pyG2

Release 1.0b1

Mar 31, 2023

Contents:

PyG2 is a python library that introduced **Grammar of Graphics** into Jupyter Notebooks. It is constructed wrapping javascript charting library G2 designed by AntV team.

With the use of PyG2 you can generate large number of charts without memorizing different functions for each and every function. PyG2 makes it more easier and logical to generate charts.

Leland Wilkinson's concept **grammar of graphics** simplifies graphic generations identifying the pipelined process of graphic generation. With grammar of graphics we can switch to a logical way of specifying graphics rather than using ad hoc specification patters.

CHAPTER 1

Create Your First Chart with pyG2

1.1 Installation

```
pip install pyG2
```

pyG2 only supports jupyter notebook. (Jupyterlab and Colab is not still supported.)

- Open jupyter notebook and create a new notebook file.
- Import G2 module:

```
from pyG2 import G2
```

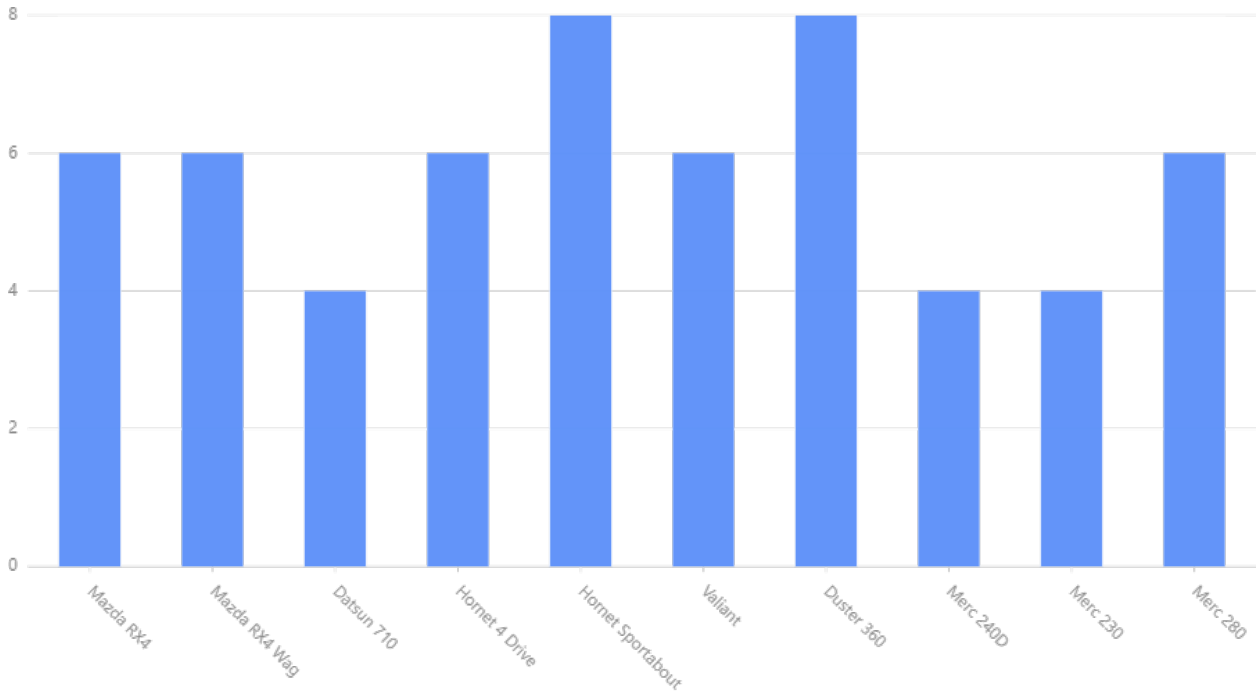
- Prepare the data set (Details of several cars from *mtcars* dataset):

```
data =[
{'name': 'Mazda RX4', 'mpg': 21.0, 'cyl': 6},
{'name': 'Mazda RX4 Wag', 'mpg': 21.0, 'cyl': 6},
{'name': 'Datsun 710', 'mpg': 22.8, 'cyl': 4},
{'name': 'Hornet 4 Drive', 'mpg': 21.4, 'cyl': 6},
{'name': 'Hornet Sportabout', 'mpg': 18.7, 'cyl': 8},
{'name': 'Valiant', 'mpg': 18.1, 'cyl': 6},
{'name': 'Duster 360', 'mpg': 14.3, 'cyl': 8},
{'name': 'Merc 240D', 'mpg': 24.4, 'cyl': 4},
{'name': 'Merc 230', 'mpg': 22.8, 'cyl': 4},
{'name': 'Merc 280', 'mpg': 19.2, 'cyl': 6}
]
```

- Create the chart (Number of cylinders of each car):

```
chart = G2.Chart(height=500,width=900)
chart.data(data)
chart.interval().position('name*cyl')
chart.render()
```

Output:



1.2 Explaination of code

`G2.Chart()` generates a chart object which holds details of the chart. Height and weight are optional. (Default is 500pt, 400pt).

`chart.data(data)` provides data to the chart. There are two supported data formats: list of dictionaries as in the example or pandas.DataFrame. (Latter is recommended.)

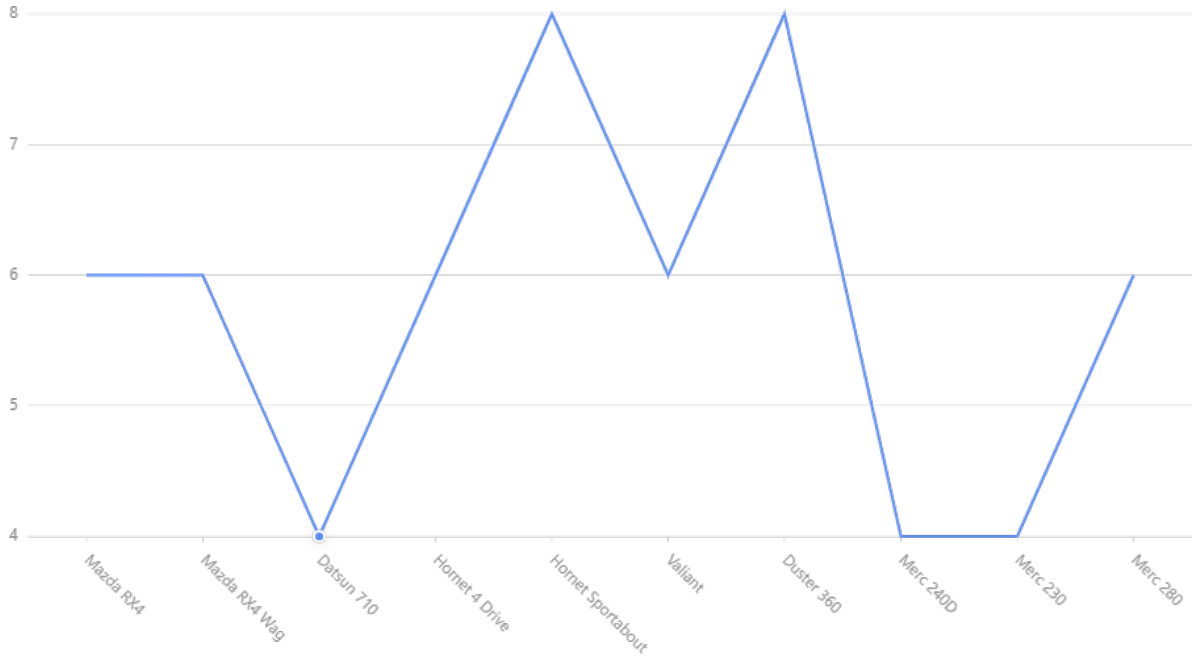
`chart.interval().position('name*cyl')` specifies geometry and aesthetics of the chart. `'name*cyl'` is the map that is represented in the chart. It is represented as a position. We choose interval geometry (bars) to visualize the chart.

`chart.render()` renders the chart to the notebook output.

1.3 Try Other Geometries

Use this code to generate a line chart:

```
chart = G2.Chart(height=500,width=900)
chart.data(data)
chart.line().position('name*cyl')
chart.render()
```



Output:

Try `area`, `point` plots. **Hint: Change `line` in above code to these words.**

1.4 Multiple plots

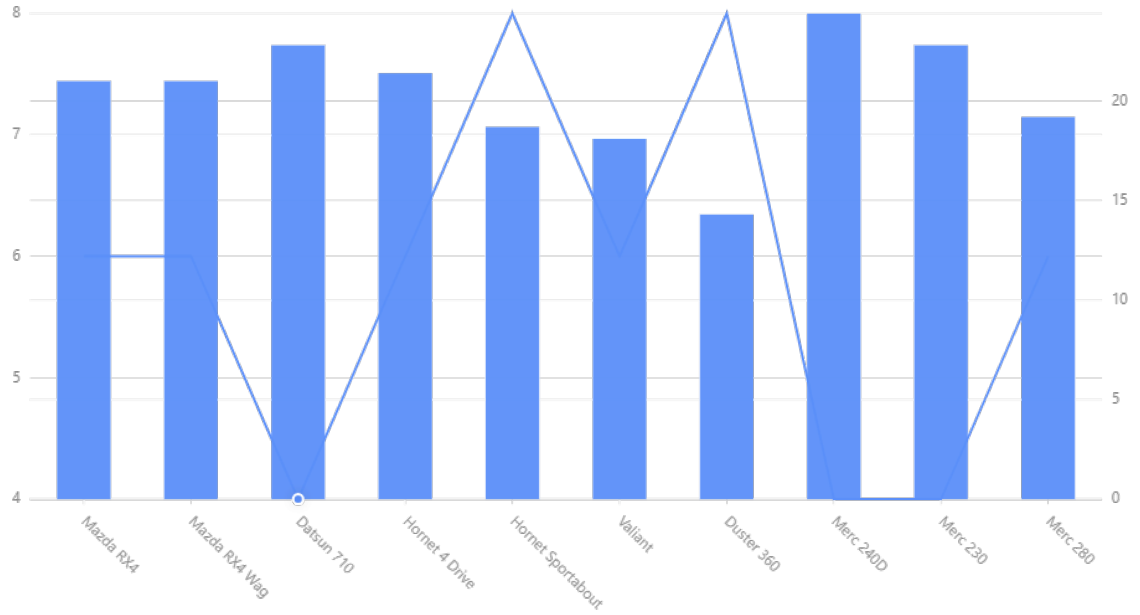
We shall create two graphs in one chart.

1. Name vs. Miles for Galon (mpg) : line
2. Name vs. No. of Cylinders (cyl) : interval (bar)

code:

```
chart = G2.Chart(height=500,width=900)
chart.data(data)
chart.line().position('name*cyl')
chart.interval().position('name*mpg')
chart.render()
```

Output:



Read our tutorials to learn how to generate various types of charts

Special Note:

If graphs are not visible when the notebook reloads, restart the kernal and run the code again.

CHAPTER 2

Grammar of Graphics

2.1 Three Steps of Grammar of Graphics

Charting graphic generation is a systematic process. There are three steps in graphic generation.

2.1.1 Specification

Describe how the graph should be created. We have to specify six elements of the chart.

2.1.2 Implementation

Construct related objects and communication between them. You do not have to worry about this. This is what chart library does. When your code runs set of objects are constructed and the graphic or the chart is created through the communication of these objects.

2.1.3 Rendering

Show graphics to the user.

PyG2 supports two rendering methods

1. **Canvas rendering**: Graphic is rendered as in a raster format.
2. **SVG rendering**: Graphic is rendered in a vector format.

Raster format stores a picture as a matrix of pixels (like painting) while vector formats stores details on mathematical formulaes that generate the parts of the picture (like drawing the picture).

Vector formats are scalable without blurring. However when the number of elements in the graphic is very higher it will take more time to render the image. You can read more about it in the internet.

- To specify canvas rendering (default):

```
chart = G2.Chart(renderer='canvas')
```

- To specify svg rendering:

```
chart = G2.Chart(renderer='svg')
```

2.2 Six Specification statements

1. Data
2. Transformations
3. Scale
4. Coordinate
5. Element
6. Guide

2.2.1 Data

In pyG2, data should be specified as a list of dictionaries or as a pandas.DataFrame object.

2.2.2 Transformations

Data transformations should be done beforehand you provide them to the Chart object. You can use pandas.DataFrame to obtain these transformations.

2.2.3 Scales

`Chart.scale(variable, **config)` is used to specify the scale and scale transformations.

2.2.4 Coordinate

`Chart.coordinate(variable, **config)` is used to specify the coordinate system and coordinate transformations.
Default is Cartesian coordinates.

2.2.5 Element

In pyG2 chart elements are named geometry. They are the main components of the graphic: line in line chart, bar in bar chart etc.

2.2.6 Guides

In G2, guides are legends, tooltips and annotations. They are the auxillary components of a chart that helps the user to understand the chart easily.

2.3 Graphic Generation Pipeline

Leland Wilkinson identified that there is a pipelined process in graphic generation.

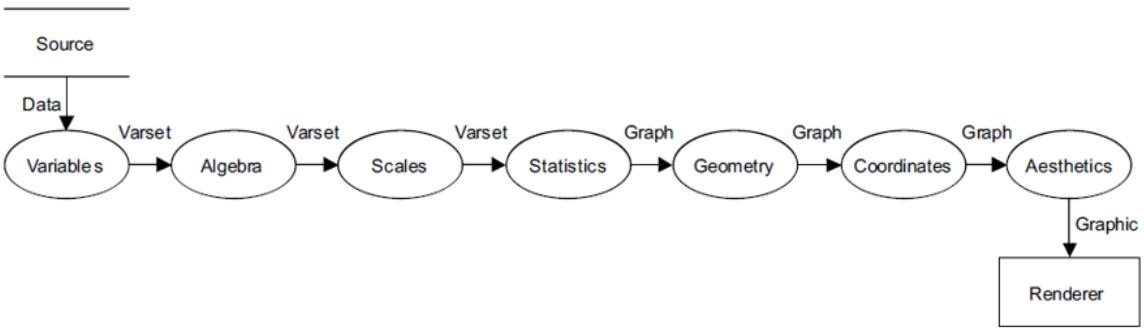


Fig. 1: Fair use: taken from **The Grammar of Graphics** of Leland Wilkinson

2.3.1 Source

The data source

2.3.2 Variable

Variable represents a property of an object.

2.3.3 Algebra

Varset algebra: there are operations between varsets. (e.g. cross, nest, blend)

This concept is not properly adopted into G2. Therefore it is not included in pyG2 also. map of variables is closely related to cross operation. e.g `name*cyl`

2.3.4 Scales

Scales are mapping values of the variables to dimensions of a coordinate system.

2.3.5 Statistics

More oftenly what we want to visualize in the graphic is differs from raw data points. (Remember linear regression) Therefore statistics is to alter positions of data points.

2.3.6 Geometry

Geometry is to map data with unbounded dimensions into bounded graph dimensions.

2.3.7 Coordinate

Coordinates are coordinates of the graph space.

2.3.8 Aesthetics

Aesthetics are the observable physical properties of the graphics.

2.3.9 Renderer

Renderer generates the graphic

2.4 Variable vs. Varset

To define a variable we need three things.

1. Set of objects
2. Set of values
3. Function mapping objects to values

To define a varset we need,

1. Set of values
2. Set of objects
3. Function mapping values to objects

What is the difference? In varsets value has more priority than objects. In a graph basically we represent values. An object may be a certain data point. Coordinates are values. Therefore graphs are made of varsets.

2.5 Graph vs. Graphic

- Graph is a mathematical representaion of a bounded multidimentional space. A graph is an abstract entity.
- Graphic is to represent a graph in observable physical properties (aesthetics) such as position, color, shape.

2.6 Explanation of Pipeline

Pipeline is the process of generating any graphic. First we have to get data from a source. Then we have to create variables and turn them into varsets. Then using varset algebra we create complex varsets. Thereafter you have to map variable into a dimension. (Dimension gives a direction or order to the values.) Then you have to alter positions of the data points to find the points that are to be visualized in the graphic. Upto here we worked with arbitrary dimension. Then we should specify coordinate system related to the graphic space. Then we can choose aesthetic to visualize the graph. Finally we render the graph through a rendering mechanism.

We cannot by-pass these steps. This is the way we naturally generate a graphic visualization.

Wilikinson suggested to go through this pipeline and to provide functions to work on relevent steps of the pipeline instead of giving ad hoc functions for certain chart types.

Instead of providing list of specific usages, this architecture provides **a grammar** to generate any graphic as you wish.

CHAPTER 3

Tutorial

PyG2 is very logical and commnds are simple when it comes to graphic generations.

- Installation:

```
pip install pyG2
```

- Open Jupyter Notebook
- Import modules:

```
from pyG2 import G2
import pandas as pd
```

- In this example we use `mtcars` dataset from https://vincentarelbundock.github.io/Rdatasets/csv/datasets/mtcars.csv Download CSV file, save it in the same folder as the notebook and change the header of the first column to **name**
- Prepare a Panda DataFrame:

```
df = pd.read_csv('mtcars.csv')
```

Columns are 'name', 'mpg', 'cyl', 'disp', 'hp', 'drat', 'wt', 'qsec', 'vs', 'am', 'gear' and 'carb'.

3.1 Data preperation

Data should be processed by panda dataframe if any transformation presents.

3.2 Chart Layout Configuration

We have to construct a graph object:

```
chart = G2.Chart(height, width, autoFit, limitInPlot, padding, pixelRatio, renderer,␣
↪visible)
```

All parameters are optional.

height: int (ppt) (default 500)

width: int (ppt) (default 400)

autoFit: 'true' / 'false' (default 'true') **'true' (str) not True (bool)**

limitInPlot: 'true' / 'false' (default 'false')

padding: int/ [int,int,int,int]

pixelRatio: int (pixelRatio for canvas rendering)

render: 'canvas'/'svg'

visible: 'true'/'false'

- Display changes:

```
chart.render()
```

All other codes comes between Chart object construction and rendering steps.

3.3 Variable Maps

One dimension map: e.g. `name` Two dimensional variable map: e.g `name*cyl`

3.4 Main Topics

3.4.1 Scales

format:

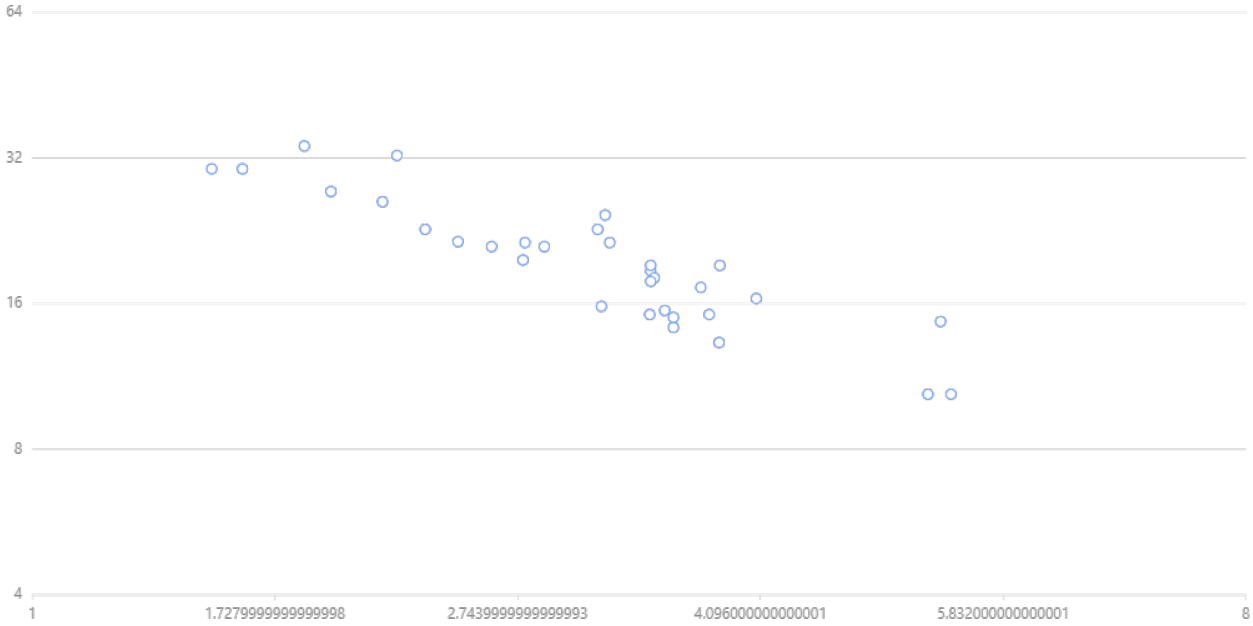
```
chart.scale(variable,type,min,max,values,range,tickCount,formatter,tickMethod,alias,
↪nice)
```

variables are optional.

e.g.

```
chart = G2.Chart(height=500, width=1000)
chart.data(df)
chart.point().position('wt*mpg')
chart.scale('wt',type='pow',exponent=3)
chart.scale('mpg',type = 'log', base=2, min=4)
chart.render()
```

Output:



There are several types. These type requires additional **optional** arguments.

types:

cat: classification metric

timeCat: time classification metrics

linear: linear metric

time: continuous time measurement

log: log metrics

pow: pow measure

quantize: segmentation metric, user can specify uneven segmentation

quantile: equal measure, automatically calculate the segment according to the distribution of data

identity: constant measure

There are suitable defaults according to the nature of data if you do not specify manually.

alias: string , name of the axis

values: domain

range: [min,max]

formatter: a javascript function definition as a string

tickCount: number of ticks

tickMethod: a javascript function definition as a string to calculate ticks.

nice: 'true'/'false' nice numbers

additional:

base: base for type `log`

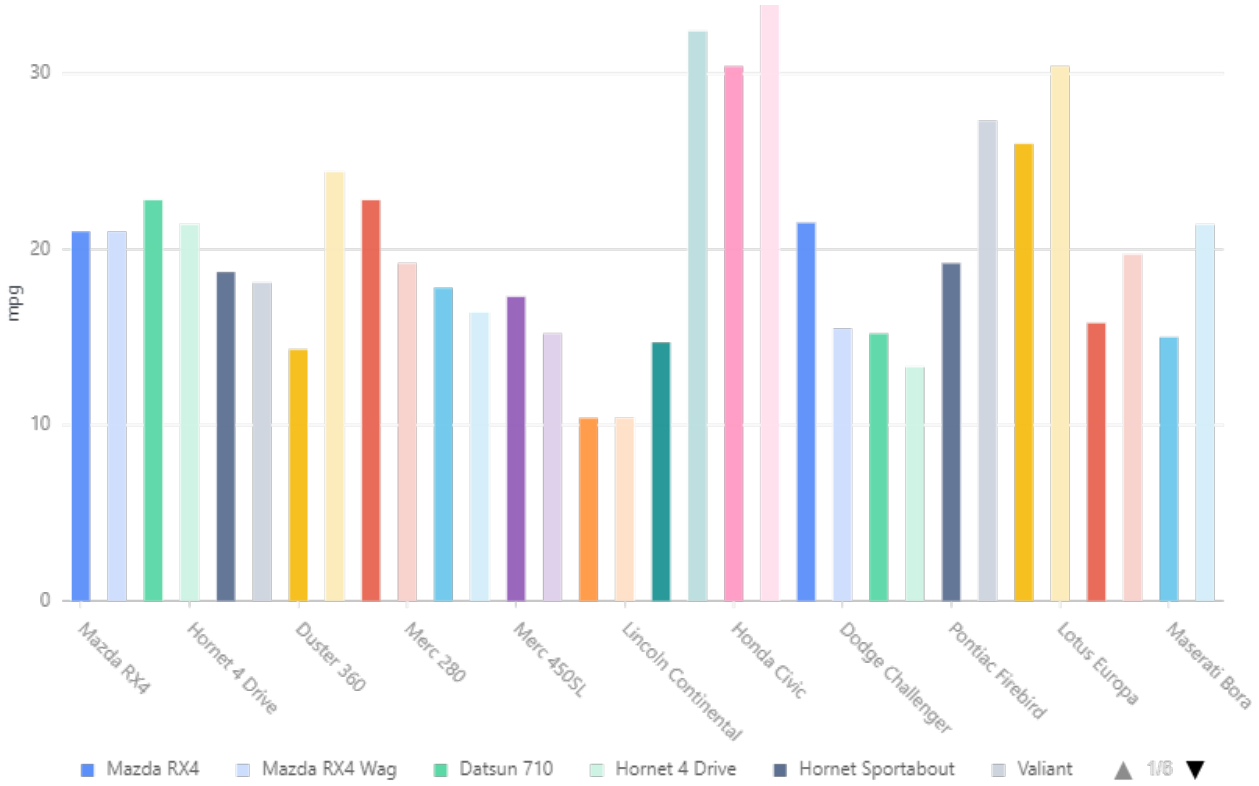
exponent: exponent for typ `pow`

ticks: list of values for ticks in `quantize`

3.4.2 Axis Configurations

Add axis title:

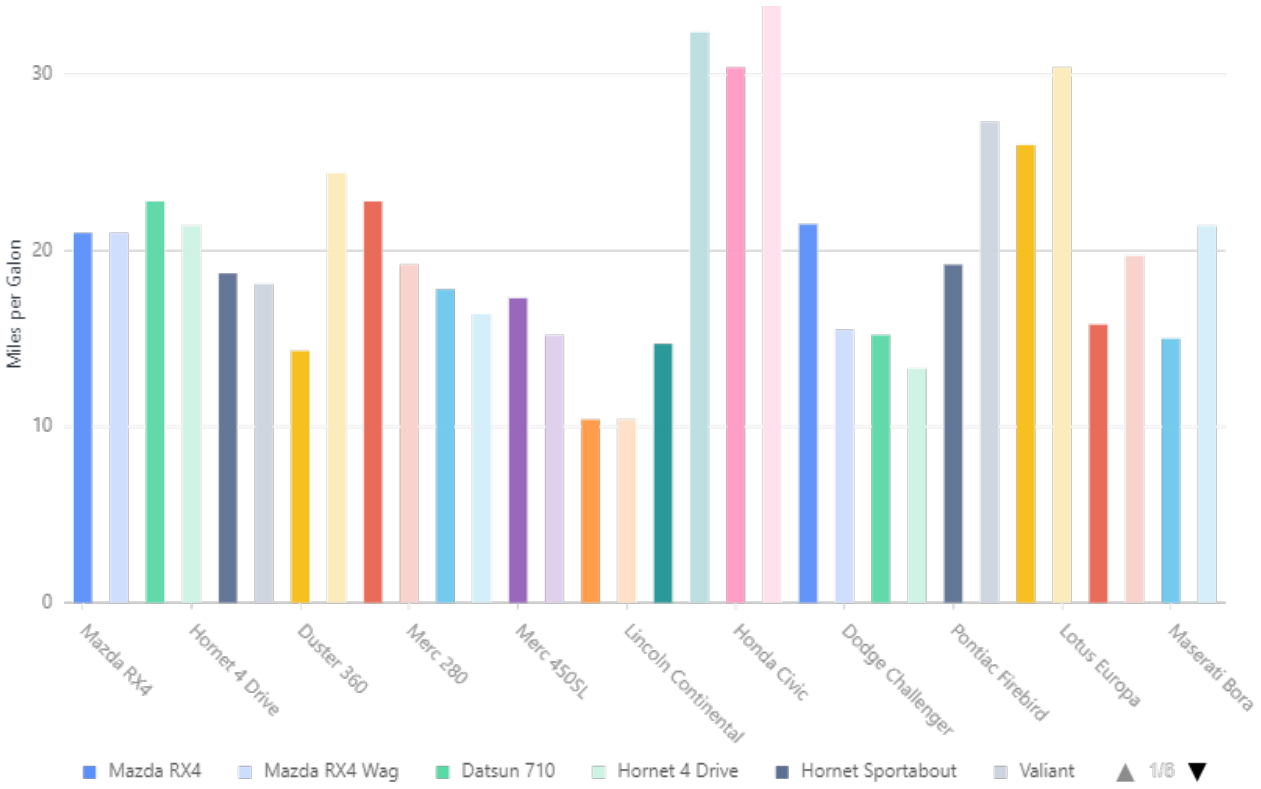
```
chart = G2.Chart(height=500, width=800)
chart.data(df)
chart.axis('mpg',title={})
chart.interval().position('name*mpg').color('name')
chart.render()
```



If you want to change axis name you have to add:

```
chart.scale('mpg',alias = 'Miles per Galon')
chart.render()
```

Output:



3.4.3 Geometry and Aesthetic

Geometry

Geometries are the main components of the chart. Geometries maps the variables into graphic space.

G2 supports:

point, line, area, interval, polygon, edge, schema geometry types.

point

```
chart.point().position('x*y')
```

This code generates points at the places specified by position aesthetic as (x,y).

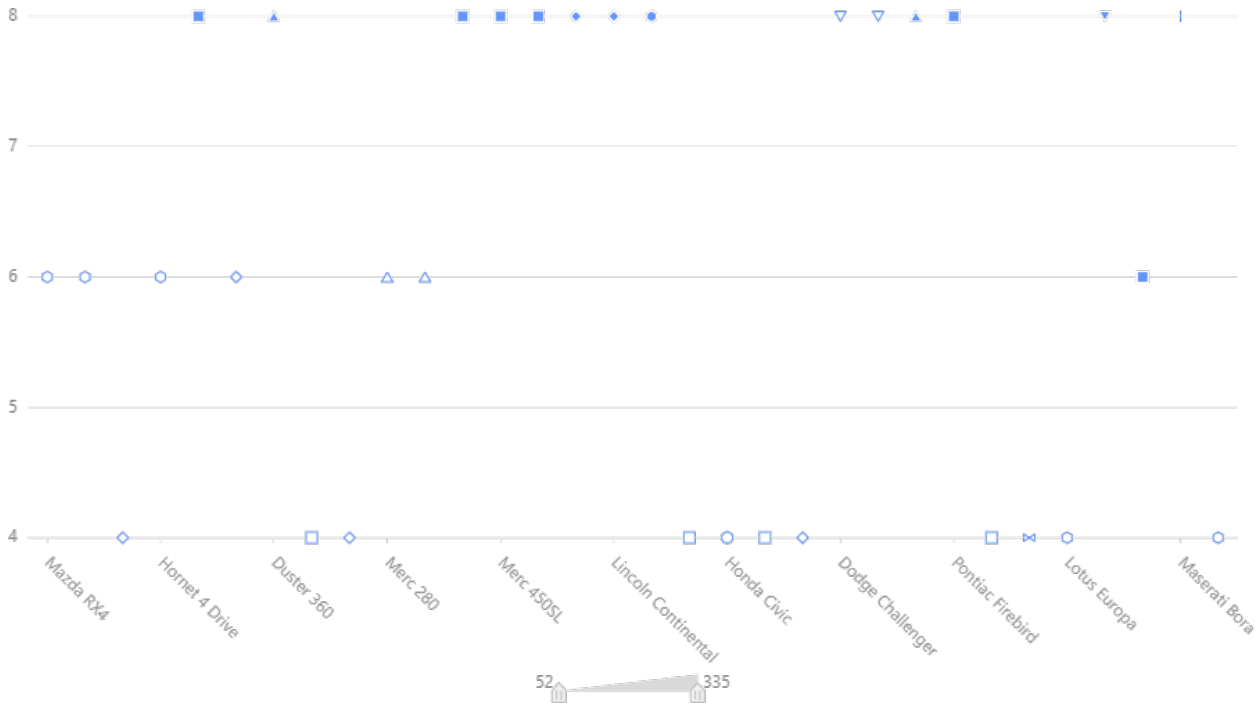
Supported shapes are (default is `hollow-circle`),

'circle', 'square', 'bowtie', 'diamond', 'hexagon', 'triangle', 'triangle-down', 'hollow-circle', 'hollow-square', 'hollow-bowtie', 'hollow-diamond', 'hollow-hexagon', 'hollow-triangle', 'hollow-triangle-down', 'cross', 'tick', 'plus', 'hyphen', 'line'

e.g. Different shapes

```
data = pd.read_csv("mtcars.csv")
chart = G2.Chart(width = 900)
chart.data(data)
chart.point().position('name*cyl').shape('hp')
chart.render()
```

Output:



line

```
chart.line().position('x*y')
```

This code generates a line chart that is drawn connecting all points specified by position aesthetic. If shape or color aesthetics are used then there will be several lines related to each aesthetic.

Supported shapes are

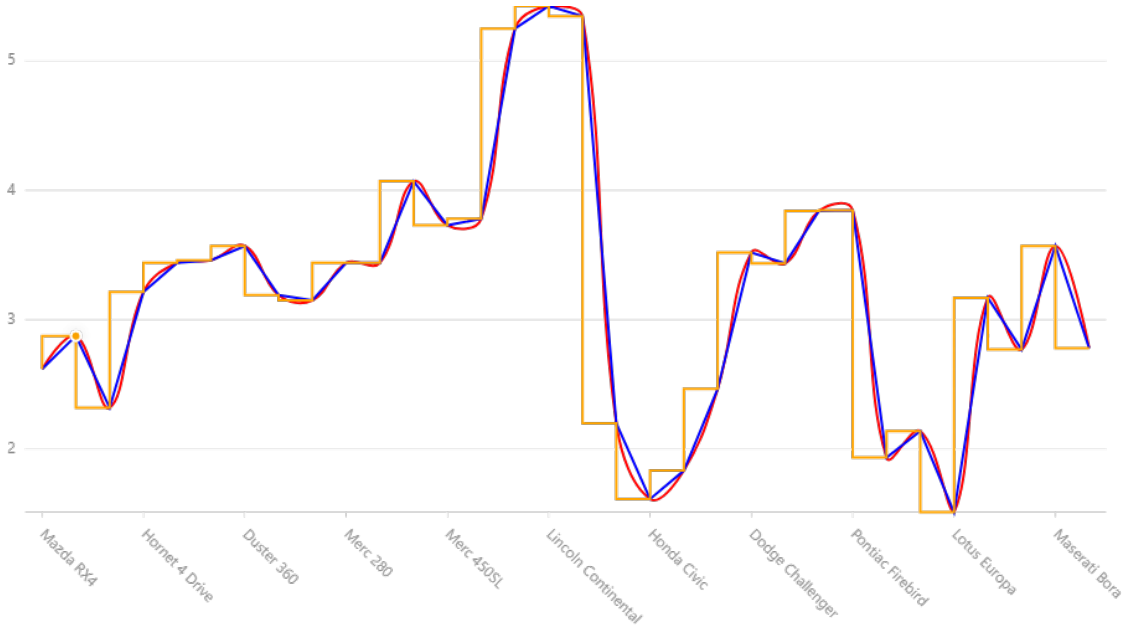
'line',' dot', 'dash',' smooth', 'hv',' vh', 'hvh',' vhv'

('hv',' vh', 'hvh',' vhv' are used for ladder diagrams: h horizontal, v vertical)

e.g. Some line shapes comparison:

```
data = pd.read_csv("mtcars.csv")
chart = G2.Chart(width = 900)
chart.data(data)
chart.line().position('name*wt').shape('smooth').color('red')
chart.line().position('name*wt').shape('line').color('blue')
chart.line().position('name*wt').shape('vh').color('orange')
chart.render()
```

Output:



area

```
chart.area().position('x*y')
```

This code generates area chart. Here under the field y there may be a single value or a list of two values

If y is a single value then the area is the area under the line drawn by connecting (x,y) points. If y is a list then the area is between two y values is the area.

Shape and line aeasthetics generates seperate area for each set. Supported shapes are

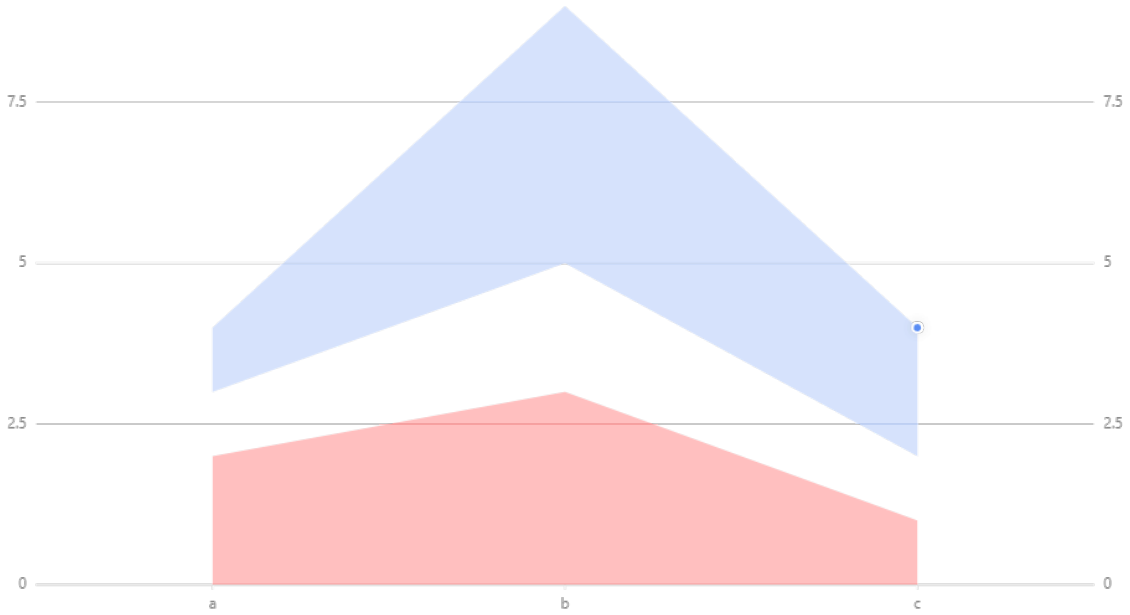
‘area’, ‘smooth’, ‘line’, ‘smooth-line’

(when line and smooth-line is used the area is not filled - only the border.)

e.g.

```
data = [{'x':'a','y':[3,4],'z':2},{'x':'b','y':[5,9],'z':3},{'x':'c','y':[2,4],'z':1}]
chart = G2.Chart(width = 900)
chart.data(data)
chart.area().position('x*y')
chart.area().position('x*z').color('red')
chart.scale('z', min=0,max=9).scale('y',min=0,max=9)
chart.render()
```

Output:



N.B:

If we use lists in lines and points then there will be several lines and points for each record.

interval

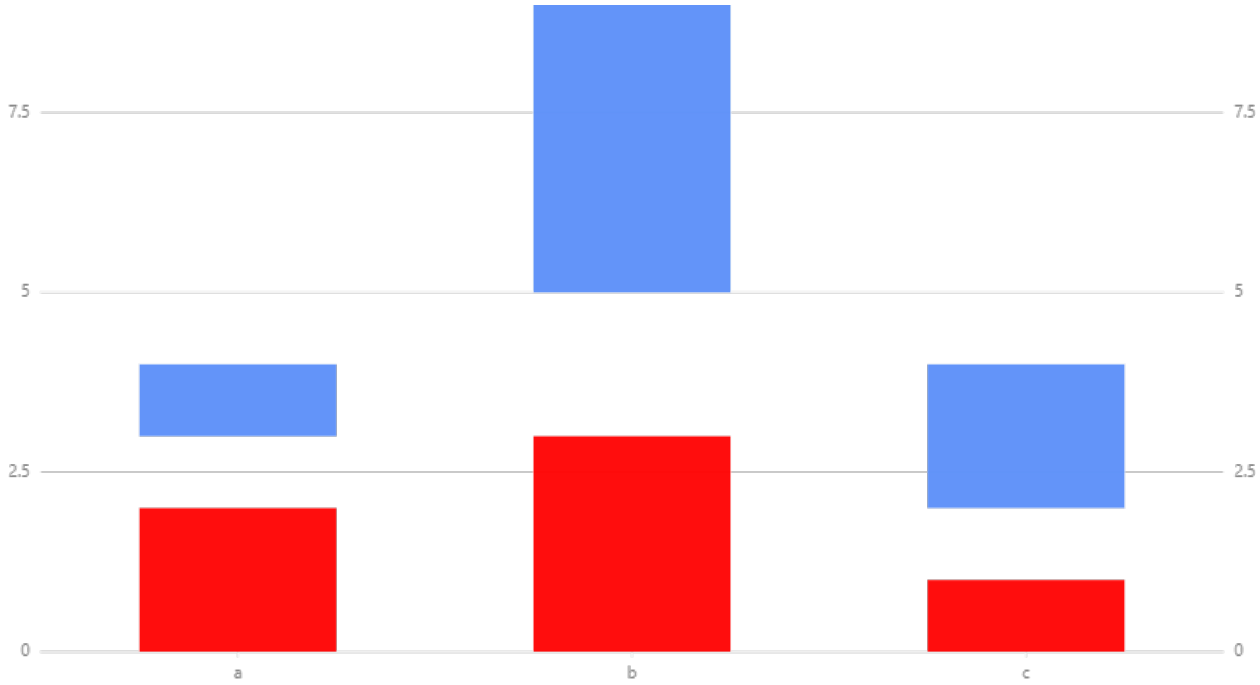
```
chart.interval().position('x*y')
```

If y is a single value then the interval is the interval between y value and 0. If y is a list then the interval is between two y values is the interval. If x is a single value the is of a default fixed width bar at (x,y) If x is a list then the interval is a bar spreaded between first and second value of the list.

e.g

```
data = [{'x':'a','y':[3,4],'z':2},{'x':'b','y':[5,9],'z':3},{'x':'c','y':[2,4],'z':1}]
chart = G2.Chart(width = 900)
chart.data(data)
chart.interval().position('x*y')
chart.interval().position('x*z').color('red')
chart.scale('z', min=0,max=9).scale('y',min=0,max=9)
chart.render()
```

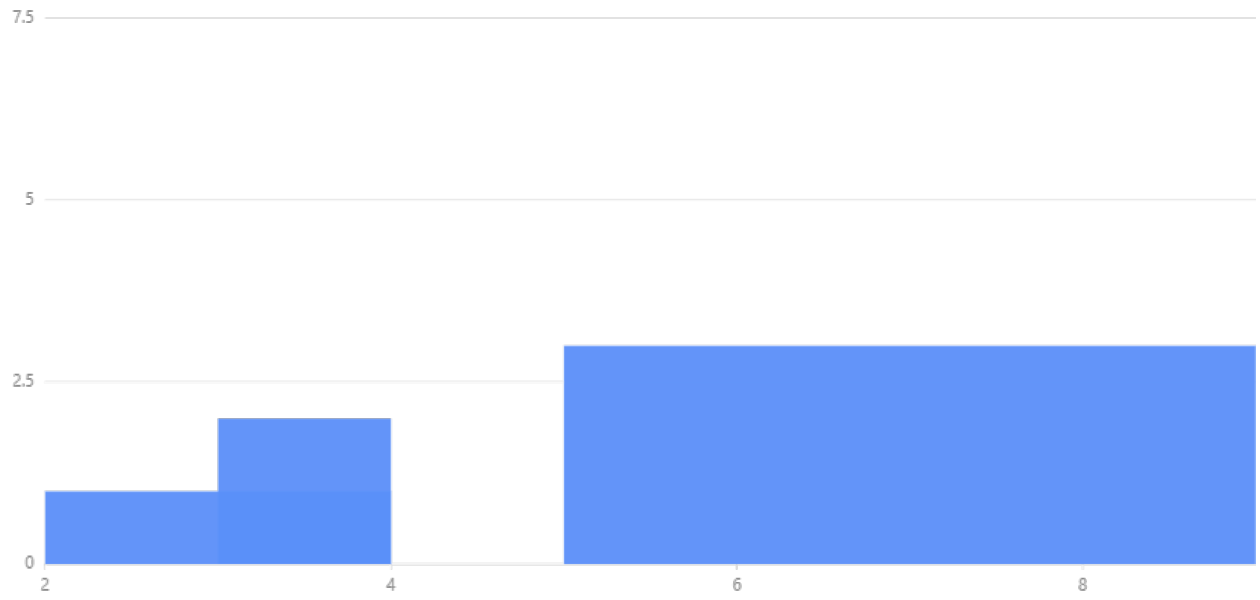
Output:



Another example:

```
data = [{'x':'a','y':[3,4],'z':2},{'x':'b','y':[5,9],'z':3},{'x':'c','y':[2,4],'z':1}]
chart = G2.Chart(width = 900)
chart.data(data)
chart.interval().position('y*z')
chart.scale('z', min=0,max=9)
chart.render()
```

Output:



Supported shapes are

'rect', 'hollow-rect', 'line', 'tick', 'funnel', 'pyramid'

'funnel' is for funnel chart. 'pyramid' is for pyramid chart.

polygon

```
chart.polygon().position('x*y')
```

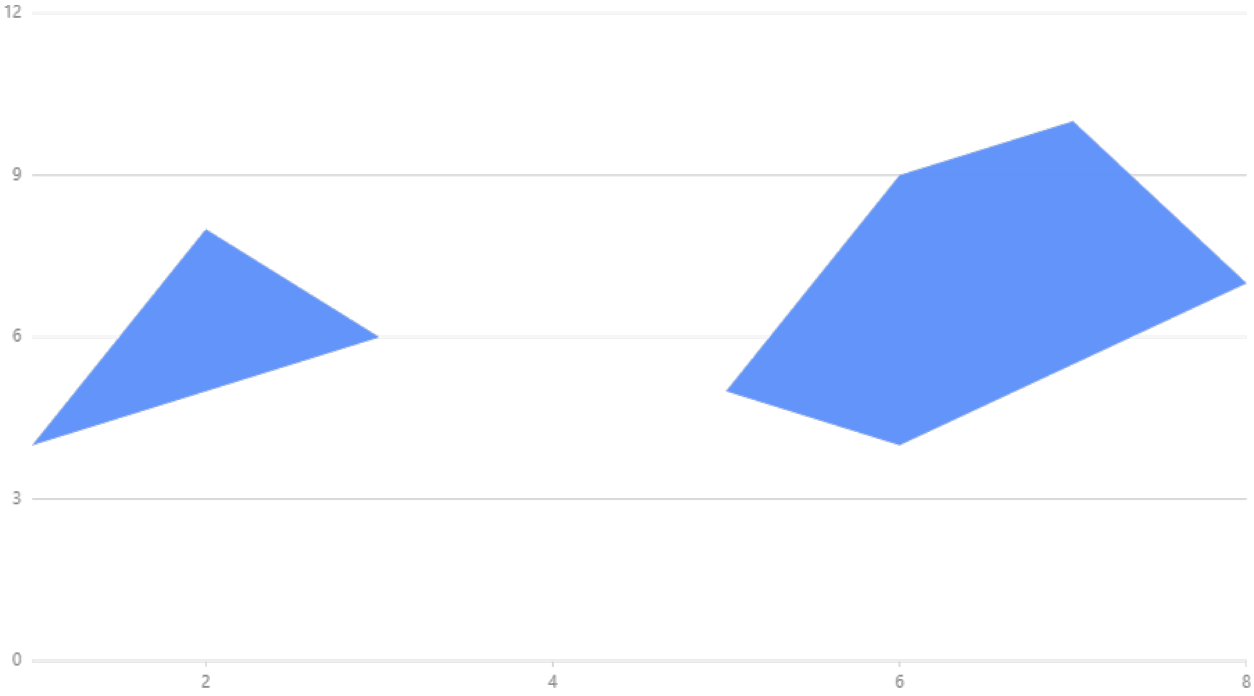
x and y are lists such that corresponding values of two lists gives coordinates of the vertices. (Vertices are connected as they are ordered in two lists.)

Shape and size aesthetics are not related to polygons.

e.g.

```
data = [{'x':[1,2,3],'y':[4,8,6]},{'x':[5,6,7,8,6],'y':[5,9,10,7,4]}]
chart = G2.Chart(width = 900)
chart.data(data)
chart.polygon().position('x*y')
chart.scale('y',min=0,max=12)
chart.render()
```

Output:



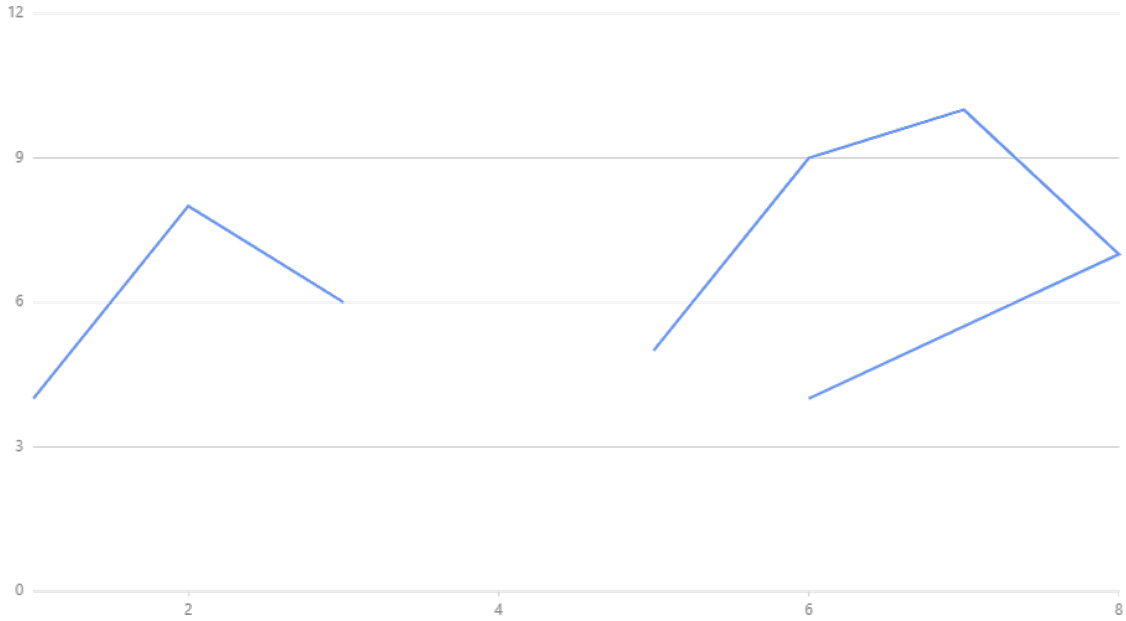
edge

Edge connects the points specified in two lists for x coordinates and y coordinates.

e.g.

```
data = [{'x':[1,2,3],'y':[4,8,6]},{'x':[5,6,7,8,6],'y':[5,9,10,7,4]}]
chart = G2.Chart(width = 900)
chart.data(data)
chart.edge().position('x*y')
chart.scale('y',min=0,max=12)
chart.render()
```

Output:



schema

```
chart.schema().position('x*y')
```

If the map is one dimensional then list must be supplied.

When the map is two dimensional,

x is categorical variable and y is an array of values.

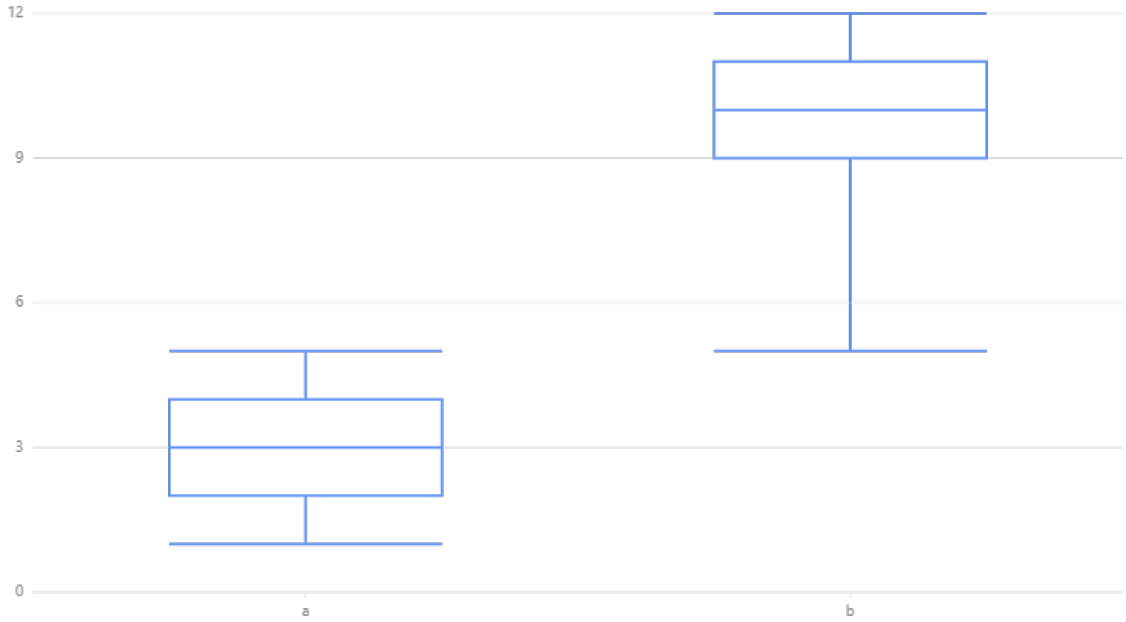
Supported shapes:

'box'

e.g.

```
data = [{'x':'a','y':[1,2,3,4,5]},{'x':'b','y':[5,9,10,11,12]}]
chart = G2.Chart(width = 900)
chart.data(data)
chart.schema().position('x*y').shape('box')
chart.scale('y',min=0)
chart.render()
```

Output:



Aesthetics

position

```
chart.<geometry>().position('x*y')
```

This part is completed under the previous section.

shape

```
chart.<geometry>().position('x*y').shape('z')
```

If we put a specific shape to z that given under geometries the geometry will take that shape. If we give a variable name to z the shape will change with the value of that variable.

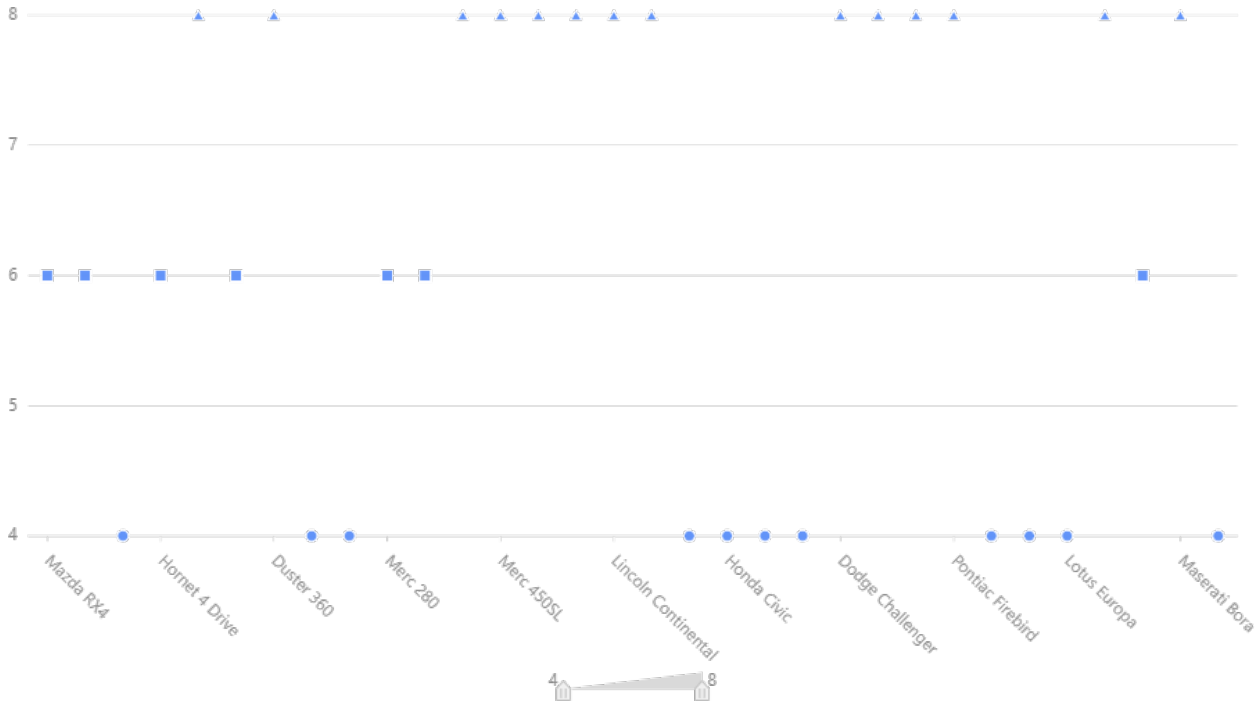
As below we can specify shapes we want to use as an optional argument

```
chart.<geometry>().position('x*y').shape('z',*values)
```

e.g.

```
data = pd.read_csv("mtcars.csv")
chart = G2.Chart(width = 900)
chart.data(data)
chart.point().position('name*cyl').shape('cyl','circle','square','triangle')
chart.render()
```

Output:

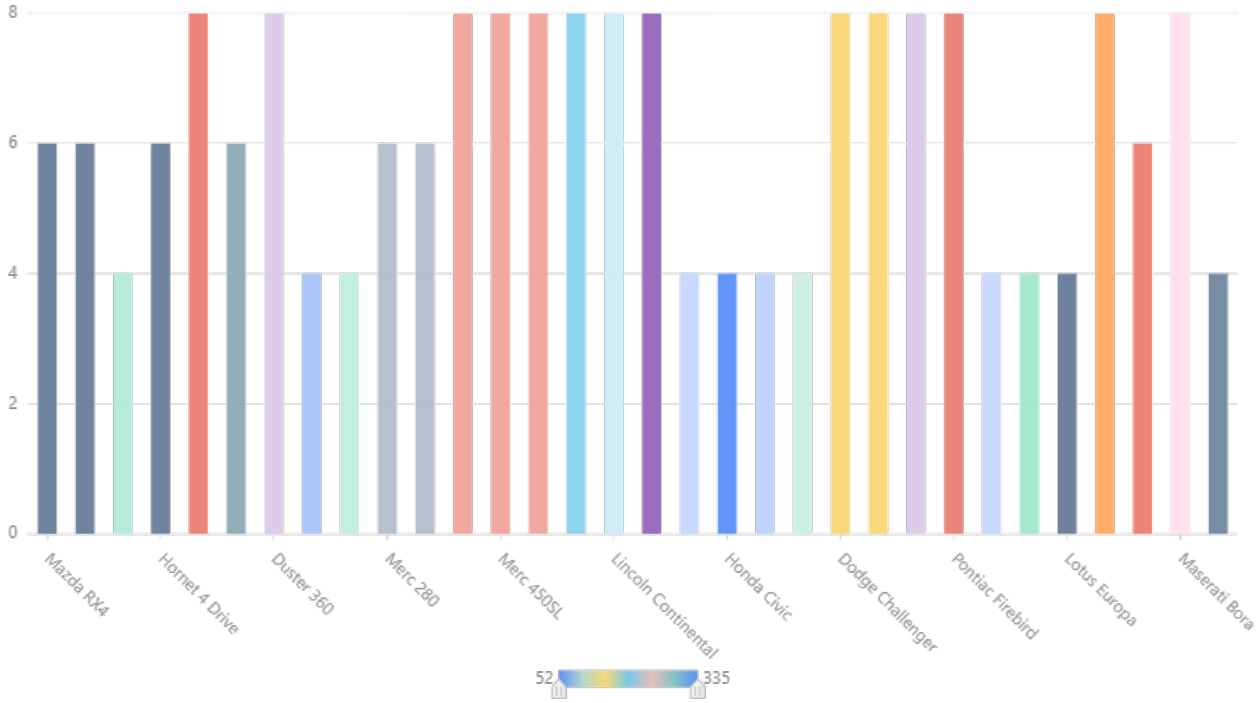


color

```
chart.<geometry>().position('x*y').color('z')
```

e.g.

```
data = pd.read_csv("mtcars.csv")
chart = G2.Chart(width = 900)
chart.data(data)
chart.interval().position('name*cyl').color('hp')
chart.render()
```

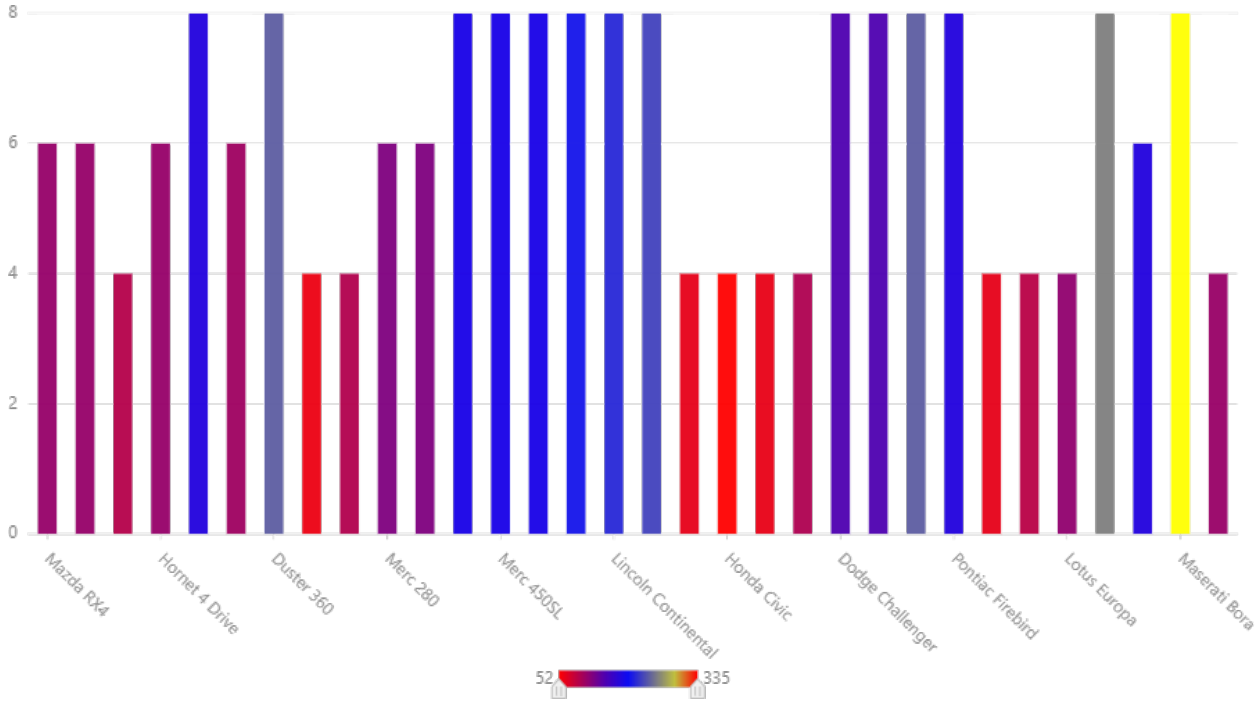


or

```
chart.<geometry>().position('x*y').color('z',*values)
```

e.g.:

```
data = pd.read_csv("mtcars.csv")
chart = G2.Chart(width = 900)
chart.data(data)
chart.interval().position('name*cyl').color('hp','red','blue','yellow')
chart.render()
```



The colour range is prepared by interpolation. Therefore if you put 'white' and 'black' you can get a greyscale chart.

```
chart.<geometry>().position('x*y').color(color)
```

Here `values` are list of colors to be used and `color` is a color if one color is used for complete geometry.

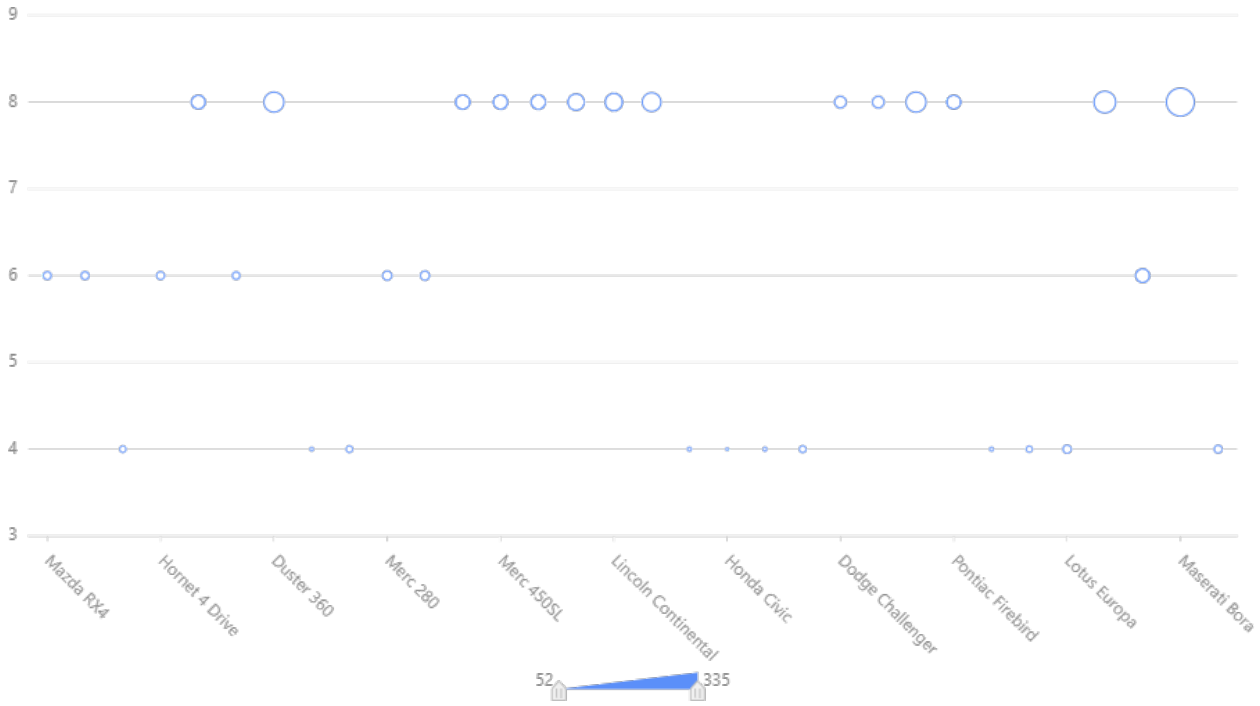
size

```
chart.<geometry>().position('x*y').size('z')
```

or

```
chart.<geometry>().position('x,y').size(value)
```

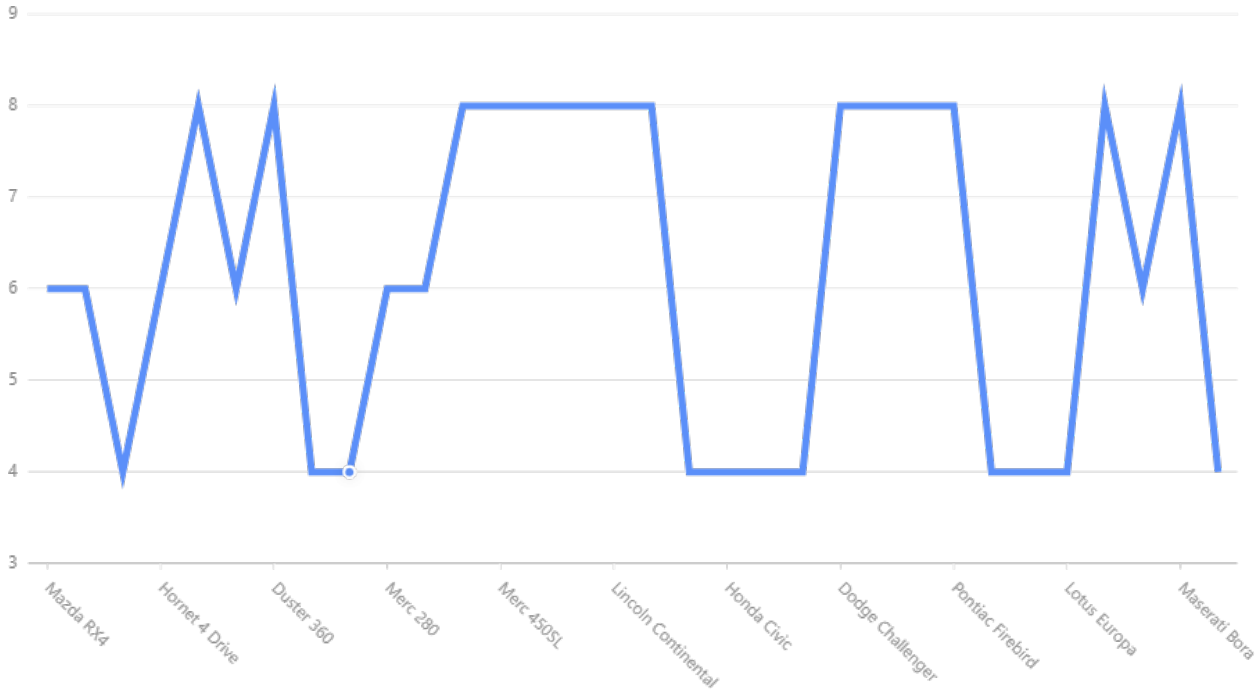
The size can be changed according to a field or you can specify a definite value to the size.



Another example:

```
data = pd.read_csv("mtcars.csv")
chart = G2.Chart(width = 900)
chart.data(data)
chart.scale('cyl', min=3,max=9)
chart.line().position('name*cyl').size(5)
chart.render()
```

Output:



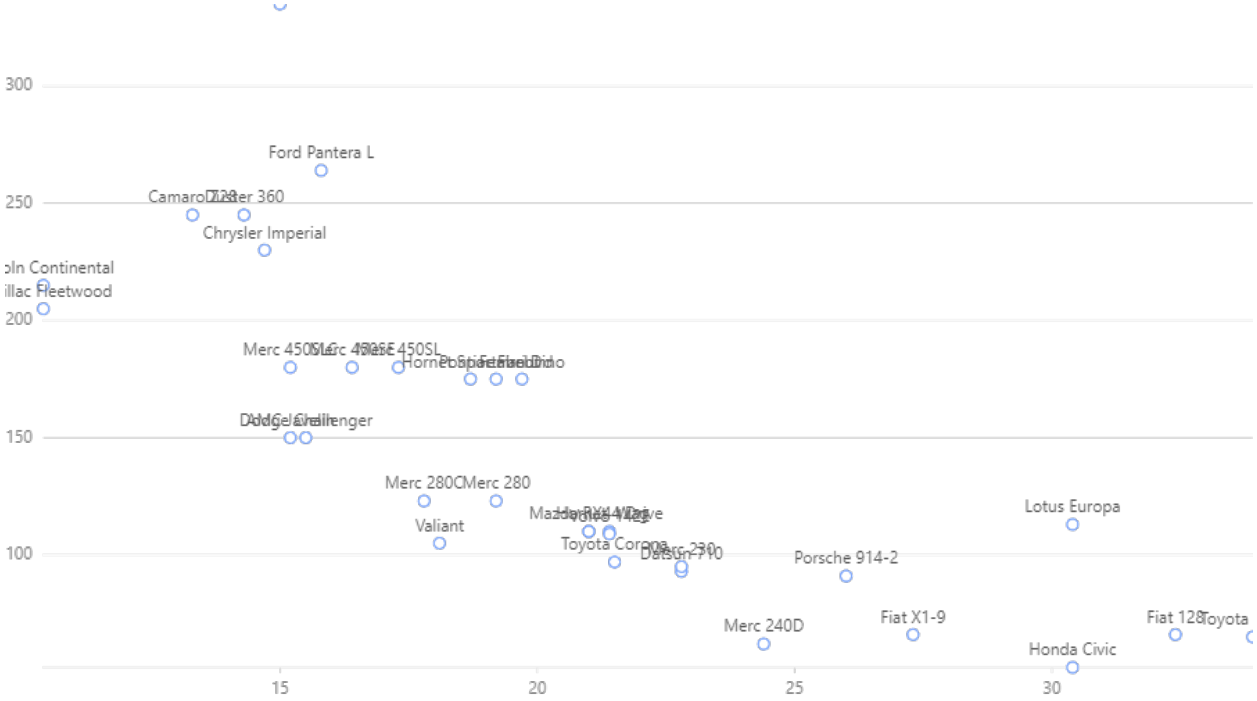
Label

```
chart.<geometry>().position('x*y').label('z')
```

e.g.:

```
data = pd.read_csv("mtcars.csv")
chart = G2.Chart(width = 900)
chart.data(data)
chart.point().position('mpg*hp').label('name')
chart.render()
```

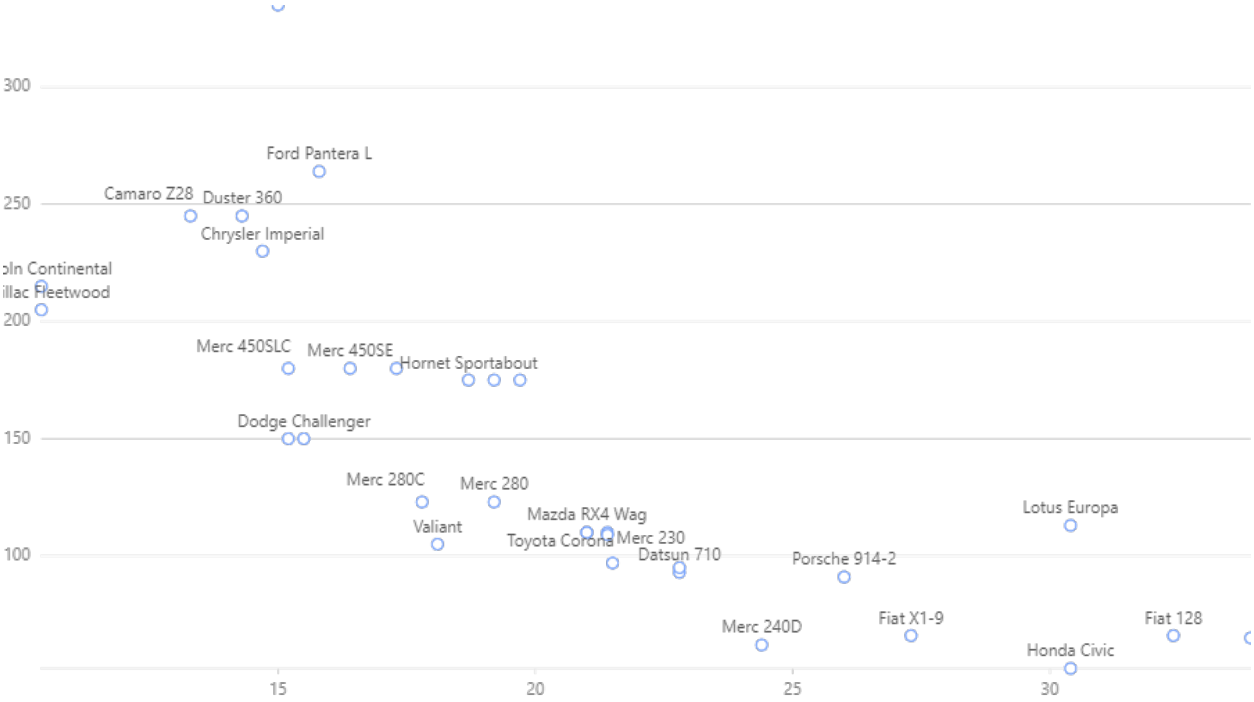
Output:



You can see this label is little messy this can be handled this way:

```
data = pd.read_csv("mtcars.csv")
chart = G2.Chart(width = 900)
chart.data(data)
chart.point().position('mpg*hp').label('name',layout={"type":'fixed-overlap'})
chart.render()
```

Output:



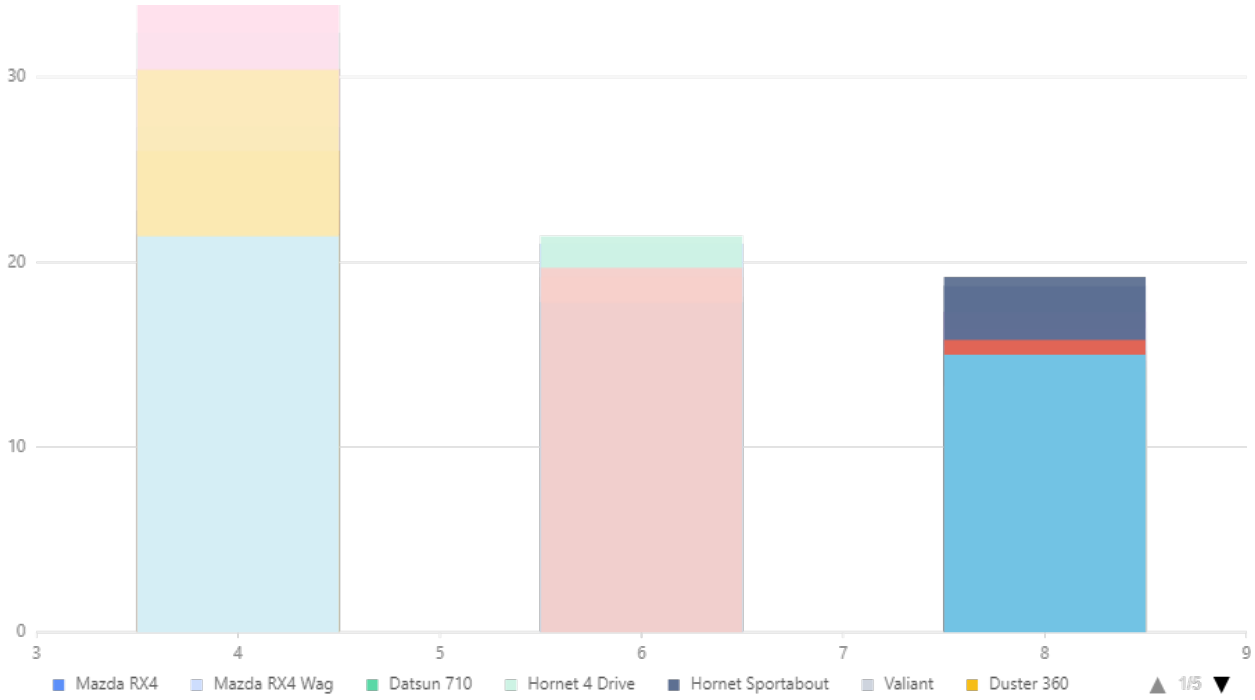
`fixed-overlap` remove some of the labels to make the graph clear. Use `overlap` to reposition colliding labels without removing any.

Collision Handelling

Sometimes several parts of the graphs collides, that is to say locate in the overlapping positions. We can fix this by `adjust()`

There are several `adjust` types. `stack` to make colliding parts in a stack. `dodge` to present them horizontally packed. `jitter` in point maps to show them as a cluster.

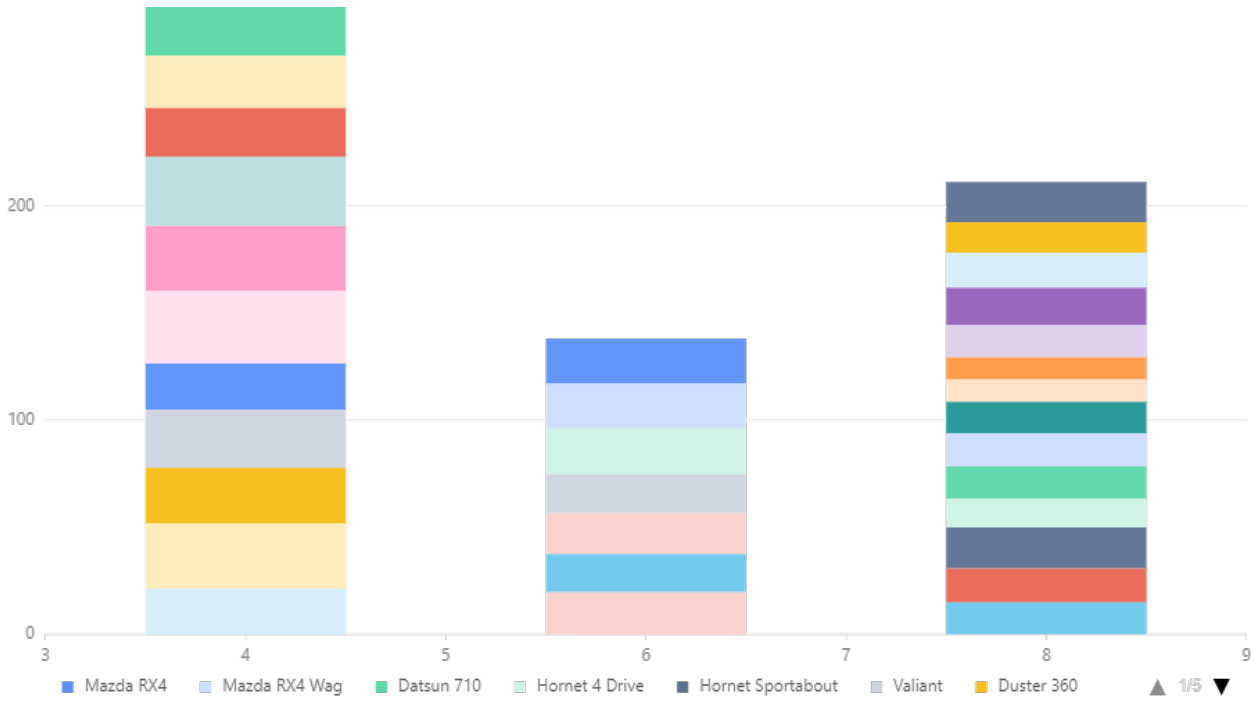
e.g. chart with Collision



e.g. stacked chart:

```
data = pd.read_csv("mtcars.csv")
chart = G2.Chart(width = 900)
chart.data(data)
chart.interval().position('cyl*mpg').color('name').adjust('stack')
chart.render()
```

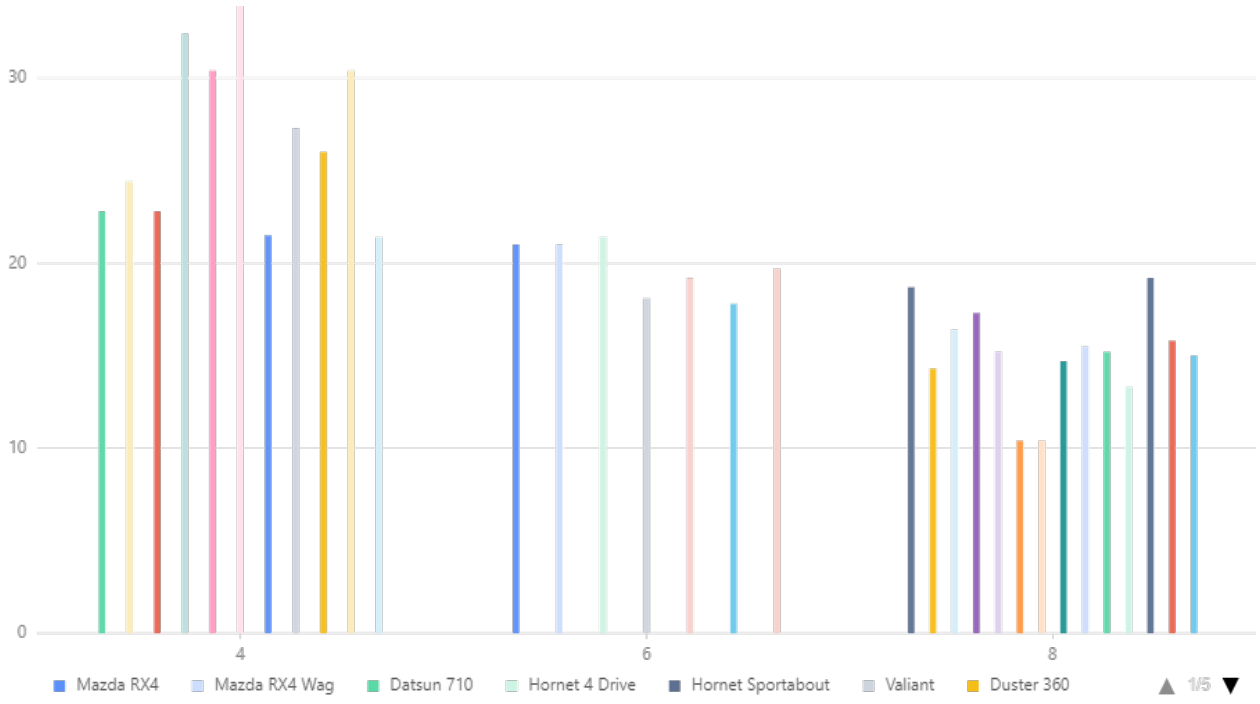
Output:



e.g. dodged chart:

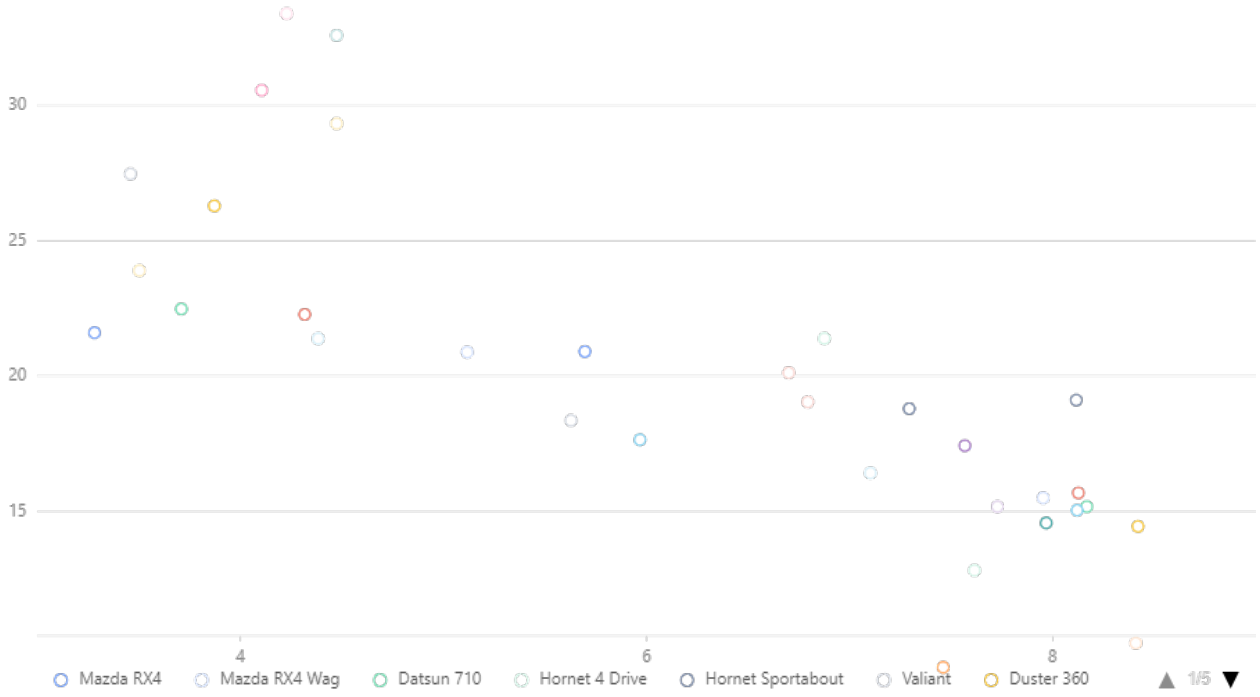
```
data = pd.read_csv("mtcars.csv")
chart = G2.Chart(width = 900)
chart.data(data)
chart.scale('cyl',type='cat', values =[4,6,8])
chart.interval().position('cyl*mpg').color('name').adjust('dodge')
chart.render()
```

Output:



e.g. jitter charts:

```
data = pd.read_csv("mtcars.csv")
chart = G2.Chart(width = 900)
chart.data(data)
chart.scale('cyl',type='cat', values =[4,6,8])
chart.point().position('cyl*mpg').color('name').adjust('jitter')
chart.render()
```



3.4.4 Coordinates

We can specify coordinates as below:

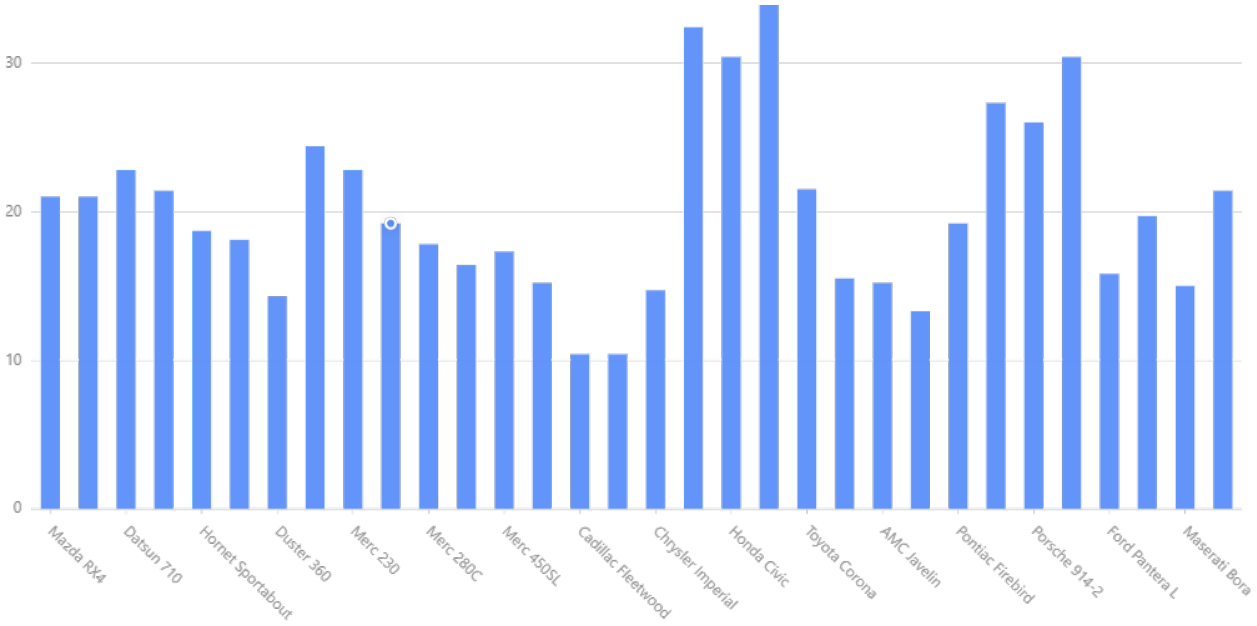
```
chart.coordinate('type')
```

There are four types.

1. `rect`/ `cartesian` (x,y):

   ```
   chart = G2.Chart(height=500, width=1000)
   chart.data(df)
   chart.interval().position('name*mpg')
   chart.coordinate('rect')
   chart.render()
   ```

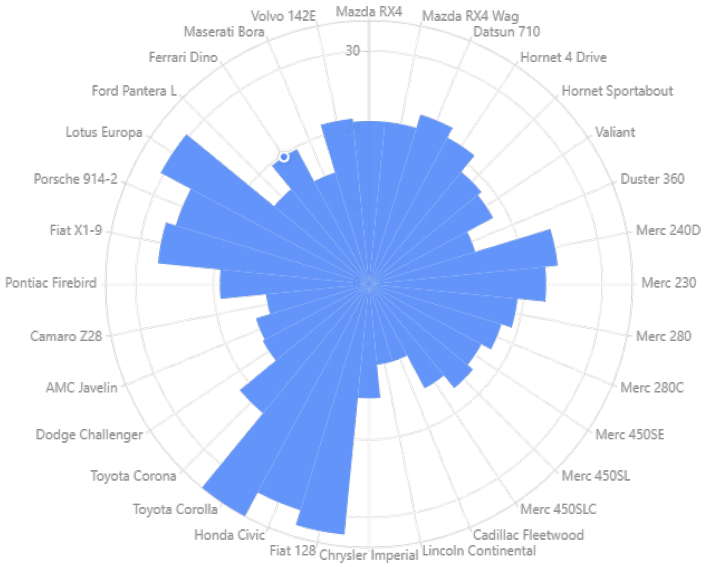
Output:



1. `polar` (r,theta):

```
chart = G2.Chart(height=500, width=1000)
chart.data(df)
chart.interval().position('name*mpg')
chart.coordinate('polar')
chart.render()
```

Output:



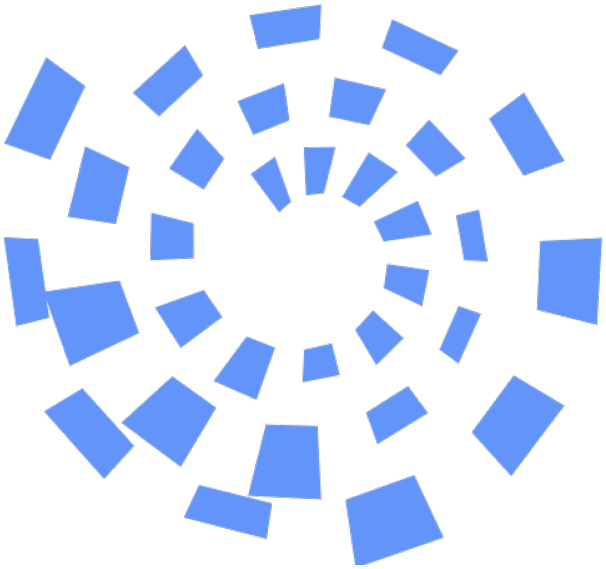
1. `theta` (theta,r):

```
chart = G2.Chart(height=500, width=1000)
chart.data(df)
chart.interval().position('name*mpg')
chart.coordinate('theta')
chart.render()
```

1. `helix`

```
chart = G2.Chart(height=500, width=1000)
chart.data(df)
chart.interval().position('name*mpg')
chart.coordinate('helix')
chart.render()
```

Output:

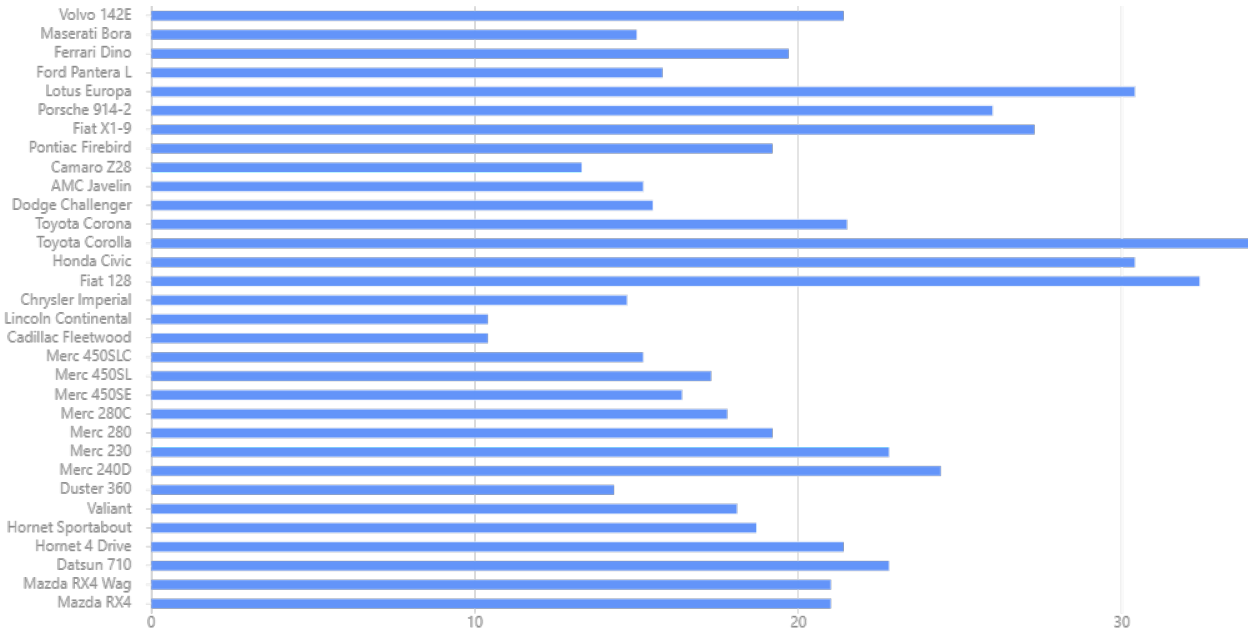


There are coordinate transformation functions.

1. transpose

```
chart = G2.Chart(height=500, width=1000)
chart.data(df)
chart.interval().position('name*mpg')
chart.coordinate('rect').transpose()
chart.render()
```

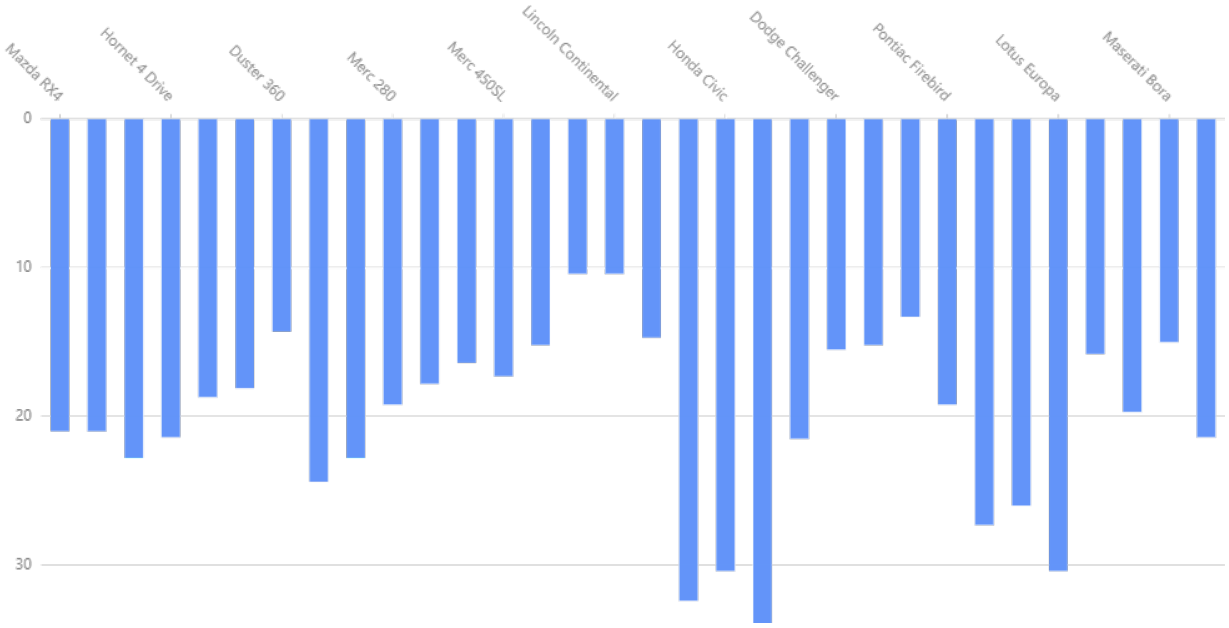
Output:



1. reflect

```
chart = G2.Chart(height=500, width=1000)
chart.data(df)
chart.interval().position('name*mpg')
chart.coordinate('rect').reflect('y')
chart.render()
```

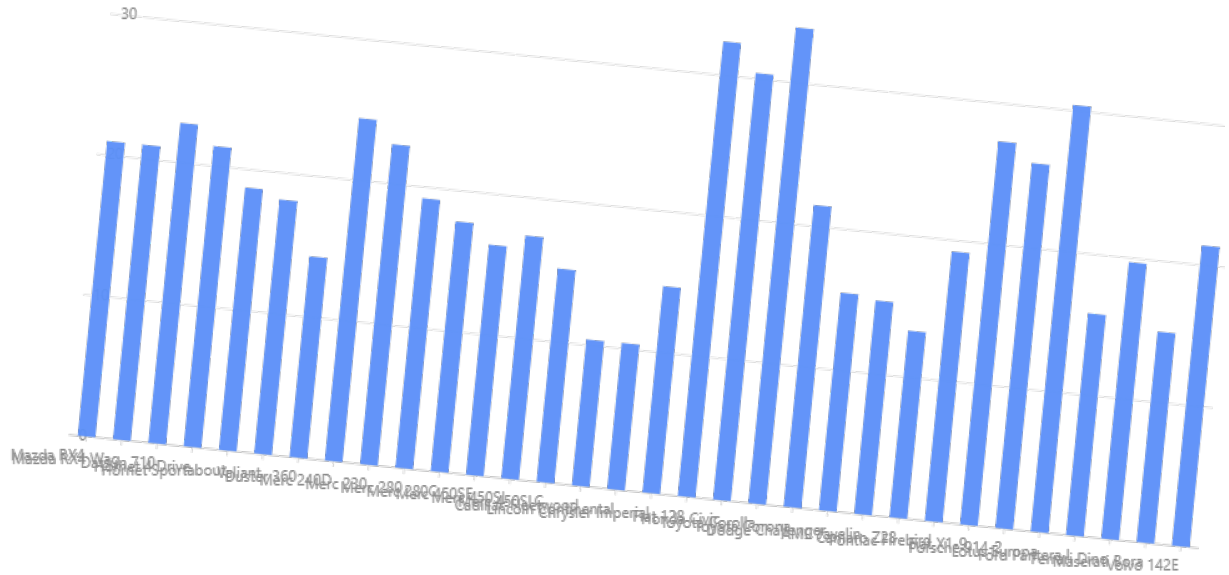
Output:



1. rotate:

```
chart = G2.Chart(height=500, width=1000)
chart.data(df)
chart.interval().position('name*mpg')
chart.coordinate('rect').rotate(0.1)
chart.render()
```

Output:



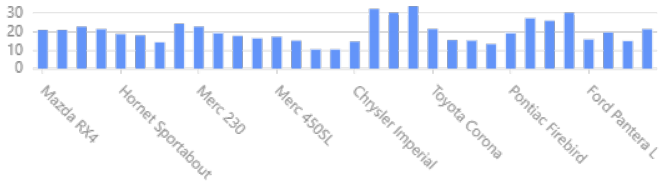
1. scale:

```
chart = G2.Chart(height=500, width=1000)
chart.data(df)
```

(continues on next page)

(continued from previous page)

```
chart.interval().position('name*mpg')
chart.coordinate('rect').scale(0.5,0.1)
chart.render()
```

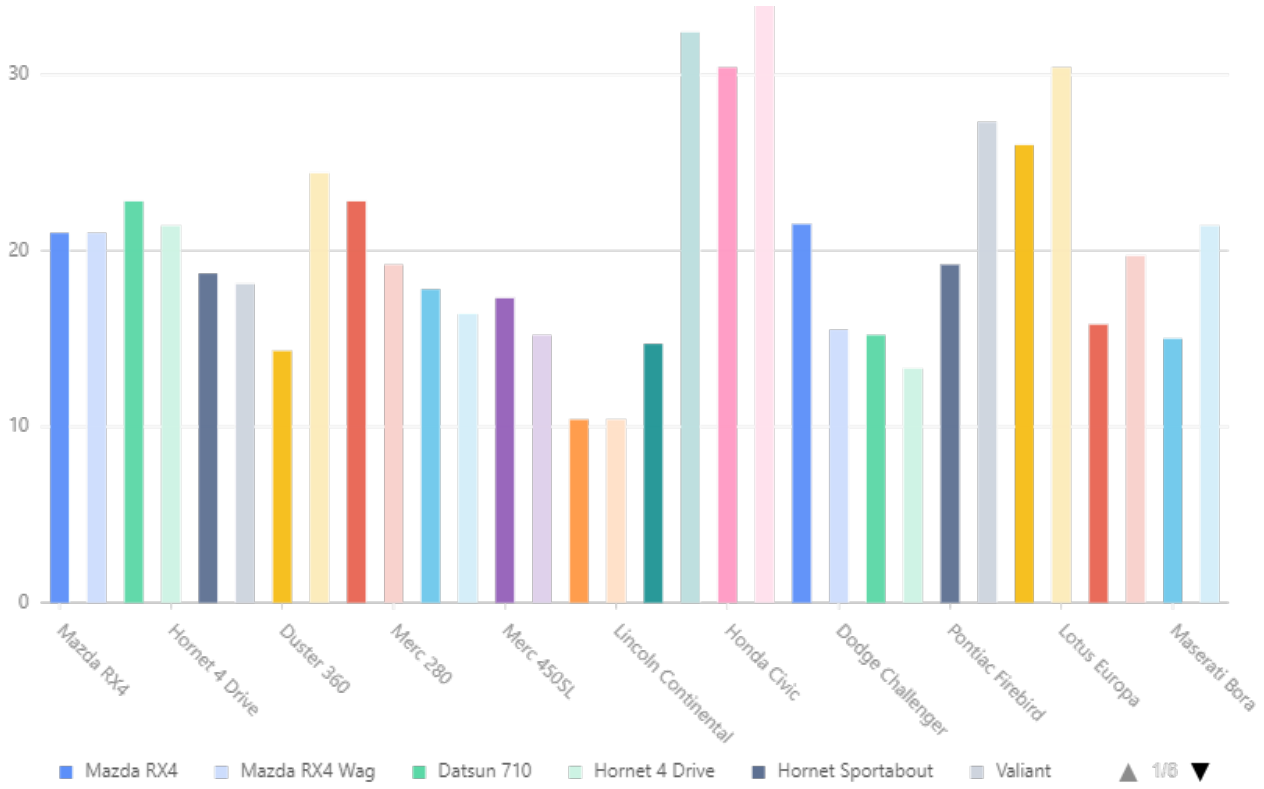


3.4.5 Legends

e.g.

```
chart = G2.Chart(height=500, width=800)
chart.data(df)
chart.interval().position('name*mpg').color('name')
chart.legend('name')
chart.render()
```

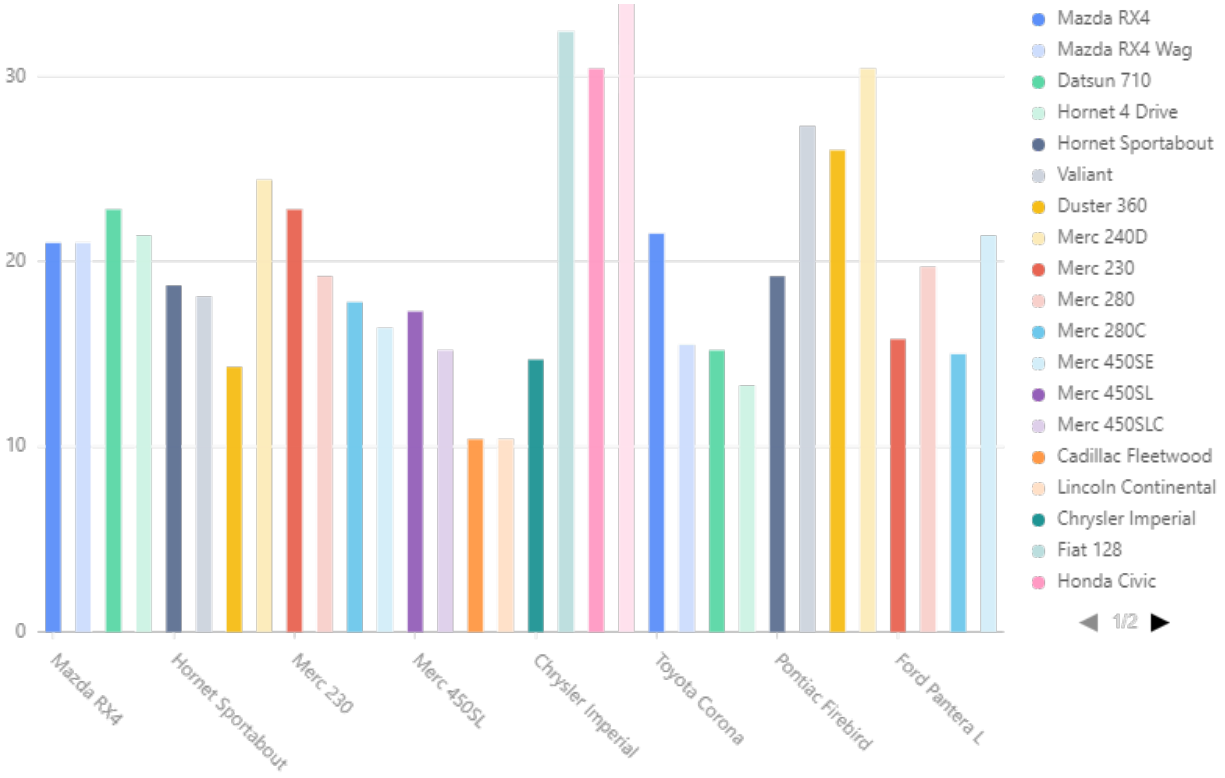
Output:



Configurations:

```
chart = G2.Chart(height=500, width=800)
chart.data(df)
chart.interval().position('name*mpg').color('name')
chart.legend('name', position = 'right', marker={'symbol':'circle'})
chart.render()
```

Output:



position : “top” | “top-left” | “top-right” | “right” | “right-top” | “right-bottom” | “left” | “left-top” | “left-bottom” | “bottom” | “bottom-left” | “bottom-right”

3.4.6 Annotations

This part has to be completed.

CHAPTER 4

Project Details

4.1 Author

Thamalu Malihttha Piyadigama, University of Moratuwa.

4.2 Home URL

https://github.com/ThamaluM/pyG2

CHAPTER 5

API

5.1 API

CHAPTER 6

Indices and tables

- genindex
- modindex
- search