08],
                          'AMZN': [89.15, 90.56, 92.64],
                          'GOOG': [707.00, 693.00, 691.48]})
print(wide_form)
```

```
         Date    AAPL   AMZN    GOOG
0  2007-10-01  189.95  89.15  707.00
1  2007-11-01  182.22  90.56  693.00
2  2007-12-01  198.08  92.64  691.48
```

On the contrary, long-form data would store the information as follows:

```
long_form = pd.DataFrame({'Date': ['2007-10-01', '2007-11-01', '2007-12-01',
                                   '2007-10-01', '2007-11-01', '2007-12-01',
                                   '2007-10-01', '2007-11-01', '2007-12-01'],
                          'company': ['AAPL', 'AAPL', 'AAPL',
                                      'AMZN', 'AMZN', 'AMZN',
                                      'GOOG', 'GOOG', 'GOOG'],
                          'price': [189.95, 182.22, 198.08,
                                    89.15,  90.56,  92.64,
                                    707.00, 693.00, 691.48]})
print(long_form)
```

```
         Date company   price
0  2007-10-01    AAPL  189.95
1  2007-11-01    AAPL  182.22
2  2007-12-01    AAPL  198.08
3  2007-10-01    AMZN   89.15
4  2007-11-01    AMZN   90.56
5  2007-12-01    AMZN   92.64
6  2007-10-01    GOOG  707.00
7  2007-11-01    GOOG  693.00
8  2007-12-01    GOOG  691.48
```

In the documentation, the authors state that Altair works better with the long form version. We can specify that the long-form is used in the creation call, as depicted next.

```
alt.Chart(long_form).mark_line().encode(
  x='Date:T',
  y='price:Q',
  color='company:N'
)
```

We can convert from wide-form to long-form using pandas' *melt* method, or directly in Altair by using the method *transform_fold*.

```
alt.Chart(wide_form).transform_fold(
    ['AAPL', 'AMZN', 'GOOG'],
    as_=['company', 'price']
).mark_line().encode(
    x='Date:T',
    y='price:Q',
    color='company:N'
)
```

If we want to generate charts directly from wide-form, we would require to generate several charts (e.g. one for each company in this example) and use layering to plot them together, such as in the following example.

```
import altair as alt
import pandas as pd


wide_form = pd.DataFrame({'Date': ['2007-10-01', '2007-11-01', '2007-12-01'],
                          'AAPL': [189.95, 182.22, 198.08],
                          'AMZN': [89.15, 90.56, 92.64],
                          'GOOG': [707.00, 693.00, 691.48]})

ch1 = alt.Chart(wide_form).mark_line().encode(
    x='Date:T',
    y='AAPL:Q'
)

ch2 = alt.Chart(wide_form).mark_line().encode(
    x='Date:T',
    y='AMZN:Q'
)

ch3 = alt.Chart(wide_form).mark_line().encode(
    x='Date:T',
    y='GOOG:Q'
)

ch1 + ch2 + ch3
```

We will talk on compound charts later in these course notes.

Finally, Altair supports 4 data types (encodings):

- Quantitative
- Ordinal
- Nominal
- Temporal

These data types can be specified explicitly (when not available from the DataFrame or to override the detected type) by verbosely (e.g. "temporal") or shorthand (e.g. "T") in the *encode* method of the chart.

An example with explicit encoding:

```
alt.Chart(cars).mark_point().encode(
    alt.X('Acceleration', type='quantitative'),
    alt.Y('Miles_per_Gallon', type='quantitative'),
    alt.Color('Origin', type='nominal')
)
```

Explicit encoding:

```
alt.Chart(cars).mark_point().encode(
    x='Acceleration:Q',
    y='Miles_per_Gallon:Q',
    color='Origin:N'
)
```

EXERCISES

Exercise 1. Check the contents of the Vega dataset co2_concentration.json . Plot with the x variable 'Date' as quantitative and y with variable 'CO2' as quantitative. What happens? Now try to use the date as a temporal variable? What happens if you use the date as a categorical variable?

Exercise 2. Create a scatterplot using the dataset gapminder.json with the X axis as the population, and the Y axis as the life expectancy. Now use the X axis as the life expectancy, and the Y axis as the fertility rate.

Exercise 3. Compare the previous chart with another one where the marks are depicted as circles. Make another experiment with the charts as squares.

Exercise 4. With the previous example, now encode in the circle mark the population of the country. Compare the sizes using circles, squares, and points.

Exercise 5. The last plot shows all the data in a single chart, but this includes every year. In order to make sense of them, color code the years.

3. Visualization Pipeline

The common process you have to follow when designing visualization applications includes the following steps:

- Understanding the problem
- Gathering data
- Cleaning data
- Visual encoding design
- View design
- Interaction design
- Evaluation

In the first step, it is necessary to understand what is the type of data we can work with, and which are the questions that the users are asking themselves

about the data. This is a crucial step to successfully design any visualization application.

After that, we need to obtain the datasets. Sometimes, these will be available from Open Data websites such as gapminder.org or the New York City council. However, oftentimes, we need to capture or generate the data ourselves. So this may be a complex, time-consuming process.

The data do analyze will commonly have many artifacts. From missing entries, to format issues, we need to cleanse the data properly. So, for most of the examples provided in the exercises proposed, the first step you need to do is to analyze the data using some software package such as Open Refine, and ensure that the input is correct. For the datasets in the *vega_datasets* module, you can assume they are already clean.

Once the data is clean, and we have a rough idea on what are the types of questions the users want to answer with the data, we can start thinking on the visual encodings, as well as the interaction techniques we want to develop to solve the questions. Note that the solution must follow the *Visualization Mantra*: Overview first, zoom and filter, details on demand. Therefore, the initial view to think of is the one providing an overview of the data. To do this, we need to think of what visual encodings are going to be used. These have two main parts: marks, and visual variables. The marks are the geometric entities that we will use to represent each data unit. The visual variables (or channels) are the attributes (e.g. color, opacity…) that will modify the marks to make the design expressive. In the following sections (*Marks*, *Channels*, *Charts*), we will deal with the different methods to encode data in Altair.

Typically, in any visualization system, several views will be needed. Sometimes an overview and detail, sometimes focus and context… Depending on the problem at hand, we will use one of those techniques. This implies designing different views for the aspects of the data we want to analyze. After the visual encodings, we will deal with the different methods of designing multiple views in the section named *Facets*.

Finally, to provide effective data exploration, we need to design and implement interaction methods that provide different operations for data manipulation, such as selection, zooming, filtering, and so on… We will deal with those in section *Interaction*.

4. Marks

In order to create a visualization, we must transform the data into visual representations. These representations have two different kinds of parameters,

the geometric element we use, and its visual properties. The first is called **mark**, while the second is called **channel** or **visual variables**.

For the initial examples, we are going to use simple charts. More advanced techniques will be presented later.

4.1 BASIC MARKS

There are eight basic mark types:

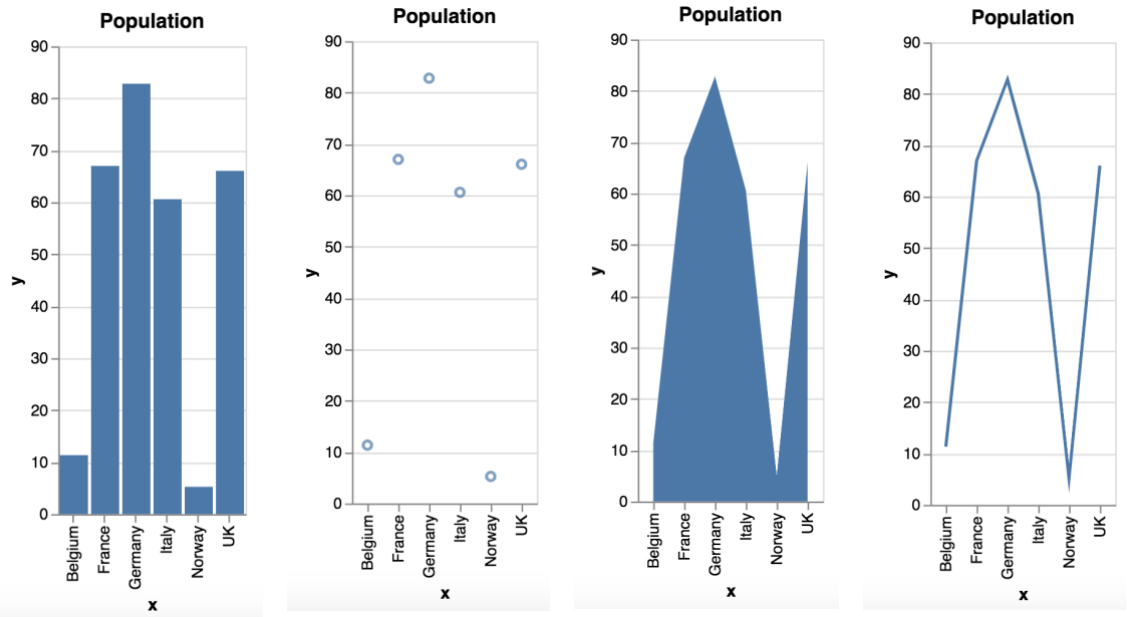
- Area
- Bar
- Circle
- Line
- Point
- Rect
- Square
- Tick

These basic types serve to encode simple elements in charts.

The marks are defined through the use of the *mark_** method, where the symbol "*" must be replaced with the mark name. Thus, for example, to design a bar chart, we will use the method *mark_bar()* of the *Chart* type. The following example builds a bar chart showing the population of a set of countries:

```
import altair as alt
import pandas as pd

data = pd.DataFrame({'x': ['Belgium', 'France', 'Germany',
                           'UK', 'Italy', 'Norway'],
                     'y': [11.35, 66.99, 82.8, 66.04, 60.59, 5.26]})
alt.Chart(data).mark_bar().encode(
    x='x',
    y='y'
).properties(title = 'Population')
```

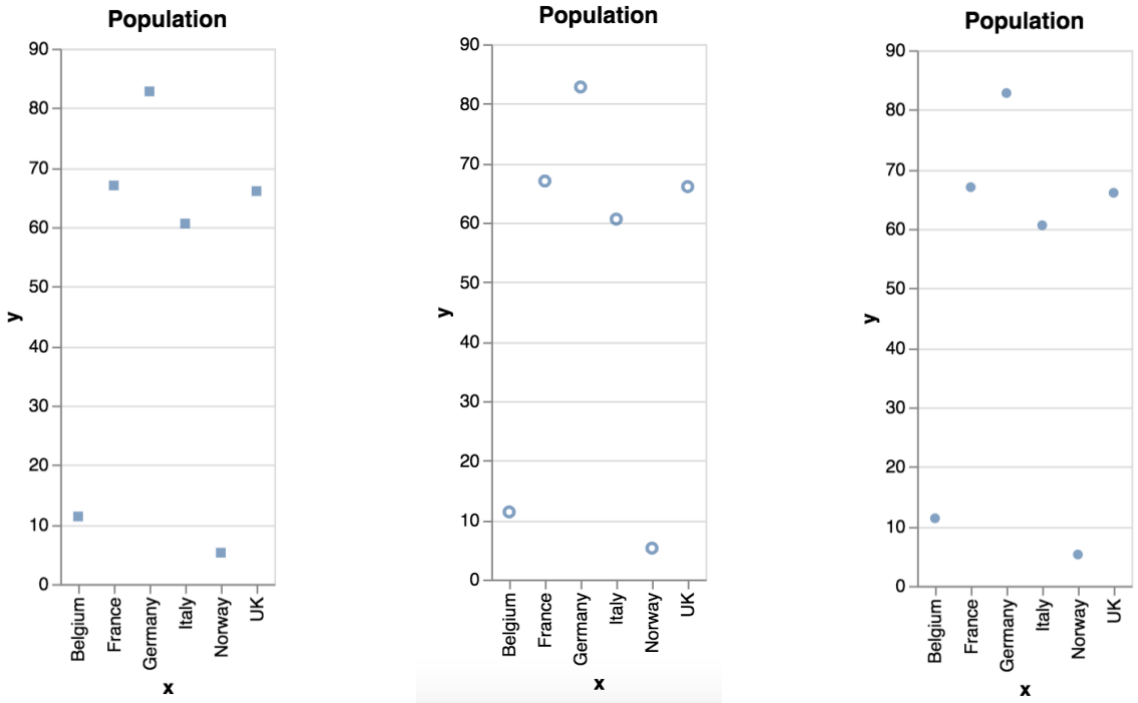
By changing the mark to point, area, or line, you will have these four variants of the same chart:



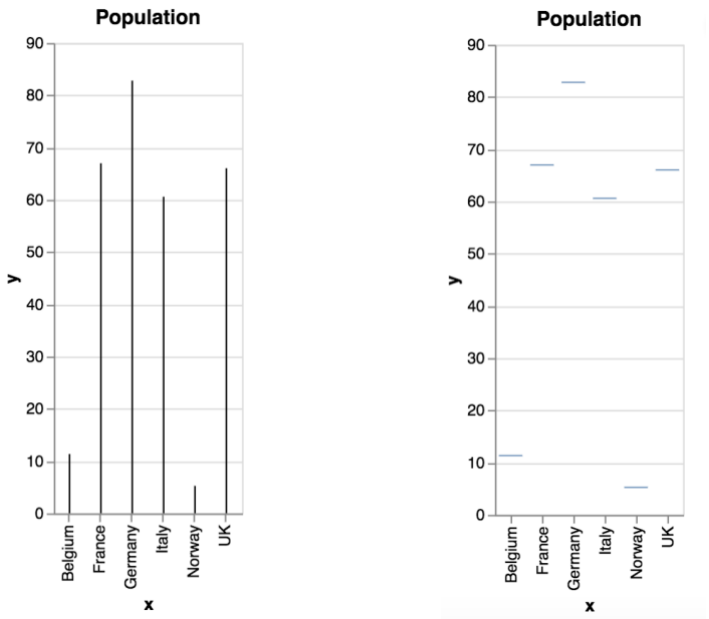
Note that the fact that Altair lets you encode the data in a certain way does not imply that the result is adequate. From the previous examples, only the two leftmost charts are correct, with the first one being better than the second. The rightmost charts may induce the user to read the data as having some continuity from one country to the other, and this is wrong. Therefore, we must carefully design the charts so that they are expressive.

Points, squares and circles can also be used to generate very similar scatterplots. Actually, when we talk about shapes, we will see that the different marks collide partially with the visual channels.

The previous dataset visualized with squares, points, and circles, for comparison, is shown in the following figure.



There are two other basic types that may result not so familiar: rule and tick. Again, we can compare them simply by showing how they plot the same data in a chart.



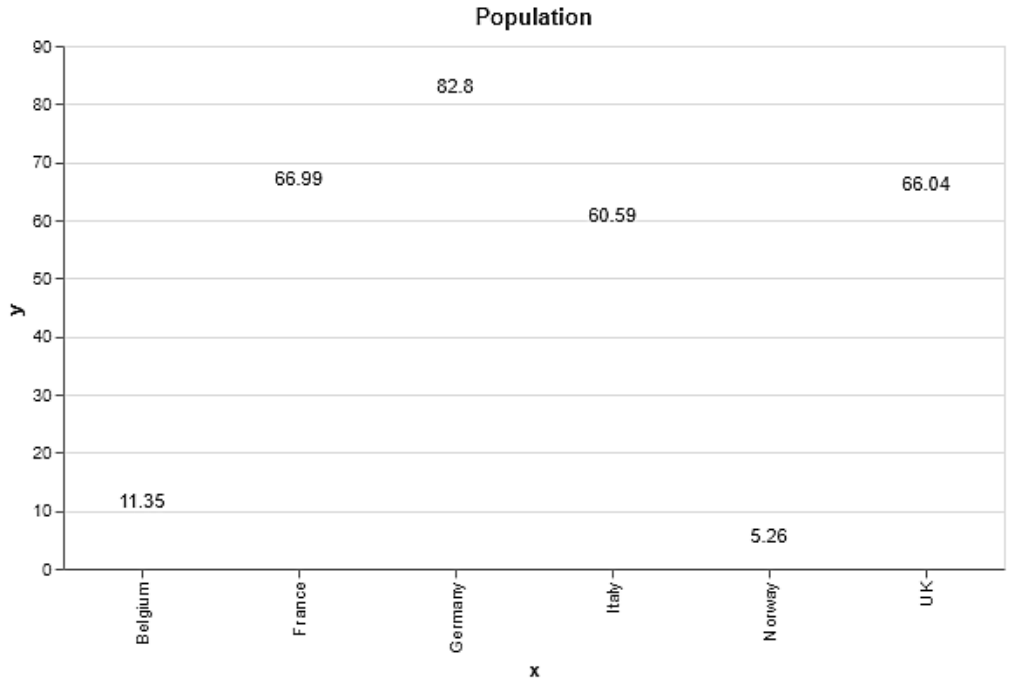
4.2SPECIFIC MARKS

Besides the basic marks, there are three other specific types:

- Geoshape: A geographic shape.
- Rule: a vertical or horizontal line spanning the axis.
- Text. A scatter plot with points represented by text.

We can use the text mark in the previous example to illustrate the population values:

```
import altair as alt
import pandas as pd

data = pd.DataFrame({'x': ['Belgium', 'France', 'Germany',
                           'UK', 'Italy', 'Norway'],
                     'y': [11.35, 66.99, 82.8, 66.04, 60.59, 5.26]})
alt.Chart(data).mark_text().encode(
    x='x',
    y='y',
    text = 'y'
).properties(title = 'Population')
```



We will deal with the other objects in more advanced sessions.

For completeness, let's point that beside these simple marks, Altair also provides some compound marks:

- Box plot: To generate box plots.
- Error band: A continuous band around a line.
- Error bar. An error bar around a point.

Again, we will work on these later in the course.

5. Channels

The visual properties of marks are called channels. Altair has several ways to modify channels. The first one is shape, that, as we saw in the previous section, can be modified using the different *markRef* properties.

5.1 VISUAL PROPERTIES

Some relevant channel modifiers are:

Channels. Color:

- color: default color of the mark
- fill: color that fills the mark (has higher precedence than color)
- fillOpacity: float indicating the opacity [0..1]
- filled: boolean indicating whether the mark is filled
- opacity: float indicating the overall opacity [0..1]
- strokeOpacity: float indicating the stroke opacity [0..1]

For defining the shape and position, Altair provides these channel modifiers:

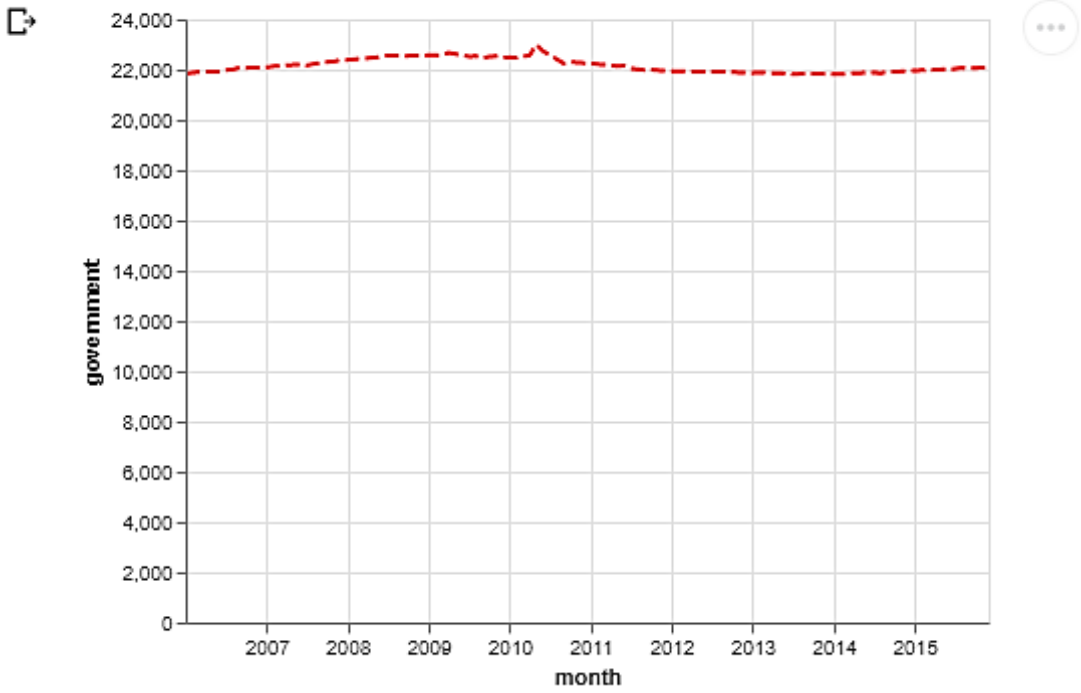
- height: height of the marks
- shape: for point marks, shape can be:
    - circle, square, cross, diamond, triangle up, triangle down, triangle right, or triangle left
- Other shapes: arrow, wedge, triangle
- A custom SVG path (defined in a rectangle between -1 and 1)
- size: the size of the shape. For point, circle and square, it will be the pixel area of the marks.
- x: X coordinates of the marks, or width of horizontal bars (and area marks).
- y: Y coordinates of the marks, or height of vertical bars (and area marks).
- x2: X2 coordinates for ranged shapes (area, bar, rect, and rule)
- y2: Y2 coordinates for ranged shapes (area, bar, rect, and rule)
- width: width of the marks.
-

Other properties of the marks refer to the strokes:

- stroke: Default color for the stroke. It has higher precedence than default color (defined using config.color)
- strokeDash: An array of alternating stroke and space lengths, for creating dashed or dotted lines, that may depend on the encoding.
- strokeWidth: The width of the stroke, in pixels.
- thickness: thickness of the tick mark.
- tooltip: Tooltip text to show upon mouse hover over the object.

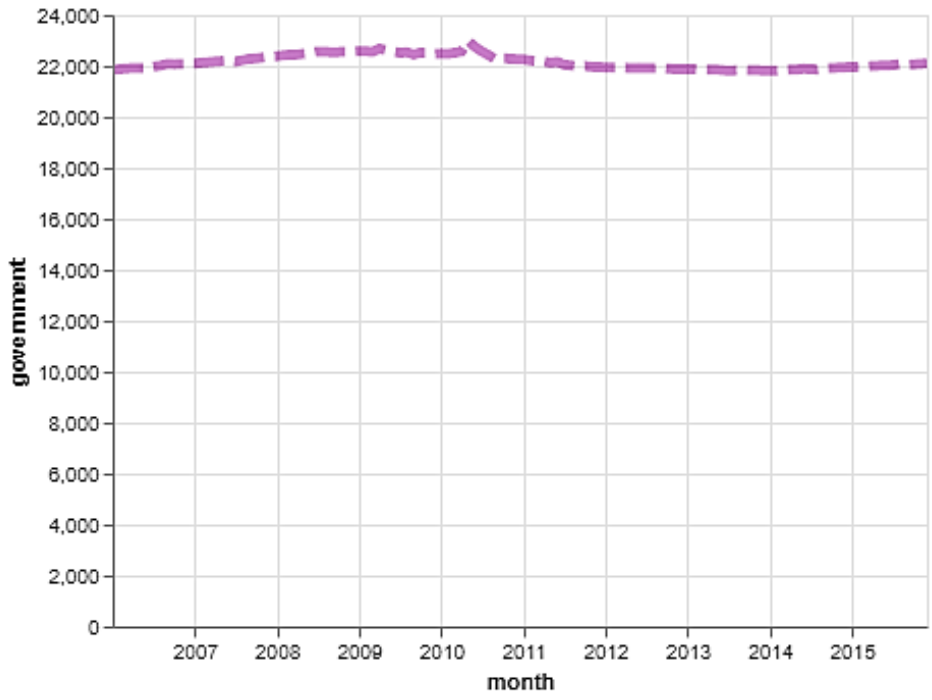
In the following example we use the color and the strokeDash to modify the plot:

```
import altair as alt
import pandas as pd
from vega_datasets import data

df = data.us_employment.url

alt.Chart(df).mark_line(strokeDash=[7,3], color = 'red').encode(
    x='month:T',
    y='government:Q'
)
```



As stated, the *stroke* property has more precedence than the *color* property. In the following chart, we use both, besides changing the stroke width.

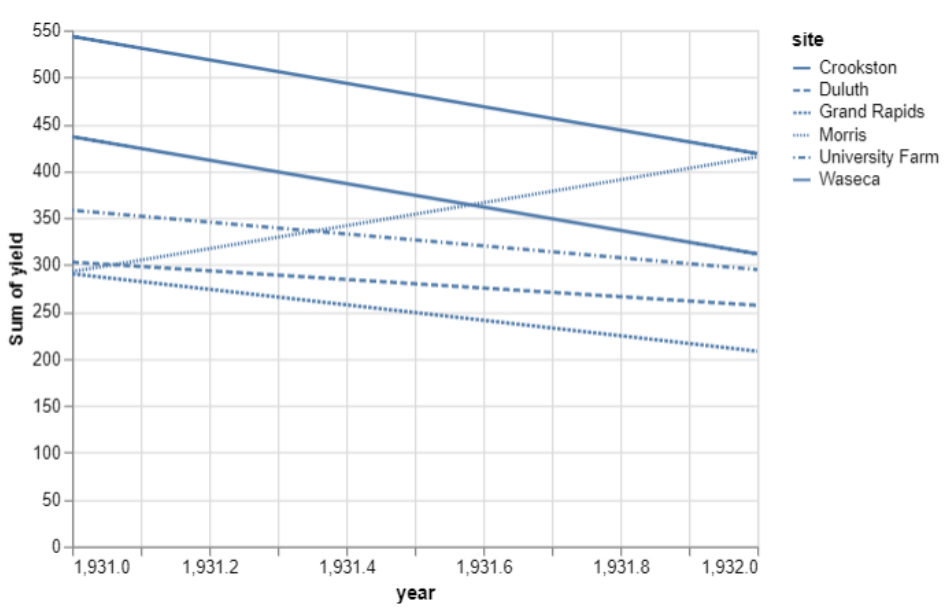
```
import altair as alt
import pandas as pd
from vega_datasets import data

df = data.us_employment.url

alt.Chart(df).mark_line(
    strokeDash=[15,5], color = 'red', stroke = 'purple',
    strokeWidth = 5, strokeOpacity = 0.5
    ).encode(
    x='month:T',
    y='government:Q'
)
```



In the following example, we use the parameter to apply different styles to the different sources of barley in the chart.

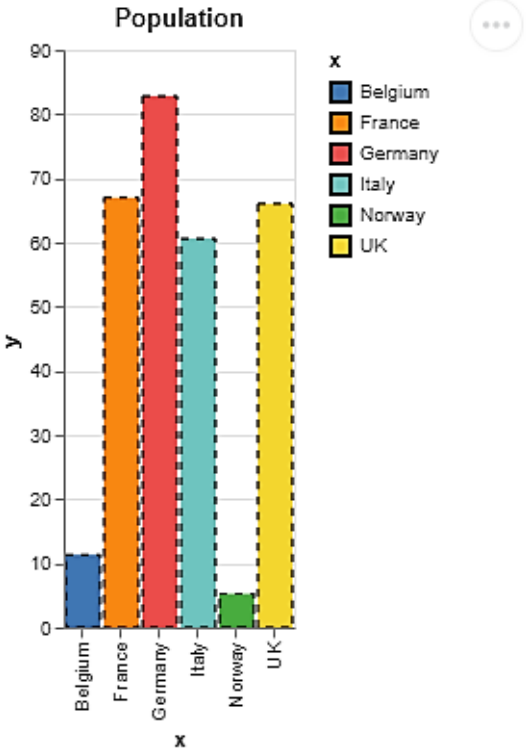
```
import altair as alt
from vega_datasets import data

source = data.barley()

alt.Chart(source).mark_line().encode(
    x=alt.X('year'),
    y='sum(yield):Q',
    strokeDash='site',
)
```

The result would be:



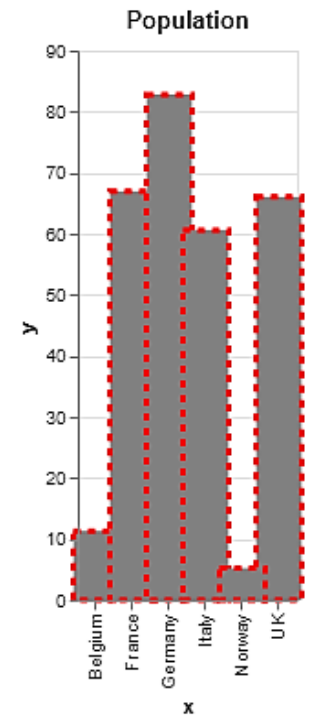
The stroke color can also be used with other marks, such as bars, as in the following example:

```
import altair as alt
import pandas as pd

data = pd.DataFrame({'x': ['Belgium', 'France', 'Germany',
                           'UK', 'Italy', 'Norway'],
                     'y': [11.35, 66.99, 82.8, 66.04, 60.59, 5.26]})
alt.Chart(data).mark_bar(stroke = 'black', strokeDash=[4,4]).encode(
    x='x',
    y='y',
    color = 'x:N'
).properties(title = 'Population')
```



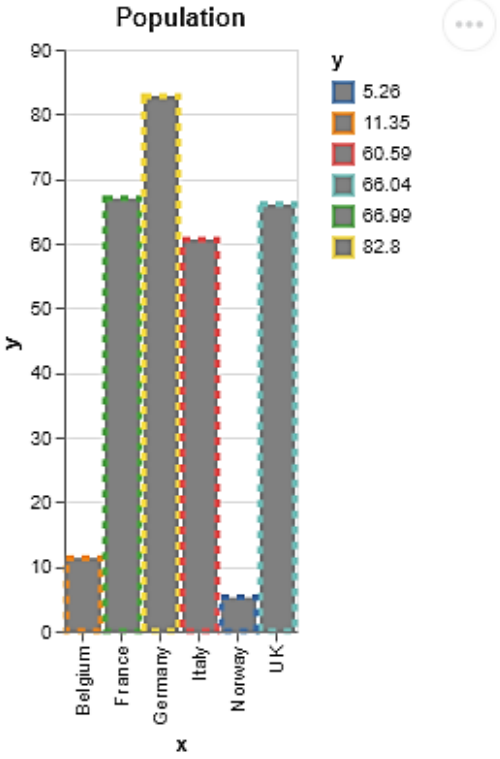
Note that we can also generate plots that are not easy to read, for instance, using colors that do not contrast enough, or by making the marks overlap unnecessarily:

```
import altair as alt
import pandas as pd

data = pd.DataFrame({'x': ['Belgium', 'France', 'Germany',
                           'UK', 'Italy', 'Norway'],
                     'y': [11.35, 66.99, 82.8, 66.04, 60.59, 5.26]})
alt.Chart(data).mark_bar(color = 'pink', fill = 'gray',
                         stroke ='red', strokeDash=[4,4],
                         strokeWidth=3, width=25).encode(
    x='x',
    y='y'
).properties(title = 'Population')
```



We can also use the data to determine the colors of the strokes:

```
import altair as alt
import pandas as pd

data = pd.DataFrame({'x': ['Belgium', 'France', 'Germany',
                           'UK', 'Italy', 'Norway'],
                     'y': [11.35, 66.99, 82.8, 66.04, 60.59, 5.26]})
alt.Chart(data).mark_bar(color = 'pink', fill = 'gray',
                         stroke ='red', strokeDash=[4,4],
                         strokeWidth=3).encode(
    x='x',
    y='y',
    stroke = 'y:N'
).properties(title = 'Population')
```



EXERCISES

Exercise 1. Use the Vega dataset wheat.json . Plot with the x variable 'year' as temporal and y with variable 'wheat' as quantitative. Use a line plot, with a dashed pink line of width 3.

Exercise 2. Use the same dataset to produce a scatterplot with gray triangles filled with blue.

Exercise 3. Increase the size of the previous triangles.

Exercise 4. With the cars dataset, render a scatterplot of the acceleration versus the horsepower, with the points encoded as red crosses with black stroke, and their size proportional to the horsepower.

Exercise 5. Modify the previous plot so that the marks are circles with an opacity of 0.25 and the outline is the same color than the filling color.

5.2 AGGREGATION AND BINNING

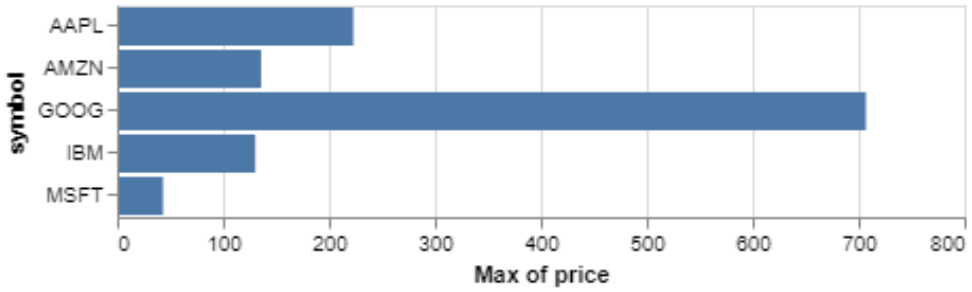
Channels can also be configured with extra options that perform operations on the data, such as aggregation and binning. As might be expected, the operations that can be applied depend on the type of data.

Options of *x* and *y* encodings:

- aggregate: An aggregation function is applied to the field, such as mean, sum, median, etc.
- axis: Modifies the properties of the axes.
- bin: It is used as a flag for binning quantitative fields.
- scale: Can be used to scale properties proportional to the data. If it is disabled, the data is directly encoded.
- sort: Defines the sort order of the encoded field.
- stack: Used to stack values of x or y if they encode values of continuous domains.
- title: Defines a title for the field.

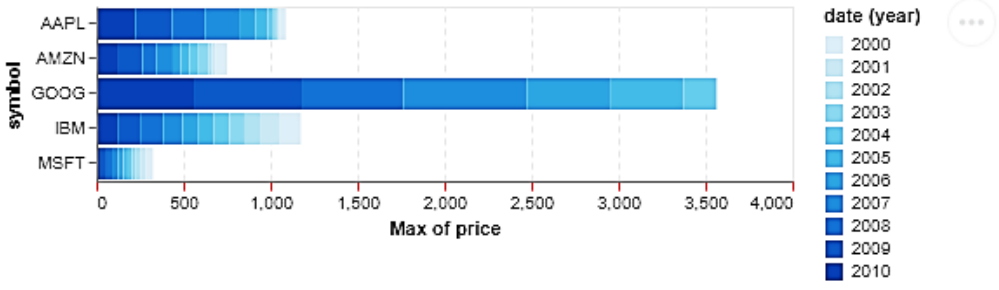
Aggregating data is simple enough, if we want to calculate the average price of each company (defined as 'symbol' in the *stocks* dataset), we can ask Altair to calculate its average from the field itself (we put the values in *x* to improve the space usage):

```
import altair as alt
import pandas as pd
from vega_datasets import data

df = data.stocks.url

alt.Chart(df).mark_bar(
    ).encode(
    x='max(price):Q',
    y='symbol:N',
)
```



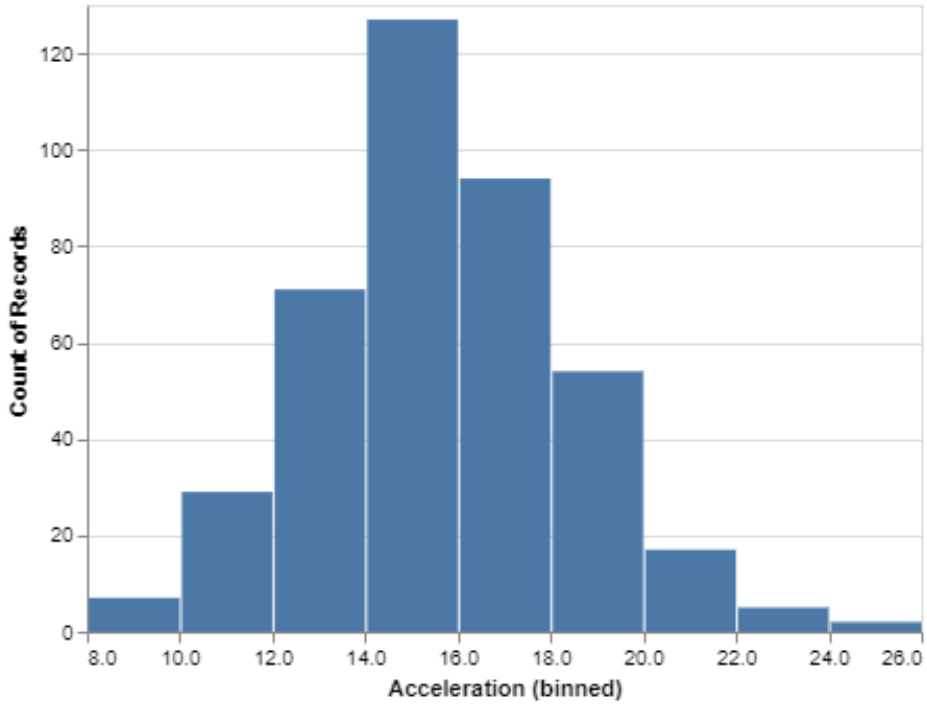
If we want to check the maximum price stock per year, we can separate per years the following way:

```
import altair as alt
import pandas as pd
from vega_datasets import data

df = data.stocks.url

alt.Chart(df).mark_bar(
    ).encode(
    alt.X('max(price):Q', axis = alt.Axis(gridDash = [4,3], tickColor='red')),
    y='symbol:N', color='year(date):O'
)
```



In order to build histograms, we can use the *bin* option. In the following example, we separate the cars per acceleration:

```
import altair as alt
import pandas as pd
from vega_datasets import data

df = data.cars.url

alt.Chart(df).mark_bar(
    ).encode(
    x=alt.X('Acceleration:Q', bin=True),
    y='count():Q'
)
```

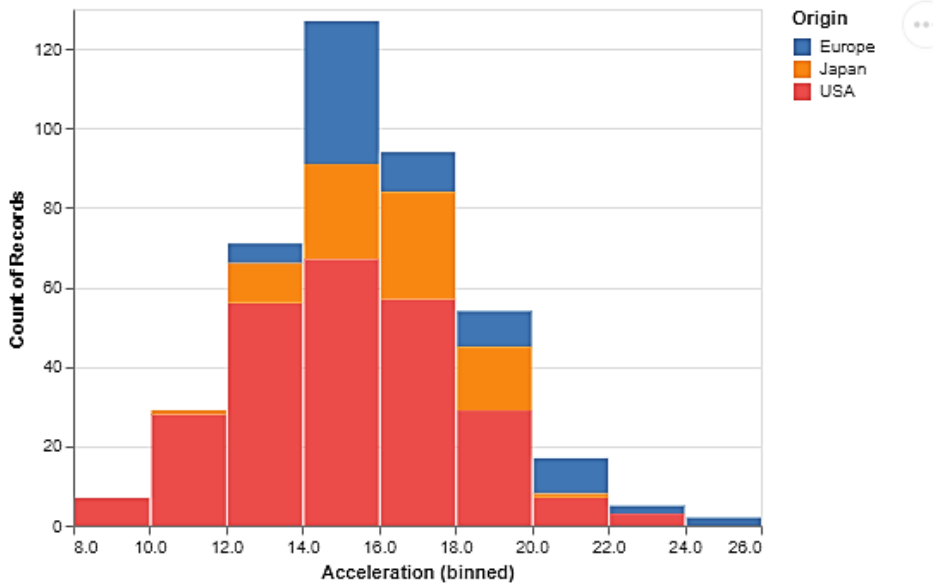


We can separate them per origin, and add a column for each origin by adding the option *column* to the plot:

```
import altair as alt
import pandas as pd
from vega_datasets import data

df = data.cars.url

alt.Chart(df).mark_bar(
    ).encode(
    x=alt.X('Acceleration:Q', bin=True),
    y='count():Q',
    column = 'Origin:N'
)
```

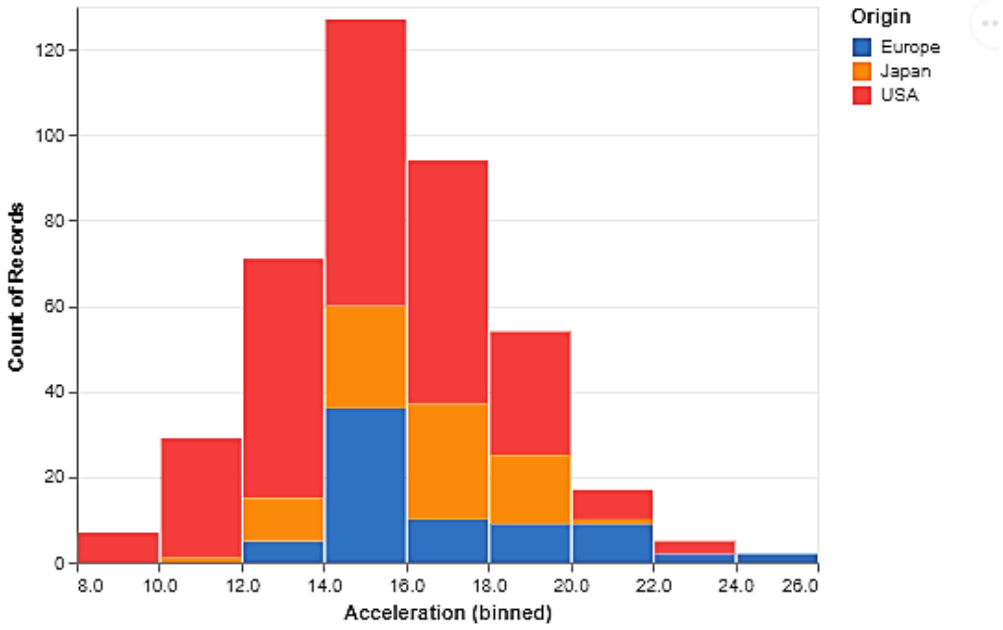
This will generate three bar charts. We could have stacked the bars:

```
import altair as alt
import pandas as pd
from vega_datasets import data

df = data.cars.url

alt.Chart(df).mark_bar(
    ).encode(
    x=alt.X('Acceleration:Q', bin=True),
    y='count():Q',
    color = 'Origin:N'
)
```



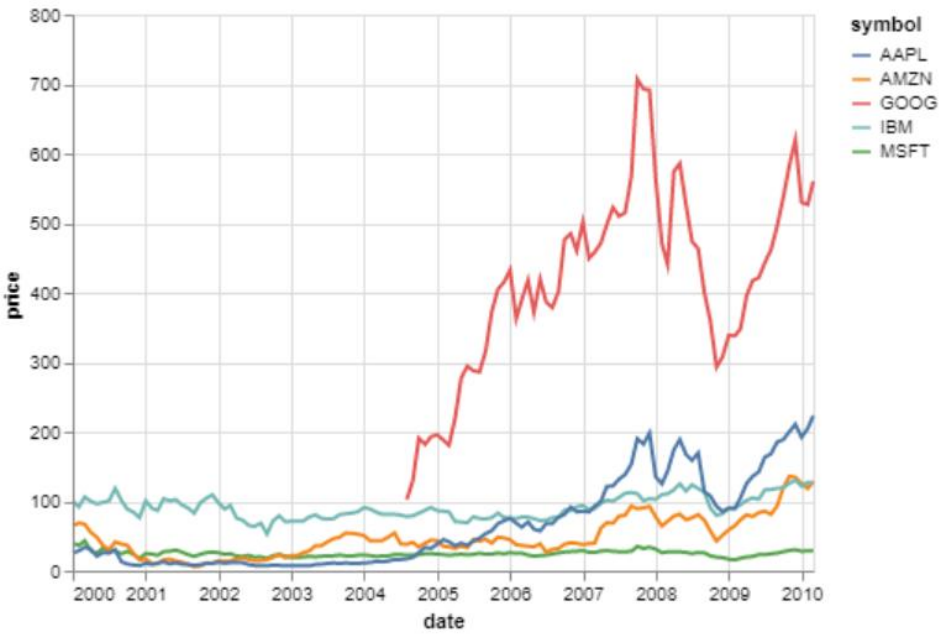
If we want to sort the segments of the bars, we can use the *sort* option:

```
import altair as alt
import pandas as pd
from vega_datasets import data

df = data.cars.url

alt.Chart(df).mark_bar(
    ).encode(
    x=alt.X('Acceleration:Q', bin=True),
    y='count():Q',
    color = 'Origin:N',
    order=alt.Order('Origin:N', sort='ascending')
)
```



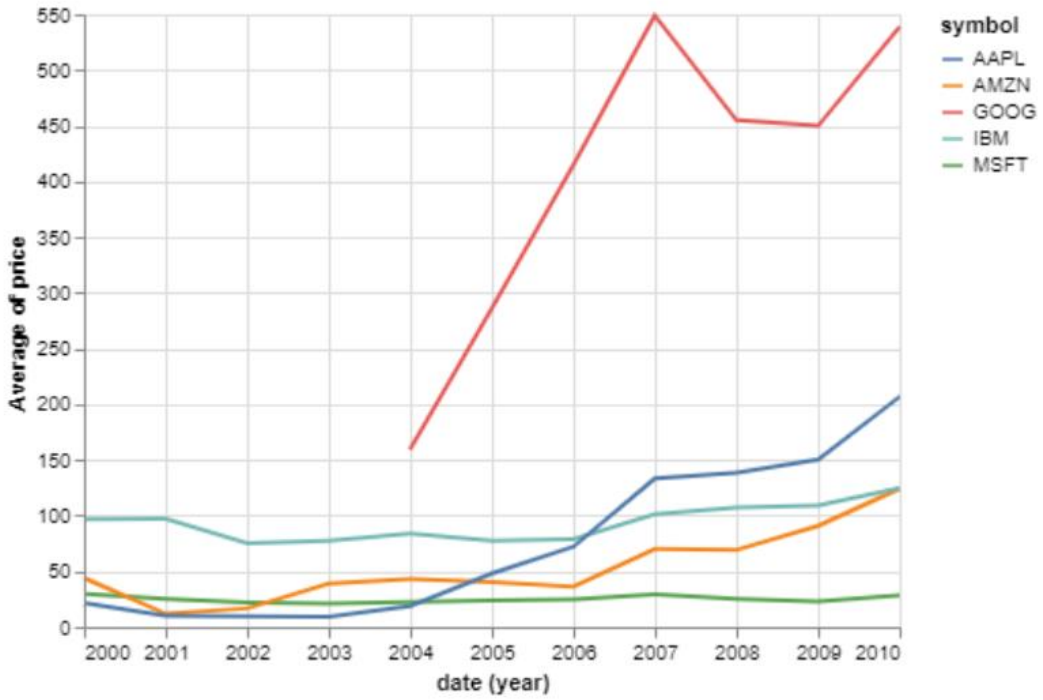
We can also bin time by using the *timeUnit* option, which allows us to group data in different time size slots. For example, the *stocks* dataset can be visualized with the fine grain data:

```
import altair as alt
import pandas as pd
from vega_datasets import data

df = data.stocks.url

alt.Chart(df).mark_line(
    ).encode(
    x='date:T',
    y='price:Q',
    color='symbol:N'
)
```



Or it can be visualized by averaging the values yearly:

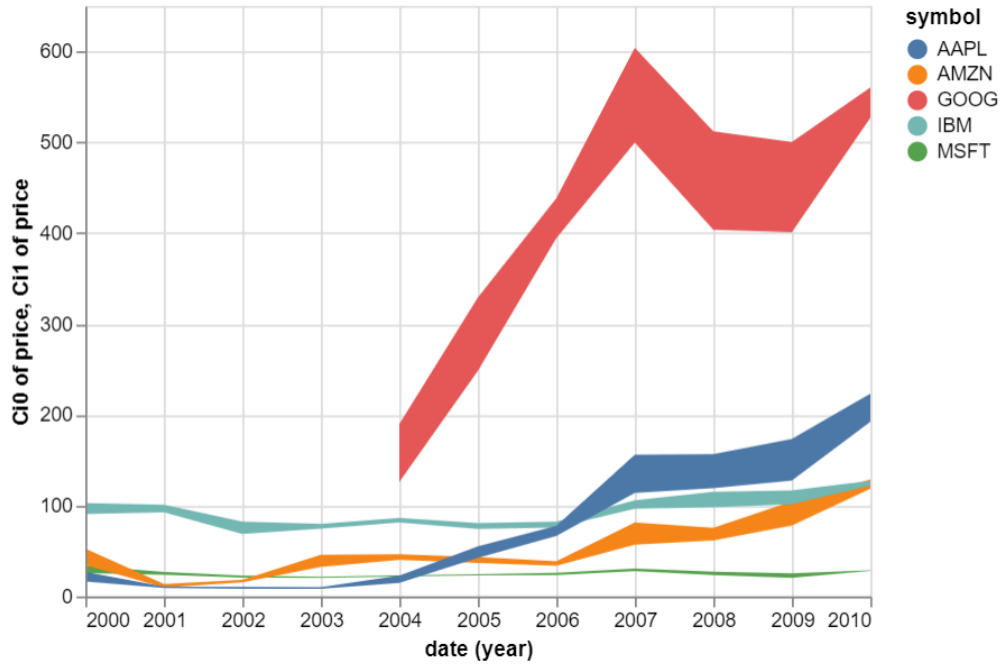
```
import altair as alt
import pandas as pd
from vega_datasets import data

df = data.stocks.url

alt.Chart(df).mark_line(
    ).encode(
    x=alt.X('date:T', timeUnit='year'),
    y='average(price):Q',
    color='symbol:N'
)
```



In this case, we do not know how much the price has changed along the line, so we could add a confidence interval to the function, using *ci0* and *ci1* and encoding the initial and final values of the intervals in the Y axis, by using the options *y* and *y2*:

```
import altair as alt
import pandas as pd
from vega_datasets import data

df = data.stocks.url

alt.Chart(df).mark_area().encode(
    x=alt.X('date:T', timeUnit='year'),
    y='ci0(price):Q',
    y2='ci1(price):Q',
    color='symbol:N'
)
```

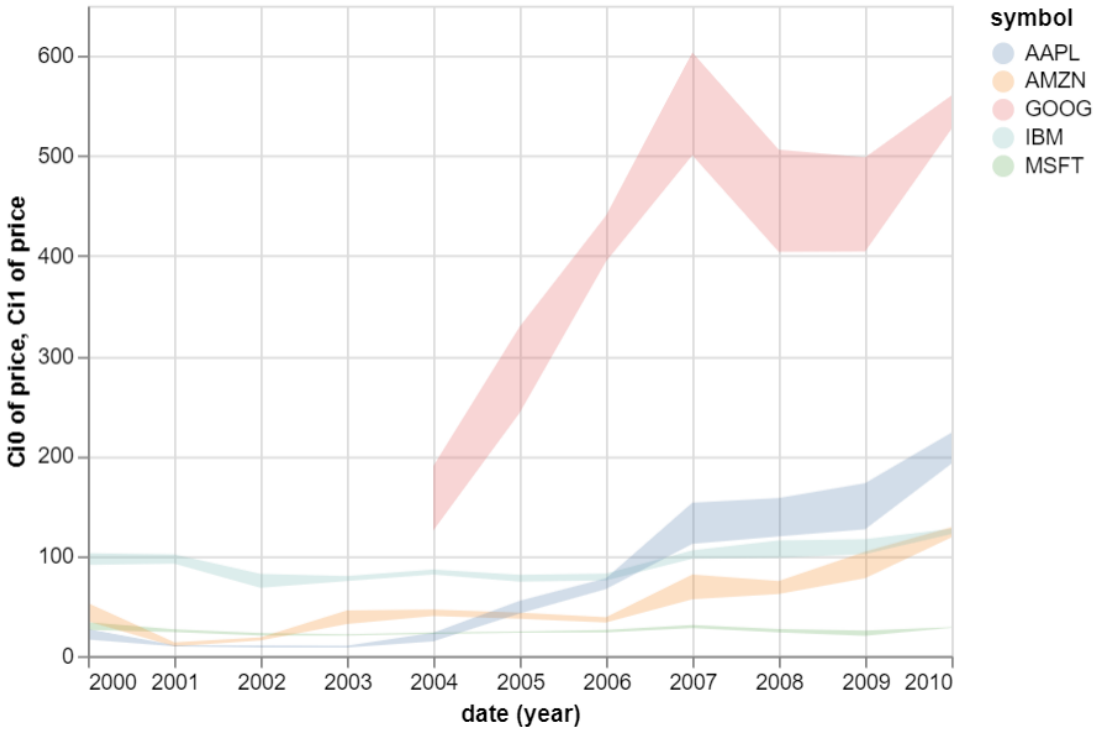
Note that now we use the area mark. And since there are values that overlap, in some regions it is not possible to see exactly where the different values start and end.



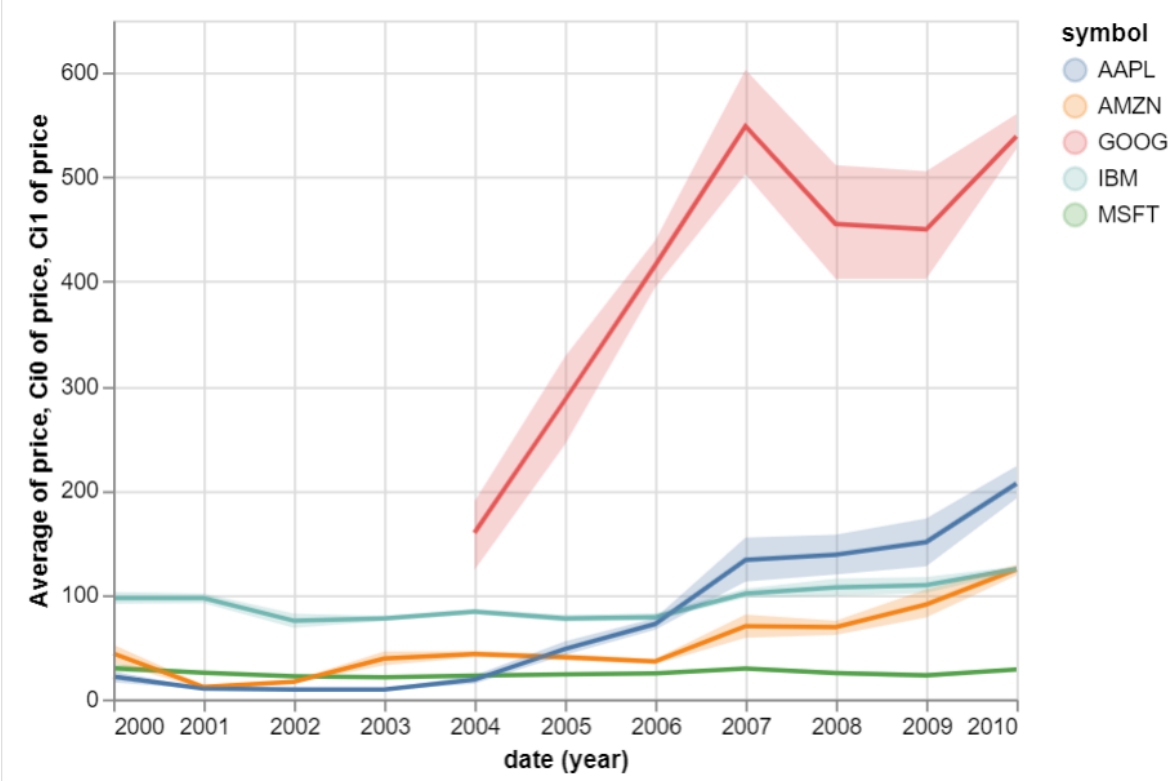
We can improve this by changing the opacity:

```
import altair as alt
import pandas as pd
from vega_datasets import data

df = data.stocks.url

alt.Chart(df).mark_area(opacity=0.25
    ).encode(
    x=alt.X('date:T', timeUnit='year'),
    y='ci0(price):Q',
    y2='ci1(price):Q',
    color='symbol:N'
)
```

Which results in a chart that can be better interpreted.



We can also combine both charts by overlapping them as we saw previously, and the result would be:

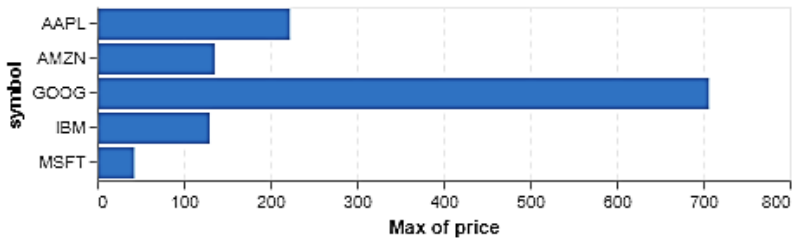


5.3 CUSTOMIZATION OPTIONS

If we need to add other configurations, we can use the alternative naming for the encodings: *alt.X* for the X axis, and *alt.Y* for the Y axis. Those are functions that accept several parameters (separated by commas) that can be used to further configure the properties.

If we want to change axis properties, such as adding a dashing style to the grid in the chart, we can do it the following way:

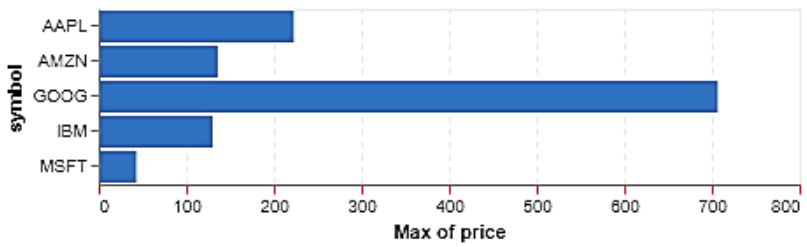
```
import altair as alt
import pandas as pd
from vega_datasets import data

df = data.stocks.url

alt.Chart(df).mark_bar(
    ).encode(
    alt.X('max(price):Q', axis = alt.Axis(gridDash = [4,3])),
    y='symbol:N',
)
```



Note that we use the same syntax for the grid dashing than for the strokes.

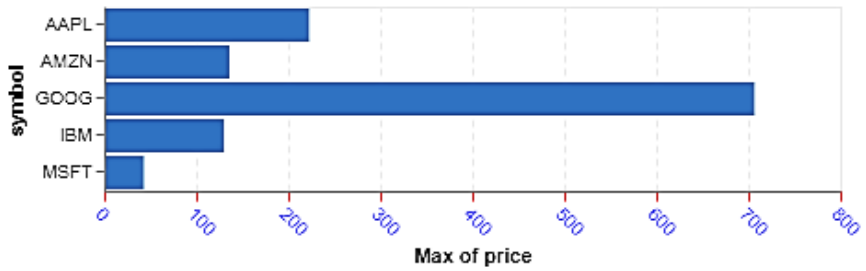
In the same manner, we can also add other features, such as colors to the ticks:

```
import altair as alt
import pandas as pd
from vega_datasets import data

df = data.stocks.url

alt.Chart(df).mark_bar(
    ).encode(
    alt.X('max(price):Q', axis = alt.Axis(gridDash = [4,3], tickColor='red')),
    y='symbol:N',
)
```



Or we can even modify how the labels are shown:

```
import altair as alt
import pandas as pd
from vega_datasets import data

df = data.stocks.url

alt.Chart(df).mark_bar(
    ).encode(
    alt.X('max(price):Q',
          axis = alt.Axis(gridDash = [4,3], tickColor='red',
                          labelColor='blue', labelAngle=45)),
    y='symbol:N',
)
```



There are all sorts of customizations that can be made with these parameters. When you start typing on the editor, the options will appear on a floating window.

But with a great power comes a great responsibility: Like in the previous case, the fact that we can do many modifications to the original layout does not mean that these will be appealing to the user. We have to make sure that we do not add extraneous embellishments that prevent the user to properly perceive the information we are plotting.

There are other customization options that affect the whole layout, instead of just a single object of the chart. Size of the chart can be changed with the *width* and *height* properties.

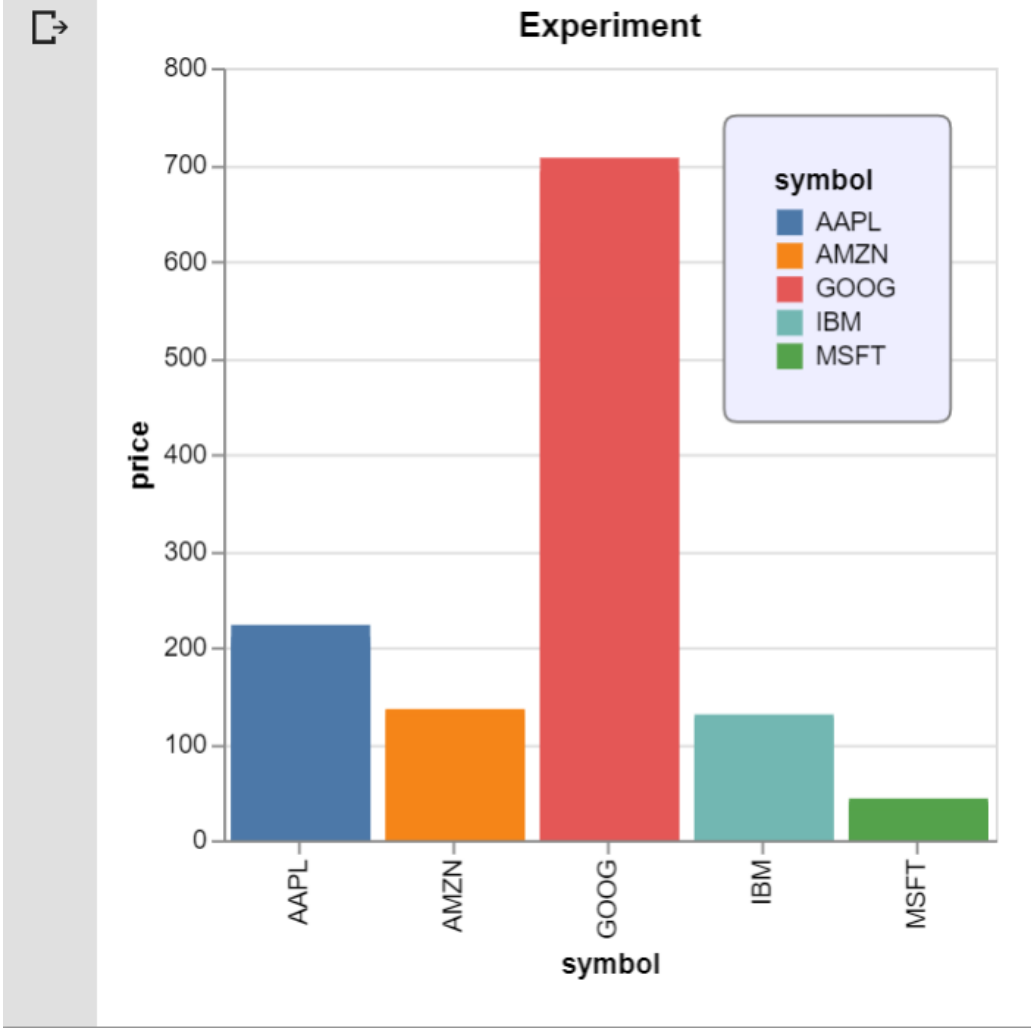
```
import altair as alt
from vega_datasets import data

source = data.stocks.url

alt.Chart(source).mark_bar().encode(
    x='symbol:N',
    y='price:Q',
    color = 'symbol:N'
).properties(width = 100, height = 200)
```

We can also change the title with the *title* property, that receives a string.

Finally, other useful configurable elements are those of the legend. We can change the title, stroke color, fill color, padding, and orientation, by using the *configure_legend* function, as can be seen in the following example:

```
import altair as alt
from vega_datasets import data

source = data.stocks.url

alt.Chart(source).mark_bar().encode(
    x='symbol:N',
    y='price:Q',
    color = 'symbol:N'
).properties(width = 300, height = 300,
             title = 'Experiment').configure_legend(
    strokeColor='gray',
    fillColor='#EEEEFF',
    padding=20,
    cornerRadius=5,
    orient='top-right'
)
```

That would have as a result the following chart:



Changing the title of the legend would require a little circumvention, since it takes the name of the axis. We can, however, disable it with the option *title = null*.

There is another option for the *width* and *height* values, make them depend on the size of the HTML page or container. In order to do so, you only need to change the *width* value to *container*. This will adjust the size to the available, given by the HTML page. The advantage of this feature is that the size of the charts will adapt to window resizing. Note that, however, this function does not seem to run well in Google Colab.

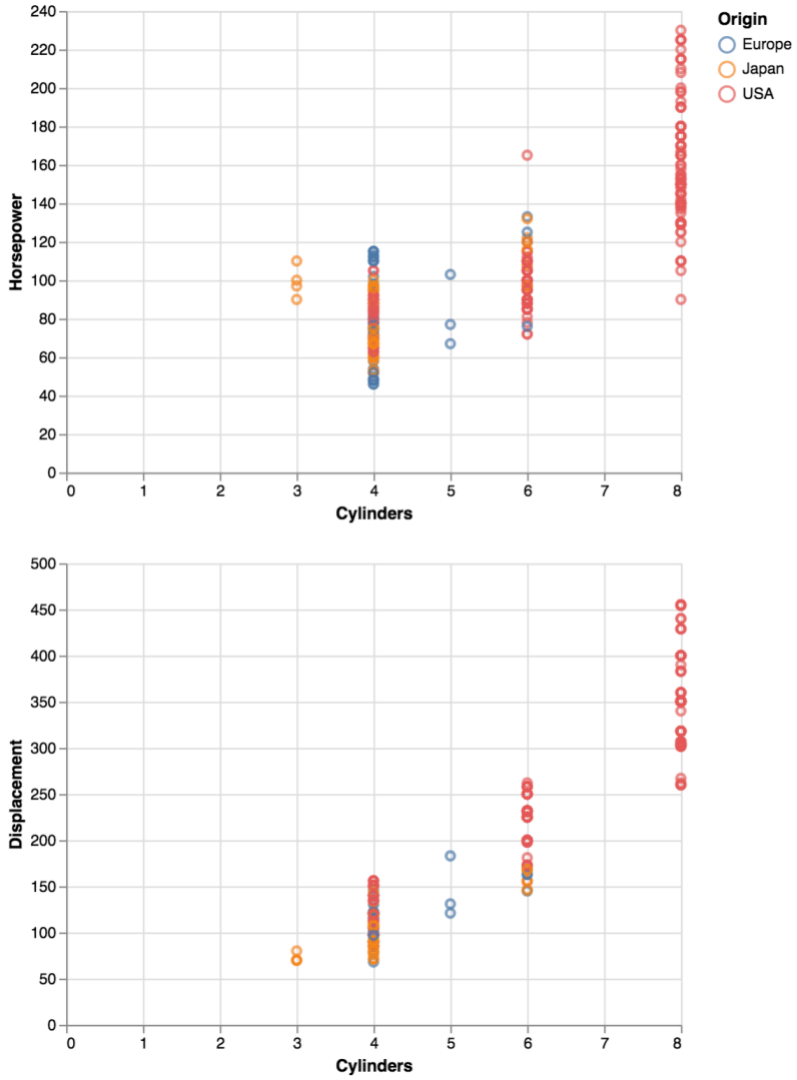
5.4 MULTIPLE CHARTS: SIMPLE COMBINATIONS

We already saw the '+' operator to plot two charts one in top of the other. There are a number of ways to combine charts, such as by using layers. In this section, we want to provide some more examples of simple chart combinations.

If we want to generate more than one chart, we can do it with the '&' operator. This will plot two charts one on top of the other. We can do it by naming the two different charts, or using the parenthesis to group the plotting method:

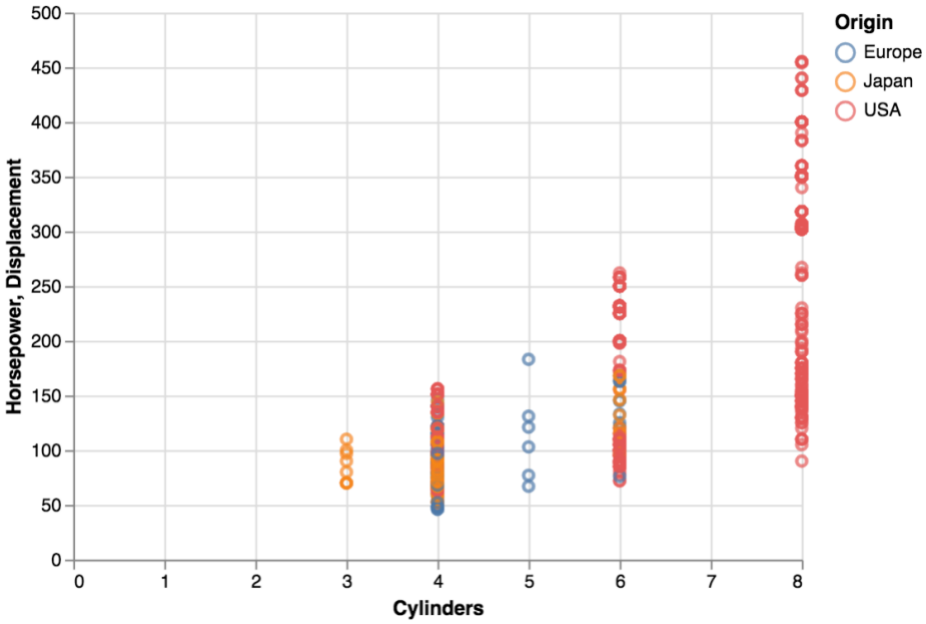
```
import altair as alt
import pandas as pd
from vega_datasets import data

df = data.cars()

(alt.Chart(df).mark_point(shape = 'circle'
    ).encode(
    x='Cylinders:Q',
    y='Horsepower:Q',
    color='Origin'
) &
alt.Chart(df).mark_point(shape = 'circle'
    ).encode(
    x='Cylinders:Q',
    y='Displacement:Q',
    color='Origin'
))
```

Which will generate:



Note that the legend is shared.

But we can also overlay one on top of the other, by substituting the '&' symbol for a '+', and the result would be:



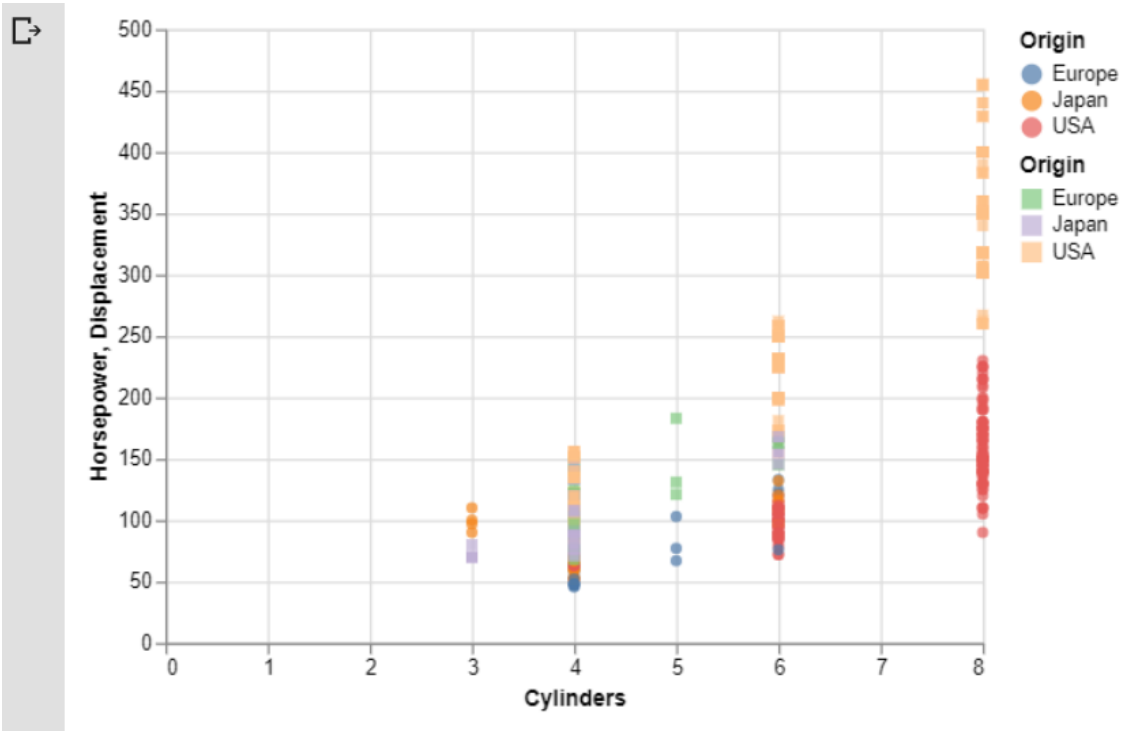
In this case, there is ambiguity since we are using color encodings of the data that are shared among both plots. If they are indicating different data, marks must be different, to avoid confusion.

It is possible to change the marks of one of the charts, to facilitate the separation. And it is also possible to change the color scheme of the charts, but since it is a property that it is by default shared among all the charts that are plot on top of each other, it is necessary to explicitly ask Altair to treat the colors independently. This can be achieved by using a combination of the *scale* property and *resolve_scale* method to make the colors independent.

In the following example of one of the charts takes squares as shape, and the other circles. Then, we set the color scheme of one of the charts as *accent* color scheme (you need to check the Vega documentation to see the available color schemes (https://vega.github.io/vega/docs/schemes/). Then, we ensure that both schemes are treated independently:

```
import altair as alt
import pandas as pd
from vega_datasets import data

df = data.cars()

first= alt.Chart(df).mark_circle(
    ).encode(
    x='Cylinders:Q',
    y=alt.Y('Horsepower:Q'),
    color = 'Origin:N'
    )
second = alt.Chart(df).mark_square(
    ).encode(
    x='Cylinders:Q',
    y=alt.Y('Displacement:Q'),
    color=alt.Color('Origin:N', scale=alt.Scale(scheme='accent'))
)

(first + second).resolve_scale(color='independent')
```

And the result is:

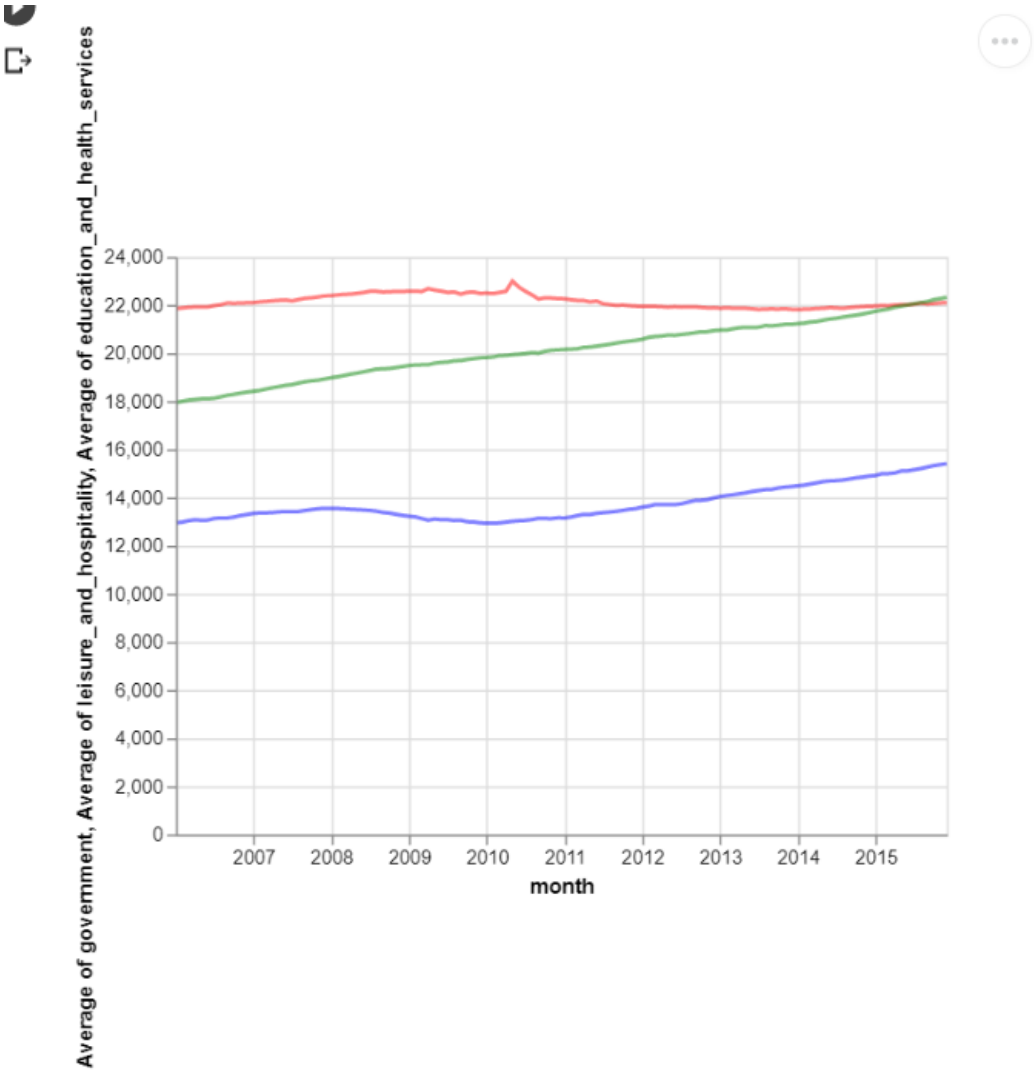


Layers can be used to visualize wide form data, such as the *stocks* dataset:

```
import altair as alt
import pandas as pd
from vega_datasets import data

df = data.us_employment()

base = alt.Chart(df).mark_line(
    color = 'red',
    strokeWidth = 2, strokeOpacity = 0.5
    ).encode(
    x='month:T',
    y='average(government):Q'
)

alt.layer(
    base,
    base.encode(y='average(leisure_and_hospitality):Q',
                color = alt.value('blue')),
    base.encode(y='average(education_and_health_services):Q',
                color = alt.value('green')))
```

Layers are the equivalent to the '+' operators on charts, but let you add more than two elements. In this case, since the data is wide form, we do not have a row per entry, and therefore, if we want to visualize all the columns, we would need to specify them one by one.

The result of the previous code is:

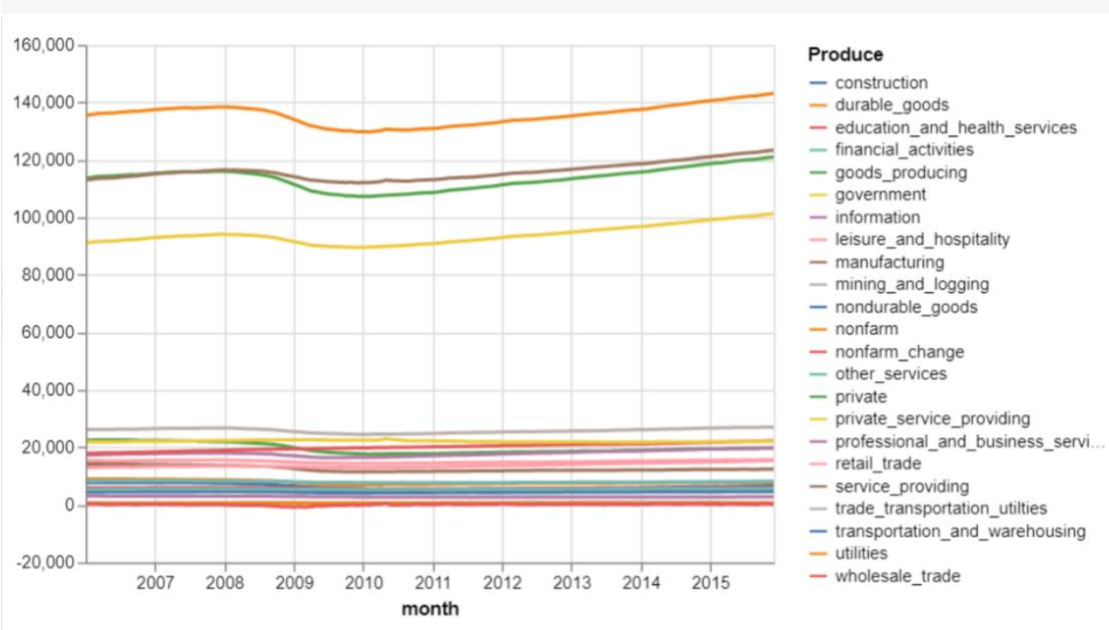


This is far less convenient than when data is in long form.

If we want to render the whole column set, we can melt the data using the function from pandas, and then render it as if it was long form:

```
import altair as alt
import pandas as pd
from vega_datasets import data

df = data.us_employment()
df2 = df.melt('month', var_name='Produce',
              value_name='amount')

alt.Chart(df2).mark_line().encode(
    x='month:T',
    y='amount:Q',
    color = 'Produce',
)
```

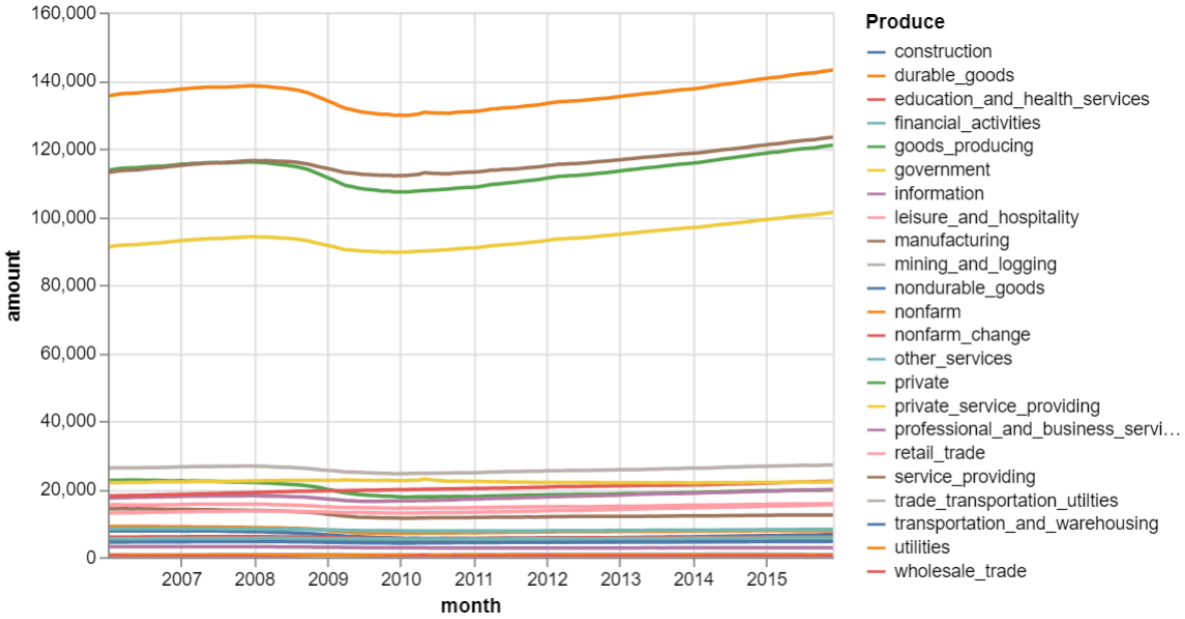
This would result in the following chart:



The previous chart is rendering negative values, which makes the Y axis span until the -20000 value. We can clip this by defining the domain of the axis using the *scale* property of the *y* parameter:

```
import altair as alt
import pandas as pd
from vega_datasets import data

df = data.us_employment()
df2 = df.melt('month', var_name='Produce',
              value_name='amount')

alt.Chart(df2).mark_line(clip=True).encode(
    x='month:T',
    y=alt.Y('amount:Q',scale = alt.Scale(domain=(0,150000))),
    color = 'Produce'
)
```

By adding the extra option of *clip* to *True*, the values negative values (that would be plotted in spite of the fact that the axis would start at 0, will be clipped. The result is shown in the following Figure.



EXERCISES

Exercise 1. Use the cars Vega dataset and plot a bar chart that counts the number of cars with each quantity of cylinders.

Exercise 2. Use the cars Vega dataset and plot a bar chart that shows the maximum displacement of the cars from each origin.

Exercise 3. Use the cars Vega dataset and plot a bar chart that shows the maximum displacement of the cars from each origin.

Exercise 4. Use the cars Vega dataset and plot chart that shows the average miles per gallon of the cars from each origin.

Exercise 5. Use the stocks Vega dataset and plot a bar chart that shows the average stock price for each company.

Exercise 6. Use the stocks Vega dataset and plot a bar chart that shows the average stock price for each company.

Exercise 6. Use the stocks Vega dataset and plot a chart that shows the average stock price for each company per year.

Exercise 7. With the cars Vega dataset, plot a histogram of the cars with different miles per gallon, stacked per origin, inverting the current sorting.

Exercise 8. Create a line plot that shows the price evolution of the yield barley in the different sites from the barley dataset.

Exercise 9. Create a bar chart with the maximum yield value of each variety using the barley dataset.

Exercise 10. Create a bar chart with the average yield value for each site using the barley dataset.

Exercise 11. Create a line chart using the cars dataset, with the average displacement of the cars per year, with different lines for each origin.

Exercise 12. Create a line chart using the cars dataset, with the average miles per gallon per year.

Exercise 13. Improve the previous chart by adding the confidence intervals of the miles per gallon. Plot both charts together.

Exercise 14. Create another line chart, also with the cars dataset, with the average horsepower per year, with confidence intervals, separated per origin, and render both of them separately.

Exercise 15. Render both charts on top of each other. Resolve the ambiguities.

Exercise 16. Render the first three columns of the employment dataset with three different colors.

Exercise 17. Add a rule on the average of the employment values for those data.

Exercise 18. Plot the monthly average of the all employment types of the employment dataset.

Exercise 19. Plot as a rule the average of the all employment types of the employment dataset.

Exercise 20. Use the dataset "Nivell academic de la població per sexe de la ciutat de Barcelona" of year 2018 from the BCN Open Data web, and show the number of people in the city of each academic degree.

Exercise 21. Show the number of people holding each academic degree ("Nivell academic") of the Horta-Guinardó neighborhood for a concrete month.

Exercise 22. Compare the higher academic degrees (high school and university) of each neighborhood in Barcelona for a certain year.

Exercise 23. Plot the evolution of the academic degree for men and women in Barcelona.

Exercise 24. Plot the evolution of the academic degree for men and women in a selected neighborhood in Barcelona.

Exercise 25. Show how the university degree has evolved in Barcelona.