vincent Documentation

Release 0.4

Rob Story, Dan Miller, et. al.

August 12, 2015

Contents

The folks at Trifacta are making it easy to build visualizations on top of D3 with Vega. Vincent makes it easy to build Vega with Python.

CHAPTER 1

Concept

The data capabilities of Python. The visualization capabilities of JavaScript.

Vincent allows you to build Vega specifications in a Pythonic way, and performs type-checking to help ensure that your specifications are correct. It also has a number of convenience chart-building methods that quickly turn Python data structures into Vega visualization grammar, enabling graphical exploration. It allows for quick iteration of visualization designs via getters and setters on grammar elements, and outputs the final visualization to JSON.

Perhaps most importantly, Vincent has Pandas-Fu, and is built specifically to allow for quick plotting of DataFrames and Series.

Contents:

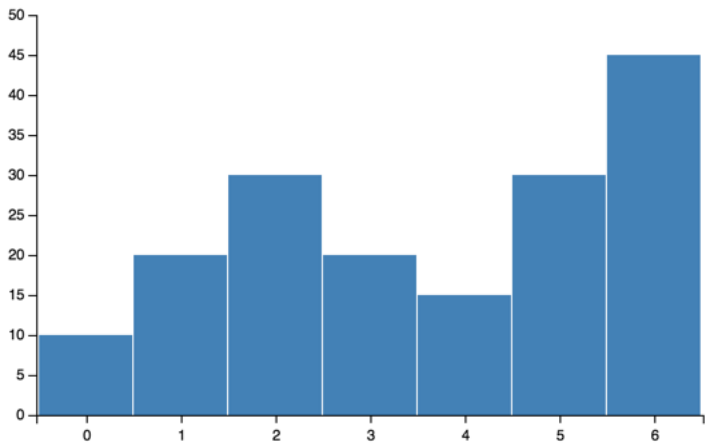
1.1 Quickstart

Have *Data*. Want to make chart.

1.1.1 Bar

Bar charts:

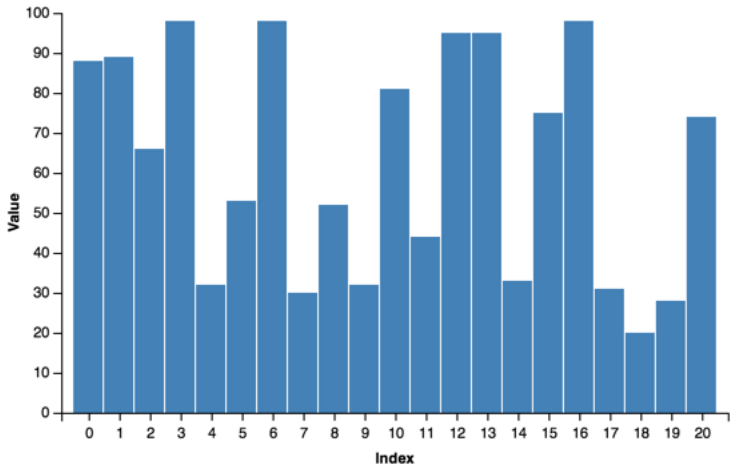
```
import vincent
bar = vincent.Bar(list_data)
```



1.1.2 Axis Labels

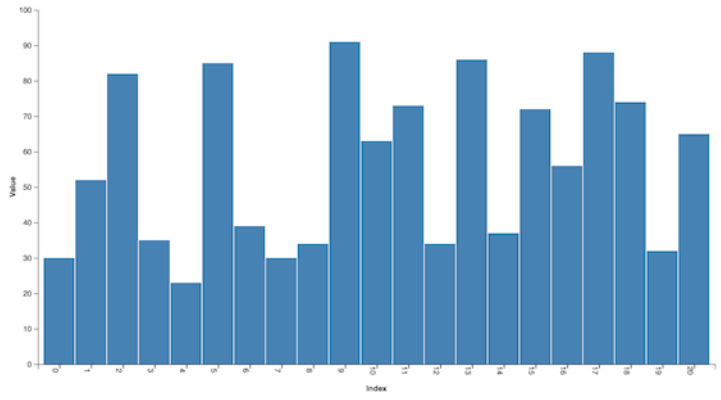
Labeling the axes is simple:

```
bar = vincent.Bar(multi_iter1['y1'])
bar.axis_titles(x='Index', y='Value')
```



You can also control aspects of the layout:

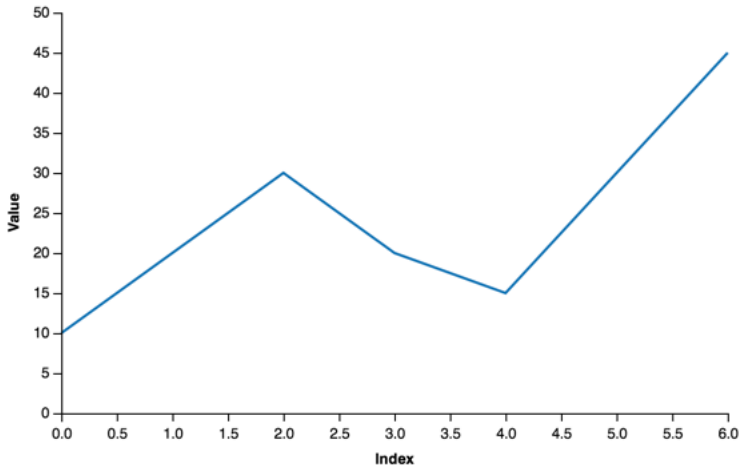
```
import vincent
from vincent import AxisProperties, PropertySet, ValueRef
bar = vincent.Bar(multi_iter1['y1'])
bar.axis_titles(x='Index', y='Value')
#rotate x axis labels
ax = AxisProperties(
         labels = PropertySet(angle=ValueRef(value=90)))
bar.axes[0].properties = ax
```



1.1.3 Line

Line charts:

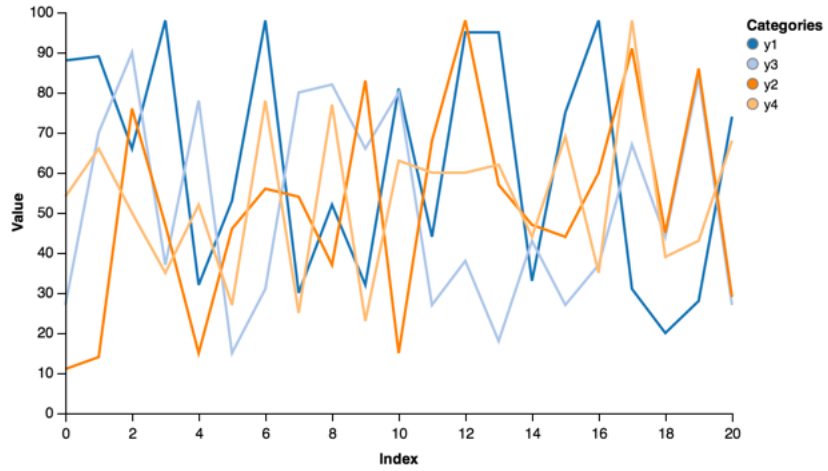
```
line = vincent.Line(list_data)
line.axis_titles(x='Index', y='Value')
```



1.1.4 Legends

Most plots create a separate set of scales that allow for categorical legends that are generated automatically. Adding the legend is straightforward:

```
line = vincent.Line(multi_iter1, iter_idx='index')
line.axis_titles(x='Index', y='Value')
line.legend(title='Categories')
```



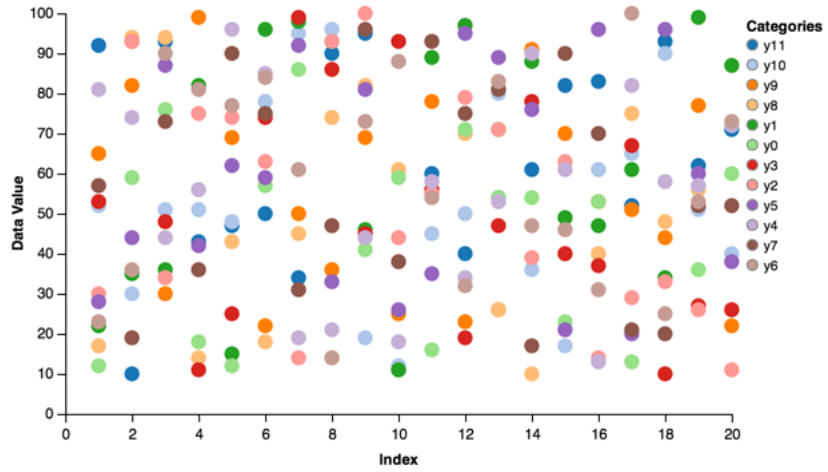
Using the stocks data:

```
line = vincent.Line(price[['GOOG', 'AAPL']])
line.axis_titles(x='Date', y='Price')
line.legend(title='GOOG vs AAPL')
```

1.1.5 Scatter

Scatter charts:

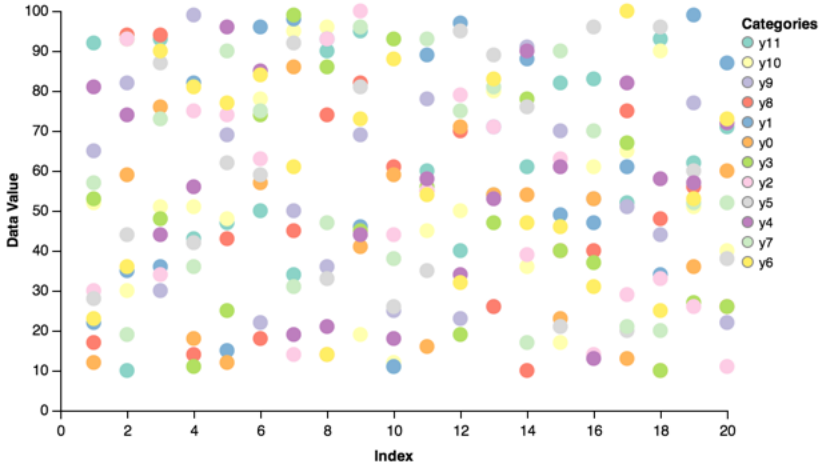
```
scatter = vincent.Scatter(multi_iter2, iter_idx='index')
scatter.axis_titles(x='Index', y='Data Value')
scatter.legend(title='Categories')
```



1.1.6 Colors

All of the Color Brewer scales are built-in to Vincent:

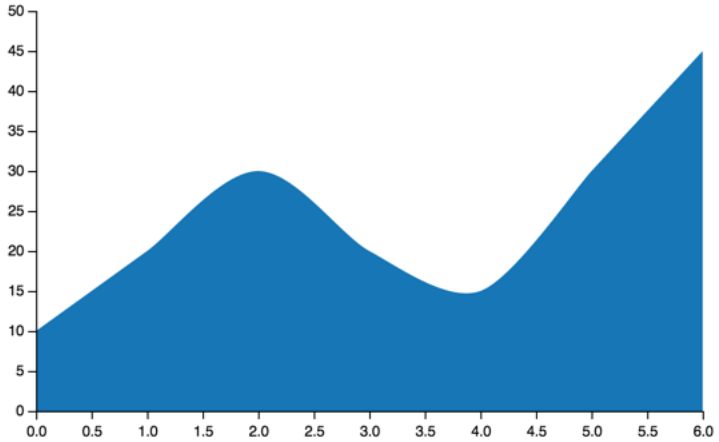
```
scatter.colors(brew='Set3')
```



1.1.7 Area

Area charts take similar data to Line:

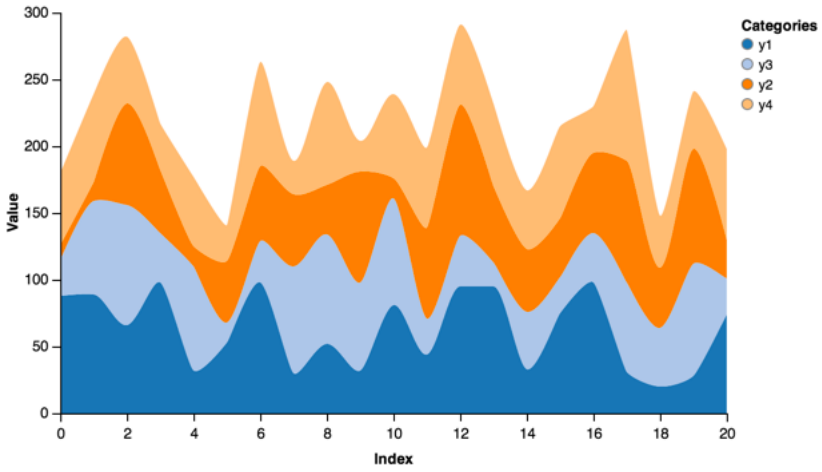
```
area = vincent.Area(list_data)
```



1.1.8 Stacked Area

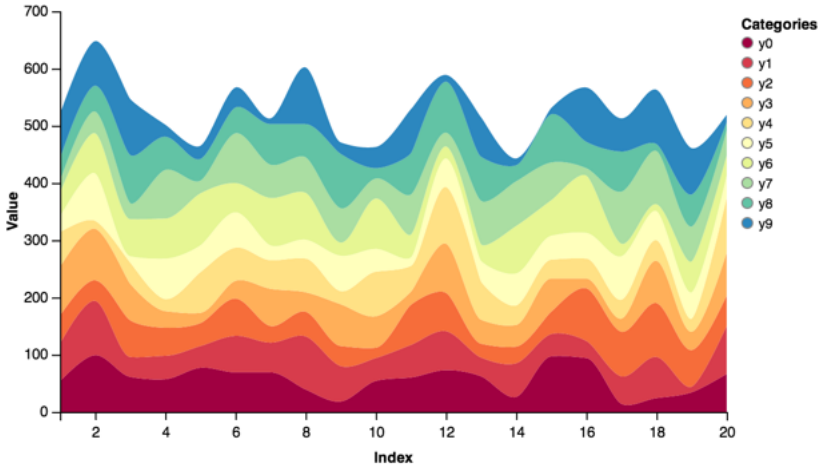
Stacked areas allow you to visualize multiple categories with one chart:

```
stacked = vincent.StackedArea(multi_iter1, iter_idx='index')
stacked.axis_titles(x='Index', y='Value')
stacked.legend(title='Categories')
```



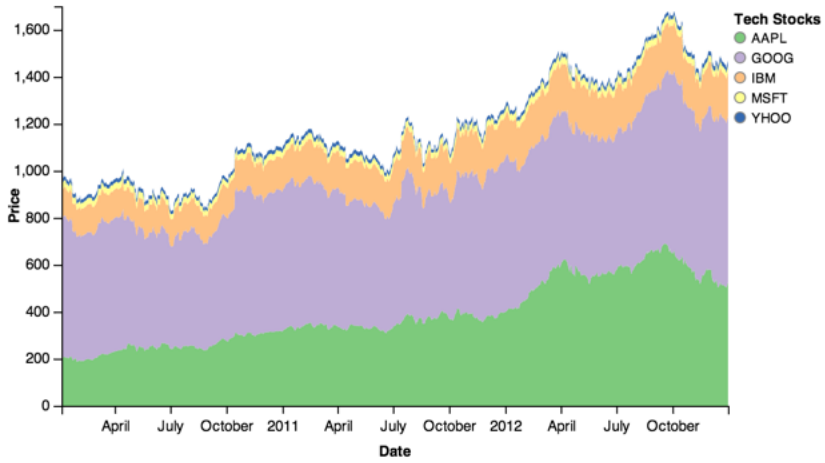
More categories, more colors:

```
stacked = vincent.StackedArea(df_1)
stacked.axis_titles(x='Index', y='Value')
stacked.legend(title='Categories')
stacked.colors(brew='Spectral')
```



Stocks data:

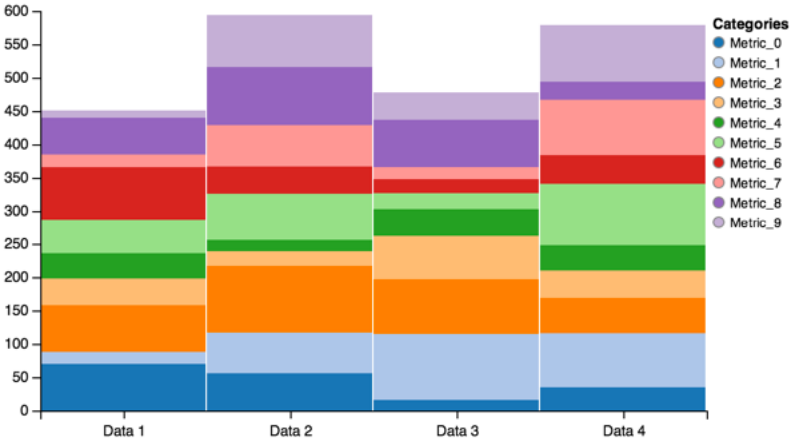
```
stacked = vincent.StackedArea(price)
stacked.axis_titles(x='Date', y='Price')
stacked.legend(title='Tech Stocks')
stacked.colors(brew='Accent')
```



1.1.9 Stacked Bar

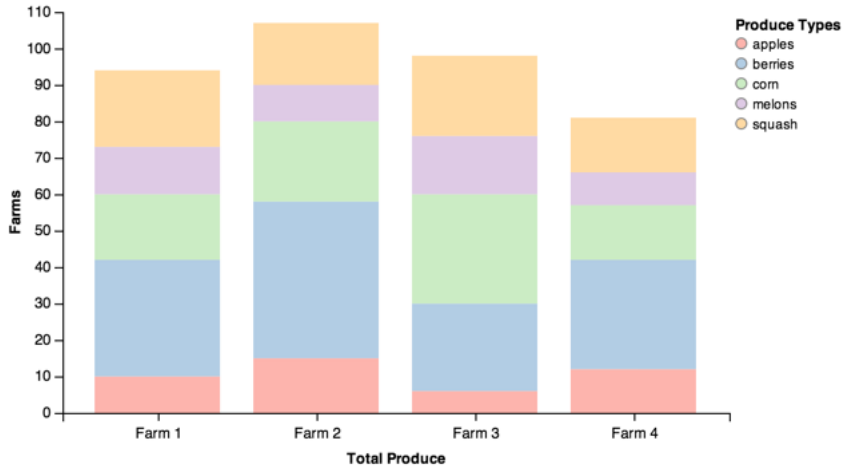
Similar to stacked areas, stacked bars let you visualize multiple ordinal categories and groups:

```
stack = vincent.StackedBar(df_2)
stack.legend(title='Categories')
```



Adding some bar padding is often helpful:

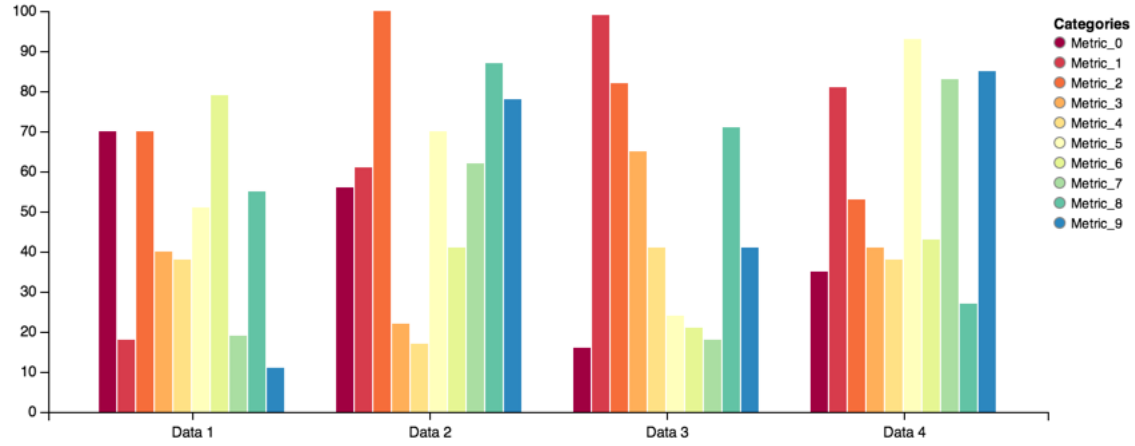
```
stack = vincent.StackedBar(df_farm)
stack.axis_titles(x='Total Produce', y='Farms')
stack.legend(title='Produce Types')
stack.scales['x'].padding = 0.2
stack.colors(brew='Pastel1')
```



1.1.10 Grouped Bar

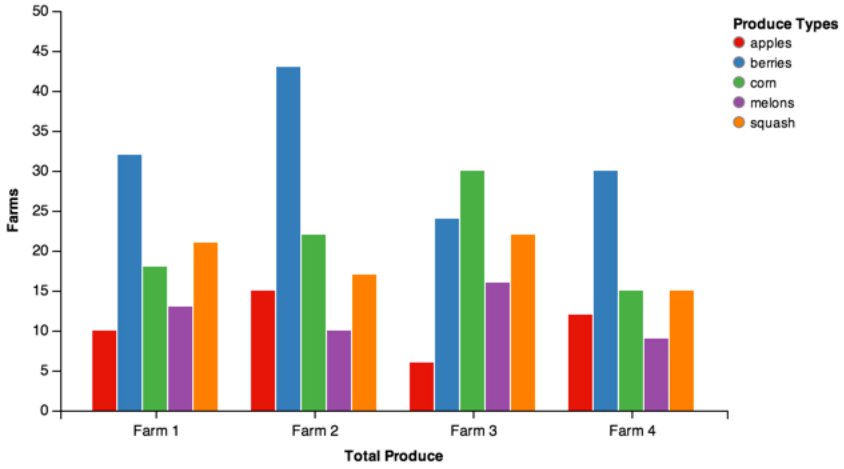
Grouped bars are another way to view grouped ordinal data:

```
group = vincent.GroupedBar(df_2)
group.legend(title='Categories')
group.colors(brew='Spectral')
group.width=750
```



Farm data:

```
group = vincent.GroupedBar(df_farm)
group.axis_titles(x='Total Produce', y='Farms')
group.legend(title='Produce Types')
group.colors(brew='Set1')
```

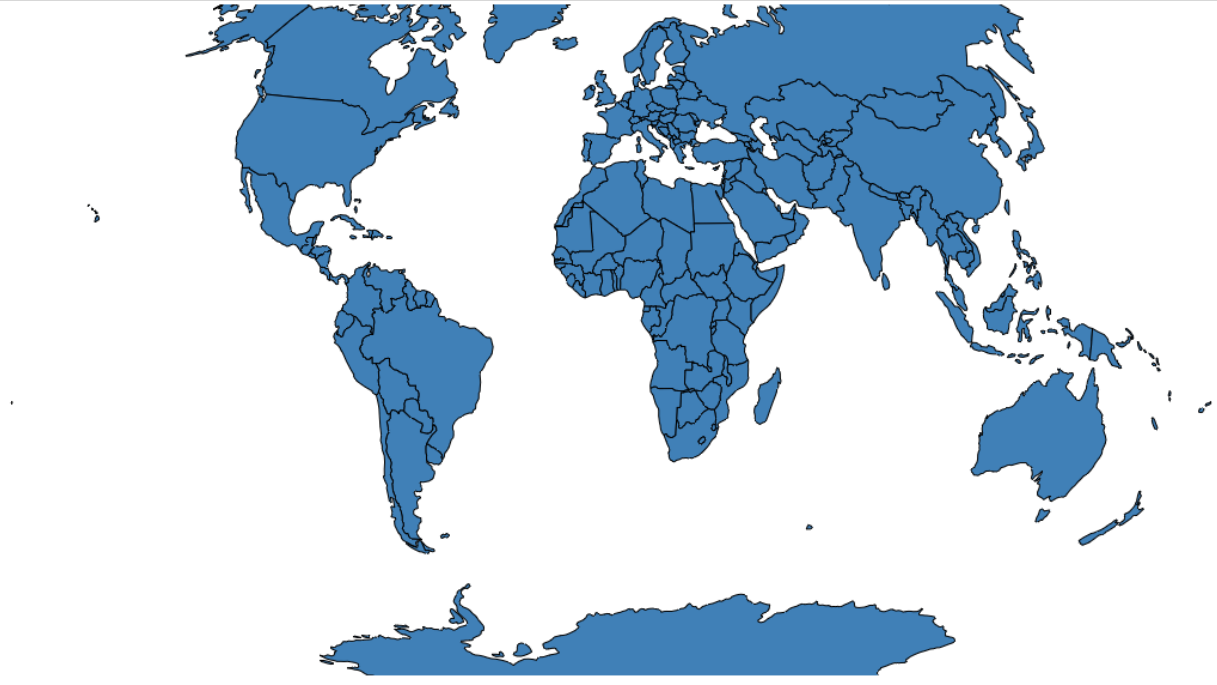


1.1.11 Simple Map

You can find all of the TopoJSON data in the vincent_map_data repo.

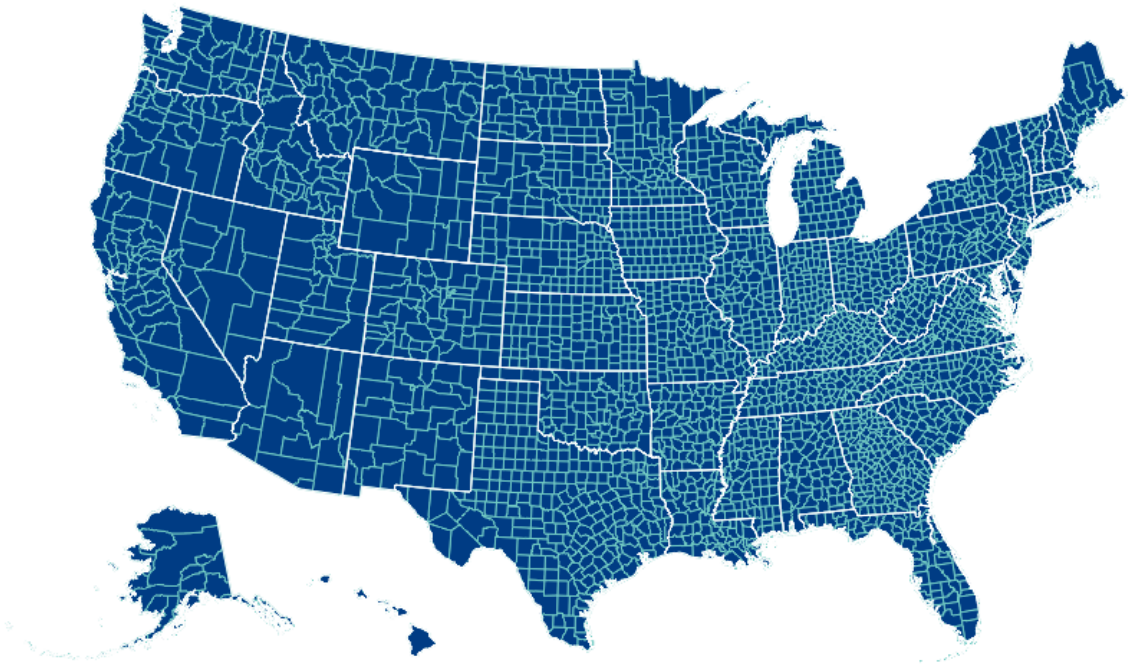
A simple world map:

```
world_topo = r'world-countries.topo.json'
geo_data = [{'name': 'countries',
             'url': world_topo,
             'feature': 'world-countries'}]

vis = vincent.Map(geo_data=geo_data, scale=200)
```



You can also pass multiple map layers:

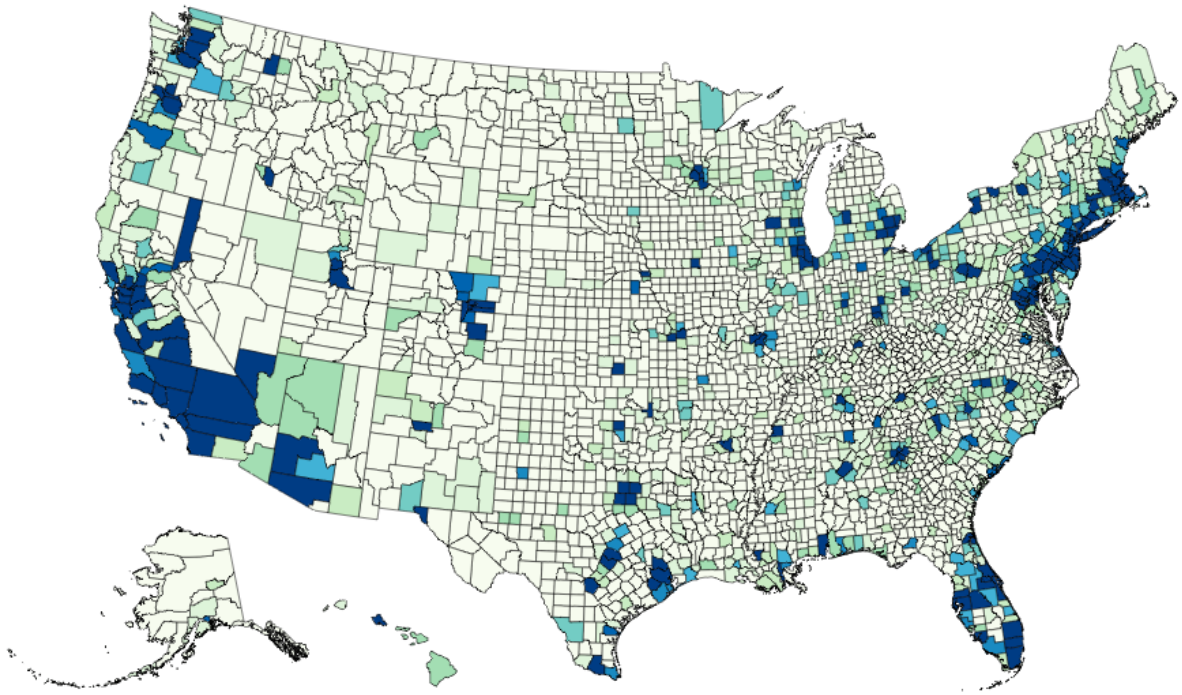
```
geo_data = [{'name': 'counties',
             'url': county_topo,
             'feature': 'us_counties.geo'},
            {'name': 'states',
             'url': state_topo,
             'feature': 'us_states.geo'}]

vis = vincent.Map(geo_data=geo_data, scale=1000, projection='albersUsa')
del vis.marks[1].properties.update
vis.marks[0].properties.update.fill.value = '#084081'
vis.marks[1].properties.enter.stroke.value = '#fff'
vis.marks[0].properties.enter.stroke.value = '#7bccc4'
```



1.1.12 Map Data Binding

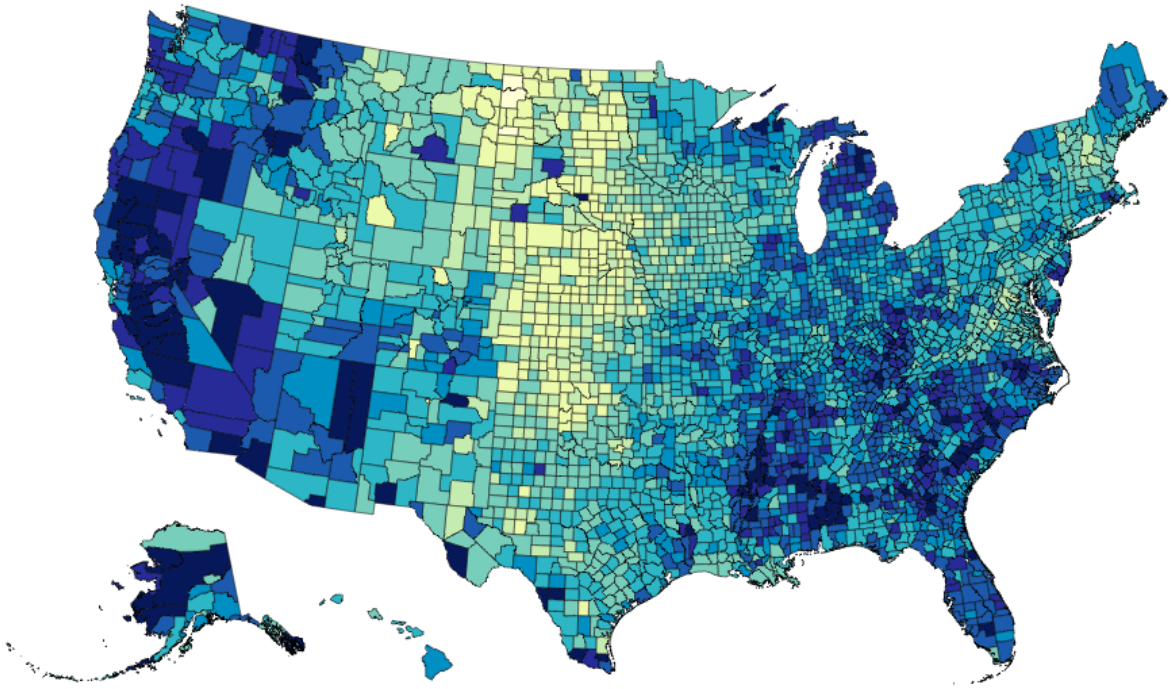
Maps can be bound to data via Pandas DataFrames to create Choropleths:

```
geo_data = [{'name': 'counties',
             'url': county_topo,
             'feature': 'us_counties.geo'}]

vis = vincent.Map(data=merged, geo_data=geo_data, scale=1100, projection='albersUsa',
          data_bind='Employed_2011', data_key='FIPS',
          map_key={'counties': 'properties.FIPS'})
vis.marks[0].properties.enter.stroke_opacity = ValueRef(value=0.5)
vis.to_json('vega.json')
```



The data can be rebound for new columns with different color brewer scales on the fly:

```
vis.rebind(column='Unemployment_rate_2011', brew='YlGnBu')
vis.to_json('vega.json')
```



1.1.13 Output

To write the Vega spec to JSON, use the `to_json` method:

```
bar.to_json('bar.json')
```

If no path is included, it writes it as a string to the REPL:

```
>>>bar.to_json()
#Really long string of JSON
```

A simple HTML template to read and display the chart is built-in to Vincent, and can be output along with the JSON:

```
>>>bar.to_json('bar.json', html_out=True, html_path='bar_template.html')
```

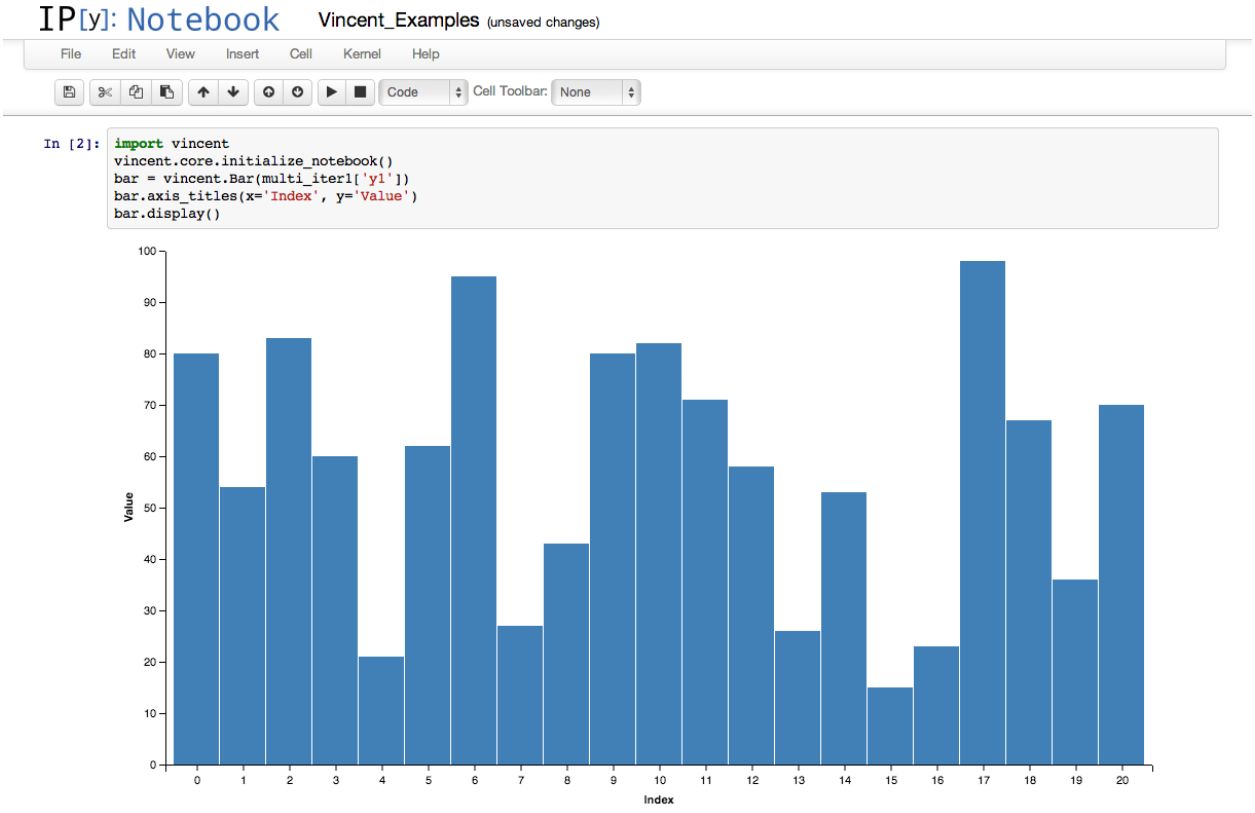
The HTML will need to be served somehow- luckily, Python makes this easy. Start a simple HTTP Server, then point your browser to localhost:8000:

```
$python -m SimpleHTTPServer 8000
```

1.1.14 IPython integration

It is possible to run the above examples inside IPython notebook by adding a few extra lines:

```
import vincent
vincent.core.initialize_notebook()

bar = vincent.Bar(multi_iter1['y1'])
bar.axis_titles(x='Index', y='Value')
bar.display()
```



1.1.15 Data

These are the datasets used in the *Quickstart* charts above:

```
import pandas as pd
import random

#Iterable
list_data = [10, 20, 30, 20, 15, 30, 45]

#Dicts of iterables
cat_1 = ['y1', 'y2', 'y3', 'y4']
index_1 = range(0, 21, 1)
multi_iter1 = {'index': index_1}
for cat in cat_1:
    multi_iter1[cat] = [random.randint(10, 100) for x in index_1]

cat_2 = ['y' + str(x) for x in range(0, 10, 1)]
index_2 = range(1, 21, 1)
multi_iter2 = {'index': index_2}
for cat in cat_2:
    multi_iter2[cat] = [random.randint(10, 100) for x in index_2]

#Pandas
import pandas as pd

farm_1 = {'apples': 10, 'berries': 32, 'squash': 21, 'melons': 13, 'corn': 18}
farm_2 = {'apples': 15, 'berries': 43, 'squash': 17, 'melons': 10, 'corn': 22}
farm_3 = {'apples': 6, 'berries': 24, 'squash': 22, 'melons': 16, 'corn': 30}
farm_4 = {'apples': 12, 'berries': 30, 'squash': 15, 'melons': 9, 'corn': 15}

farm_data = [farm_1, farm_2, farm_3, farm_4]
farm_index = ['Farm 1', 'Farm 2', 'Farm 3', 'Farm 4']
df_farm = pd.DataFrame(farm_data, index=farm_index)

#As DataFrames
index_3 = multi_iter2.pop('index')
df_1 = pd.DataFrame(multi_iter2, index=index_3)
df_1 = df_1.reindex(columns=sorted(df_1.columns))

cat_4 = ['Metric_' + str(x) for x in range(0, 10, 1)]
index_4 = ['Data 1', 'Data 2', 'Data 3', 'Data 4']
data_3 = {}
for cat in cat_4:
    data_3[cat] = [random.randint(10, 100) for x in index_4]
df_2 = pd.DataFrame(data_3, index=index_4)

import pandas.io.data as web
all_data = {}
for ticker in ['AAPL', 'GOOG', 'IBM', 'YHOO', 'MSFT']:
    all_data[ticker] = web.get_data_yahoo(ticker, '1/1/2010', '1/1/2013')
price = pd.DataFrame({tic: data['Adj Close']
                      for tic, data in all_data.iteritems()})

#Map Data Binding
import json
import pandas as pd
#Map the county codes we have in our geometry to those in the
#county_data file, which contains additional rows we don't need
```

```
with open('us_counties.topo.json', 'r') as f:
    get_id = json.load(f)

#A little FIPS code munging
new_geoms = []
for geom in get_id['objects']['us_counties.geo']['geometries']:
    geom['properties']['FIPS'] = int(geom['properties']['FIPS'])
    new_geoms.append(geom)

get_id['objects']['us_counties.geo']['geometries'] = new_geoms

with open('us_counties.topo.json', 'w') as f:
    json.dump(get_id, f)

#Grab the FIPS codes and load them into a dataframe
geometries = get_id['objects']['us_counties.geo']['geometries']
county_codes = [x['properties']['FIPS'] for x in geometries]
county_df = pd.DataFrame({'FIPS': county_codes}, dtype=str)
county_df = county_df.astype(int)

#Read into Dataframe, cast to string for consistency
df = pd.read_csv('data/us_county_data.csv', na_values=[' '])
df['FIPS_Code'] = df['FIPS'].astype(str)

#Perform an inner join, pad NA's with data from nearest county
merged = pd.merge(df, county_df, on='FIPS', how='inner')
merged = merged.fillna(method='pad')
```

1.2 Charts Library

Vincent provides a charts library that allows for rapid creation and iteration of different chart types, with data inputs from a number of Python data structures. This library is built using the Vincent API to construct Vega grammar, with some adding conveniences for simple data input.

1.2.1 Chart

The `Chart` class is a base container for ingesting data and creating a Vega scaffold:

```
>>>chart = vincent.Chart([10, 20, 30, 40, 50])
>>>chart.grammar()
{u'axes': [],
 u'data': [{u'name': u'table',
   u'values': [{u'col': u'data', u'idx': 0, u'val': 10},
    {u'col': u'data', u'idx': 1, u'val': 20},
    {u'col': u'data', u'idx': 2, u'val': 30},
    {u'col': u'data', u'idx': 3, u'val': 40},
    {u'col': u'data', u'idx': 4, u'val': 50}]}],
 u'height': 300,
 u'legends': [],
 u'marks': [],
 u'padding': {u'bottom': 50, u'left': 50, u'right': 100, u'top': 10},
 u'scales': [],
 u'width': 500}
```

Note the use of `chart.grammar()` to output the specification to Python data structures. If at any point you wish to view the current specification, use the `grammar()` call. This works at almost any level of nesting depth as well:

```
>>>chart.data[0].grammar()
{u'name': u'table',
 u'values': [{u'col': u'data', u'idx': 0, u'val': 10},
  {u'col': u'data', u'idx': 1, u'val': 20},
  {u'col': u'data', u'idx': 2, u'val': 30},
  {u'col': u'data', u'idx': 3, u'val': 40},
  {u'col': u'data', u'idx': 4, u'val': 50}]}
```

Charts will take a number of different data sources. All of the following produce equivalent data output:

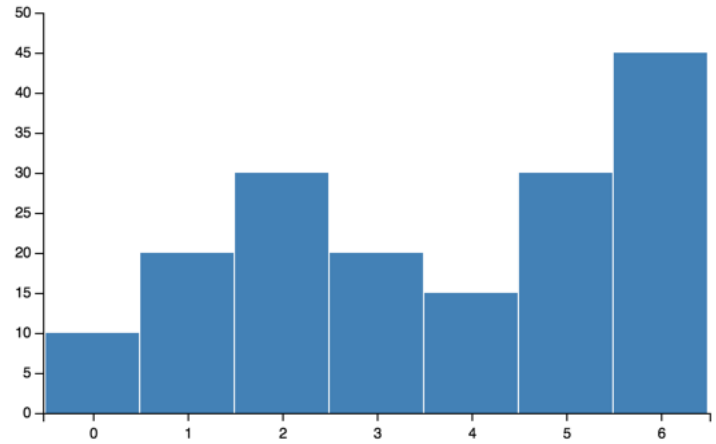
```
list_data = [10, 20, 30, 40, 50]
dict_of_iters = {'x': [0, 1, 2, 3, 4], 'data': [10, 20, 30, 40, 50]}
series = pd.Series([10, 20, 30, 40, 50])
dataframe = pd.DataFrame({'data': [10, 20, 30, 40, 50]})
#All of the following are equivalent
chart = vincent.Chart(list_data)
chart = vincent.Chart(dict_of_iters, iter_idx='x')
chart = vincent.Chart(series)
chart = vincent.Chart(dataframe)
```

`vincent.Chart` is the abstract base class on which all other chart types are built.

1.2.2 Bar

A subclass of chart:

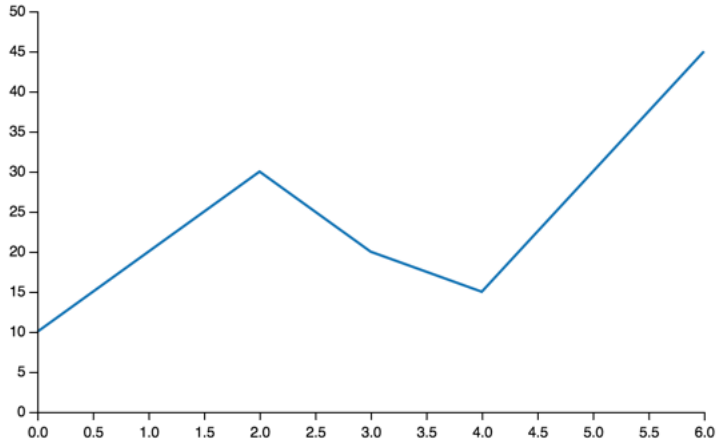
```
bar = vincent.Bar([10, 20, 30, 20, 15, 30, 45])
```



1.2.3 Line

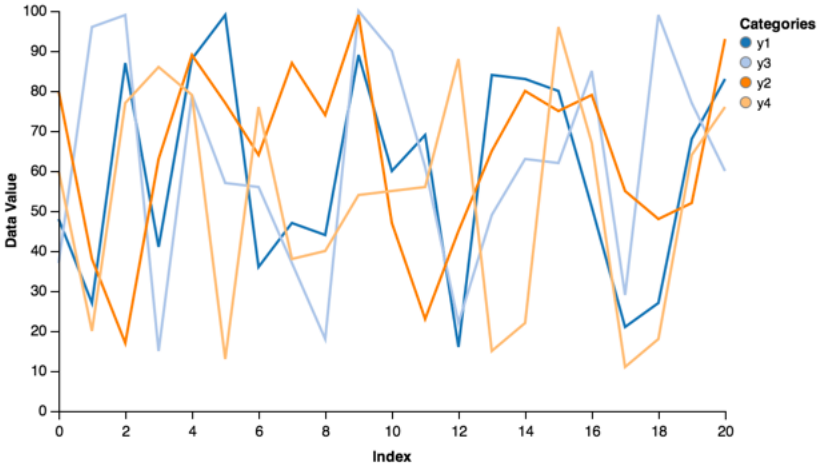
Similar to bar, you can plot just one line:

```
line = vincent.Line([10, 20, 30, 20, 15, 30, 45])
```



Multiple lines can also be plotted easily:

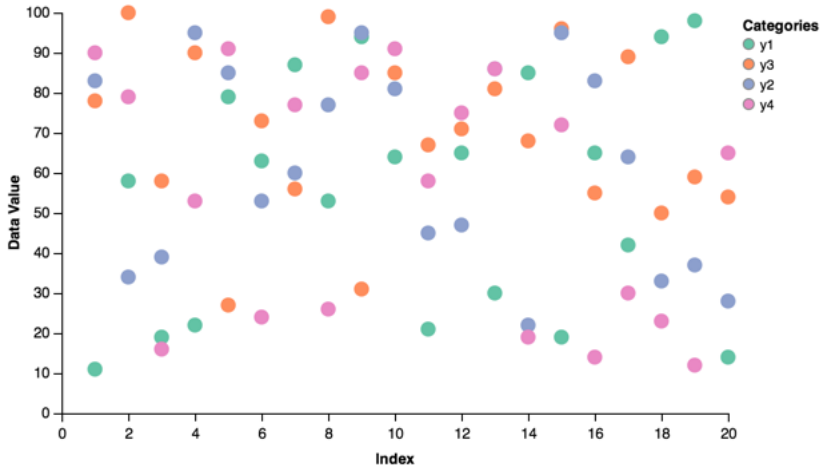
```
cats = ['y1', 'y2', 'y3', 'y4']
index = range(1, 21, 1)
multi_iter1 = {'index': index}
for cat in cats:
    multi_iter1[cat] = [random.randint(10, 100) for x in index]
lines = vincent.Line(multi_iter1, iter_idx='index')
lines.legend(title='Categories')
lines.axis_titles(x='Index', y='Data Value')
```



1.2.4 Scatter

Using the same data from above, with some different color choices:

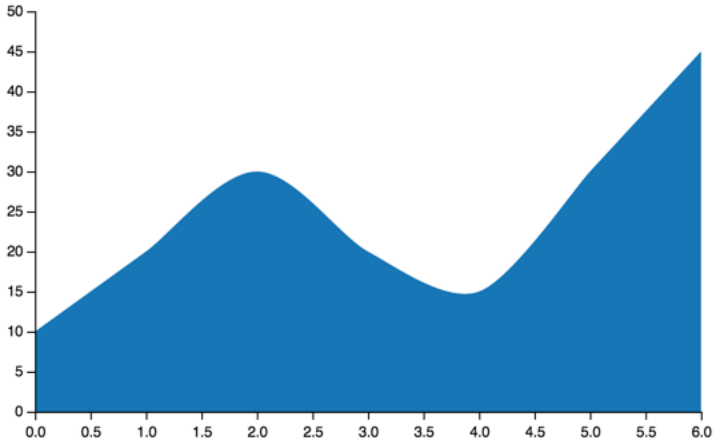
```
scatter = vincent.Scatter(data, iter_idx='index')
scatter.axis_titles(x='Index', y='Data Value')
scatter.legend(title='Categories')
scatter.colors(brew='Set2')
```



1.2.5 Area

Area charts are basically an extension of Line:

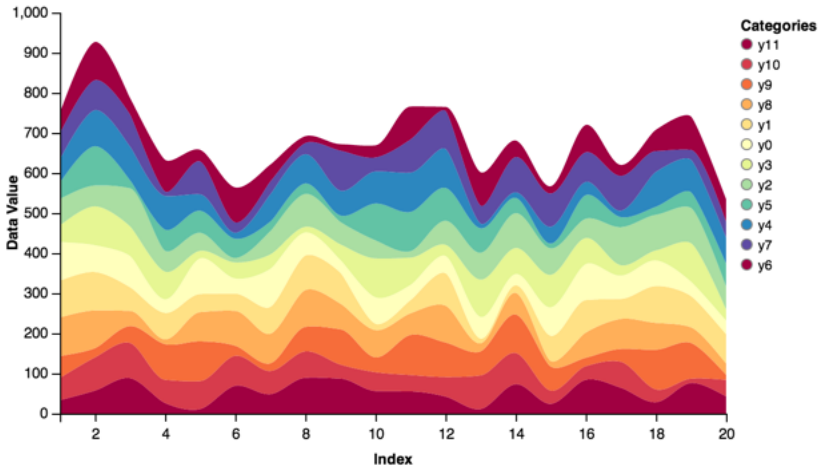
```
area = vincent.Area([10, 20, 30, 20, 15, 30, 45])
```



1.2.6 Stacked Area

Stacked areas allow you to visualize multiple pieces of data with an area-type chart. Lets look at a large number of categories:

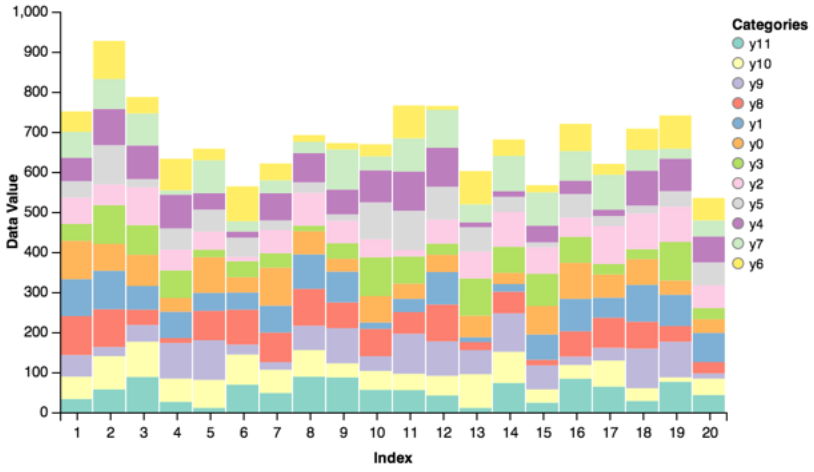
```
cats = ['y' + str(x) for x in range(0, 12, 1)]
index = range(1, 21, 1)
data = {'index': index}
for cat in cats:
    data[cat] = [random.randint(10, 100) for x in index]
stacked = vincent.StackedArea(data, iter_idx='index')
stacked.axis_titles(x='Index', y='Data Value')
stacked.legend(title='Categories')
stacked.colors(brew='Spectral')
```



1.2.7 Stacked Bar

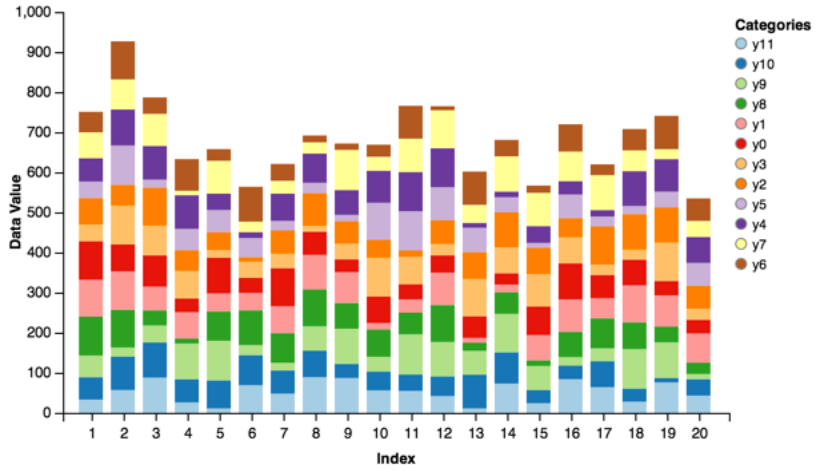
A variation that allows you to stack bars similar to areas for ordinal quantities. Using the data from above:

```
stacked = vincent.StackedBar(data, iter_idx='index')
stacked.axis_titles(x='Index', y='Data Value')
stacked.legend(title='Categories')
stacked.colors(brew='Set3')
```



For bar charts with large numbers of bars, its often useful to pad each bar:

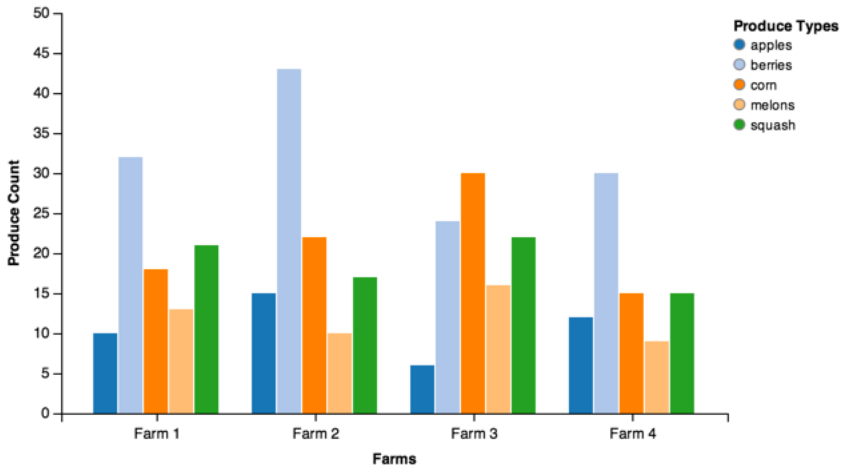
```
stacked.scales['x'].padding = 0.2
stacked.colors(brew='Paired')
```



1.2.8 Grouped Bar

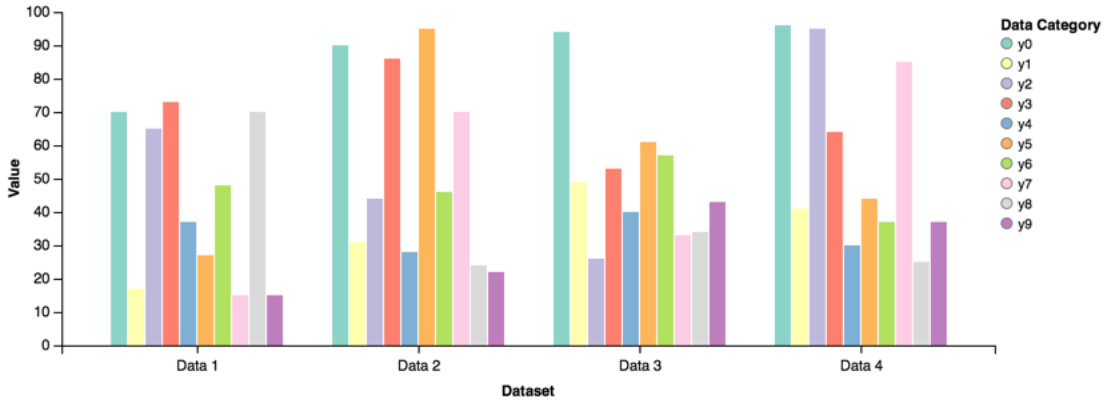
It's often useful to plot bars with quantities associated with different groups. For example, produce output at different farms:

```
import pandas as pd

farm_1 = {'apples': 10, 'berries': 32, 'squash': 21, 'melons': 13, 'corn': 18}
farm_2 = {'apples': 15, 'berries': 43, 'squash': 17, 'melons': 10, 'corn': 22}
farm_3 = {'apples': 6, 'berries': 24, 'squash': 22, 'melons': 16, 'corn': 30}
farm_4 = {'apples': 12, 'berries': 30, 'squash': 15, 'melons': 9, 'corn': 15}

data = [farm_1, farm_2, farm_3, farm_4]
index = ['Farm 1', 'Farm 2', 'Farm 3', 'Farm 4']

df = pd.DataFrame(data, index=index)

grouped = vincent.GroupedBar(df)
grouped.axis_titles(x='Farms', y='Produce Count')
grouped.legend(title='Produce Types')
```



Currently grouped sets only work with Pandas DataFrames, but that should change soon. In the meantime, getting data into a DataFrame is straightforward:

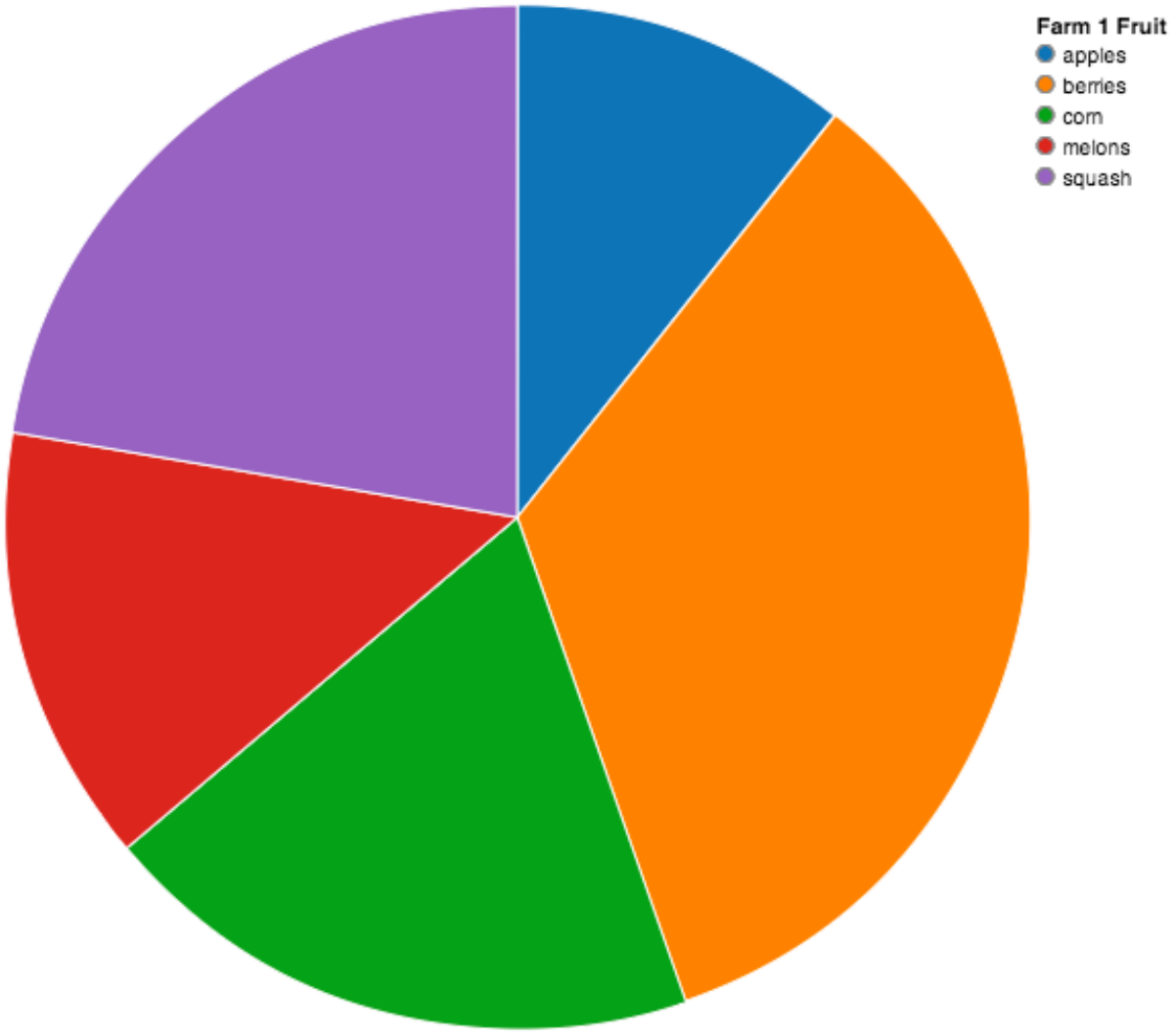
```
cats = ['y' + str(x) for x in range(0, 10, 1)]
index = ['Data 1', 'Data 2', 'Data 3', 'Data 4']
data = {}
for cat in cats:
    data[cat] = [random.randint(10, 100) for x in index]
df = pd.DataFrame(data, index=index)
grouped = vincent.GroupedBar(df)
grouped.width = 700
grouped.height = 250
grouped.colors(brew='Set3')
grouped.axis_titles(x='Dataset', y='Value')
grouped.legend(title='Data Category')
```



1.2.9 Pie/Donut Chart

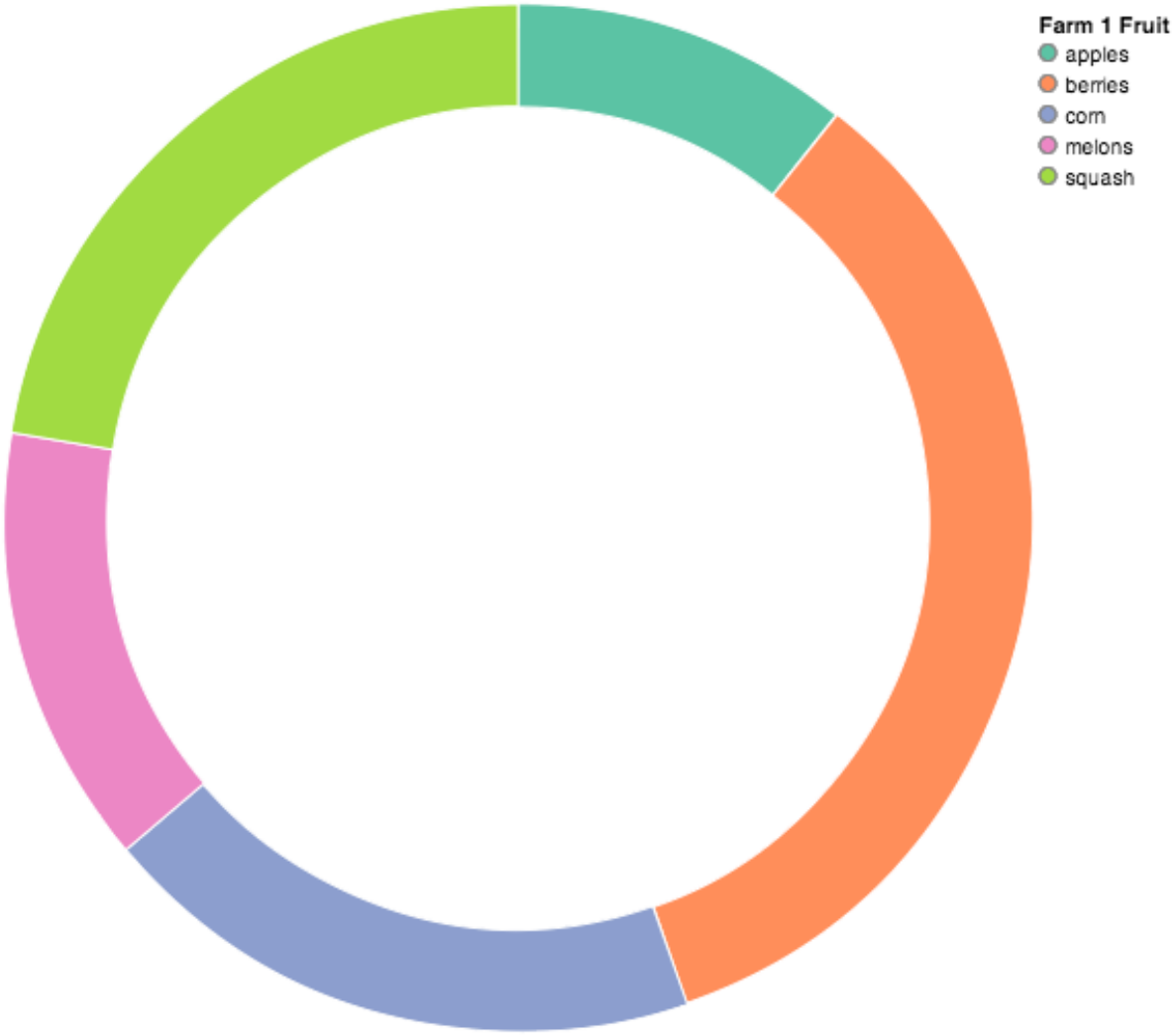
Pie chart outer radius defaults to 1/2 min(width/height):

```
pie = vincent.Pie(farm_1)
pie.legend('Farm 1 Fruit')
```



Donut charts can be created by passing an inner radius:

```
donut = vincent.Pie(farm_1, inner_radius=200)
donut.colors(brew="Set2")
donut.legend('Farm 1 Fruit')
```



1.2.10 Simple Map

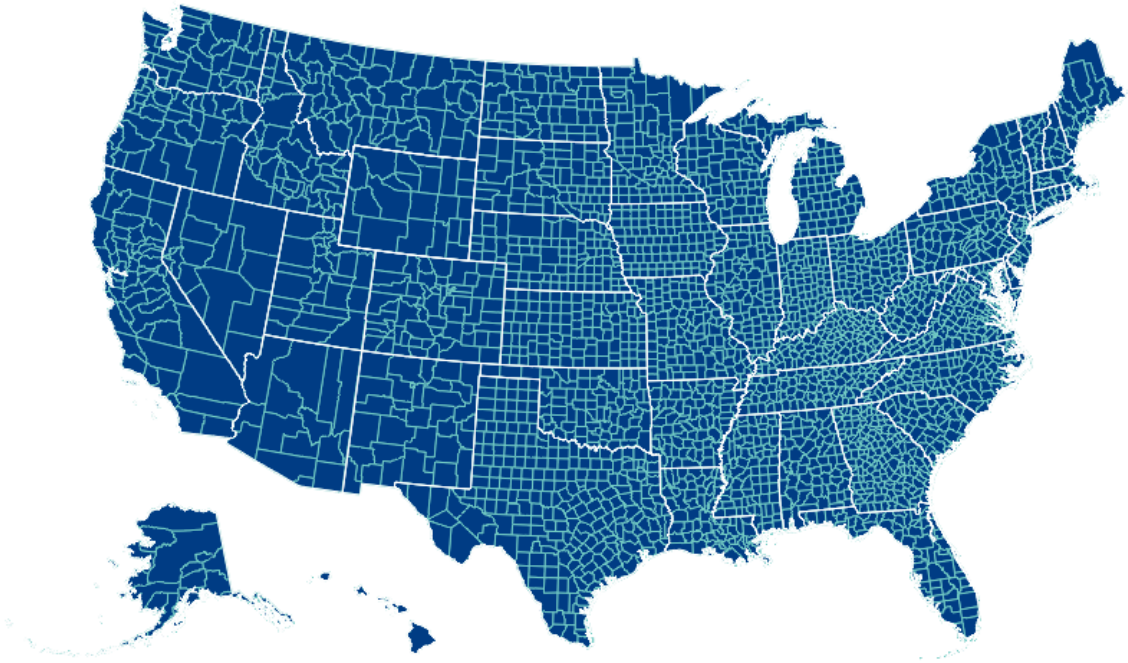
You can find all of the TopoJSON data in the vincent_map_data repo.

A simple world map:

```
world_topo = r'world-countries.topo.json'
geo_data = [{'name': 'countries',
             'url': world_topo,
             'feature': 'world-countries'}]

vis = Map(geo_data=geo_data, scale=200)
```

You can also pass multiple map layers:

```
geo_data = [{'name': 'counties',
             'url': county_topo,
             'feature': 'us_counties.geo'},
            {'name': 'states',
             'url': state_topo,
             'feature': 'us_states.geo'}
             ]

vis = Map(geo_data=geo_data, scale=1000, projection='albersUsa')
del vis.marks[1].properties.update
vis.marks[0].properties.update.fill.value = '#084081'
vis.marks[1].properties.enter.stroke.value = '#fff'
vis.marks[0].properties.enter.stroke.value = '#7bccc4'
```

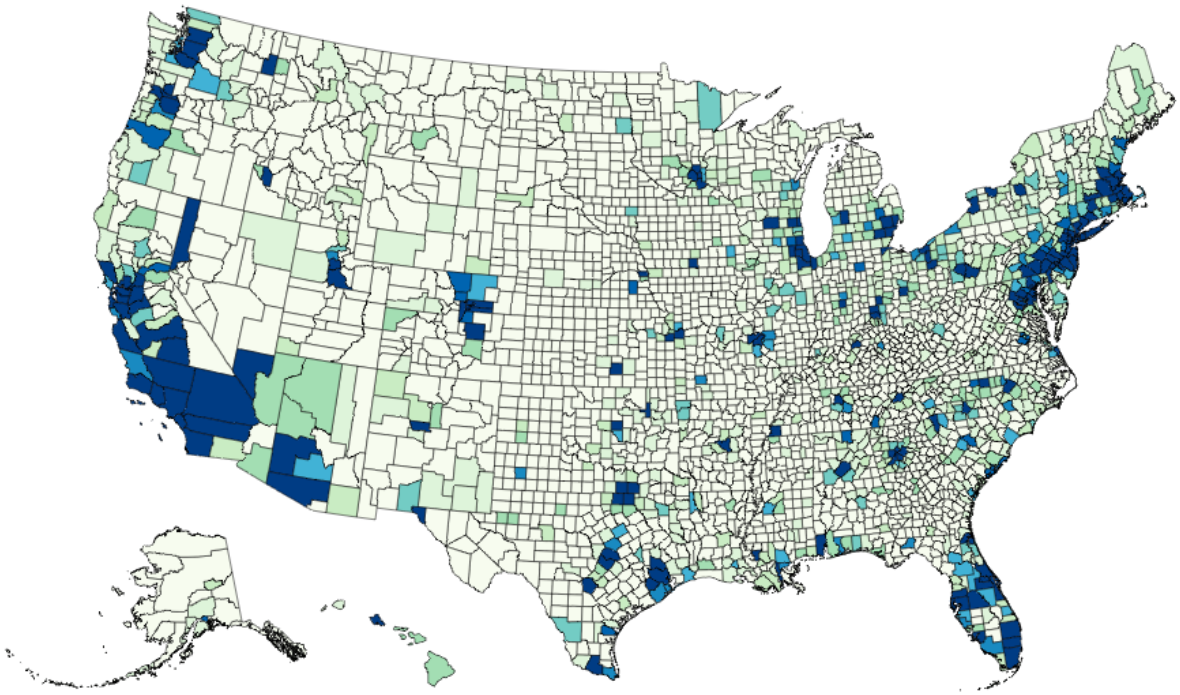


1.2.11 Map Data Binding

Maps can be bound to data via Pandas DataFrames to create Choropleths, with some data munging to match keys:

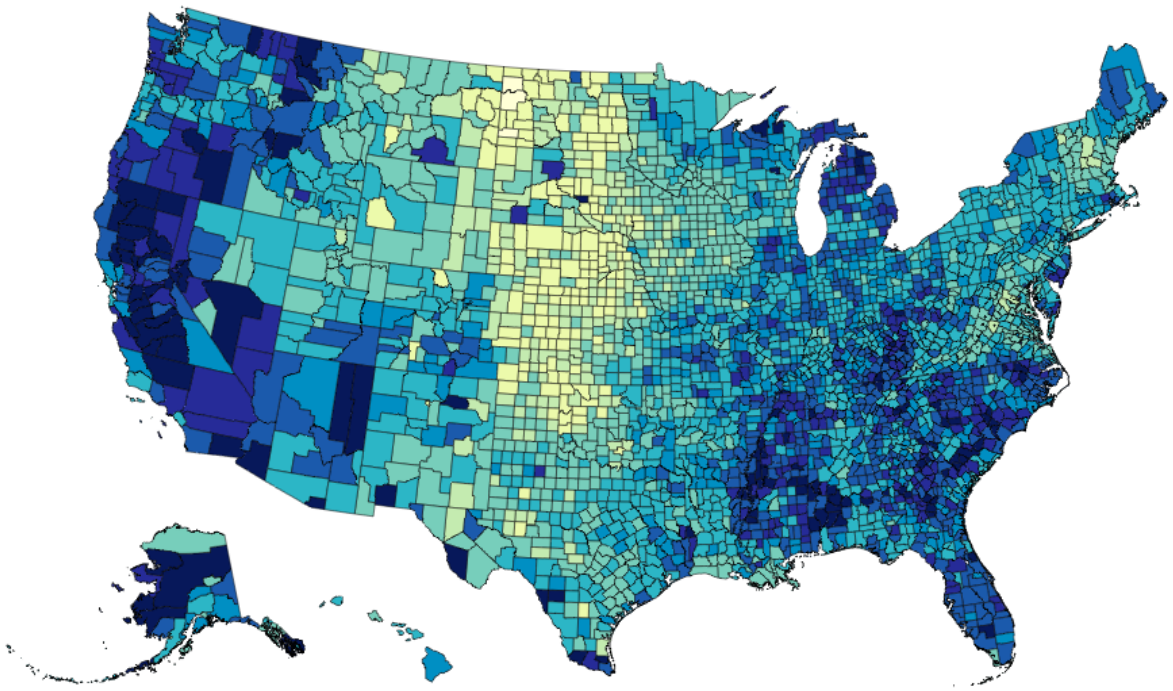
```
import json
import pandas as pd
#Map the county codes we have in our geometry to those in the
#county_data file, which contains additional rows we don't need
with open('us_counties.topo.json', 'r') as f:
    get_id = json.load(f)

#A little FIPS code munging
new_geoms = []
for geom in get_id['objects']['us_counties.geo']['geometries']:
    geom['properties']['FIPS'] = int(geom['properties']['FIPS'])
    new_geoms.append(geom)

get_id['objects']['us_counties.geo']['geometries'] = new_geoms

with open('us_counties.topo.json', 'w') as f:
    json.dump(get_id, f)

#Grab the FIPS codes and load them into a dataframe
geometries = get_id['objects']['us_counties.geo']['geometries']
county_codes = [x['properties']['FIPS'] for x in geometries]
county_df = pd.DataFrame({'FIPS': county_codes}, dtype=str)
county_df = county_df.astype(int)

#Read into Dataframe, cast to string for consistency
df = pd.read_csv('data/us_county_data.csv', na_values=[' '])
```

```
df['FIPS_Code'] = df['FIPS'].astype(str)

#Perform an inner join, pad NA's with data from nearest county
merged = pd.merge(df, county_df, on='FIPS', how='inner')
merged = merged.fillna(method='pad')

geo_data = [{'name': 'counties',
             'url': county_topo,
             'feature': 'us_counties.geo'}]

vis = Map(data=merged, geo_data=geo_data, scale=1100, projection='albersUsa',
          data_bind='Employed_2011', data_key='FIPS',
          map_key={'counties': 'properties.FIPS'})
vis.marks[0].properties.enter.stroke_opacity = ValueRef(value=0.5)
vis.to_json('vega.json')
```

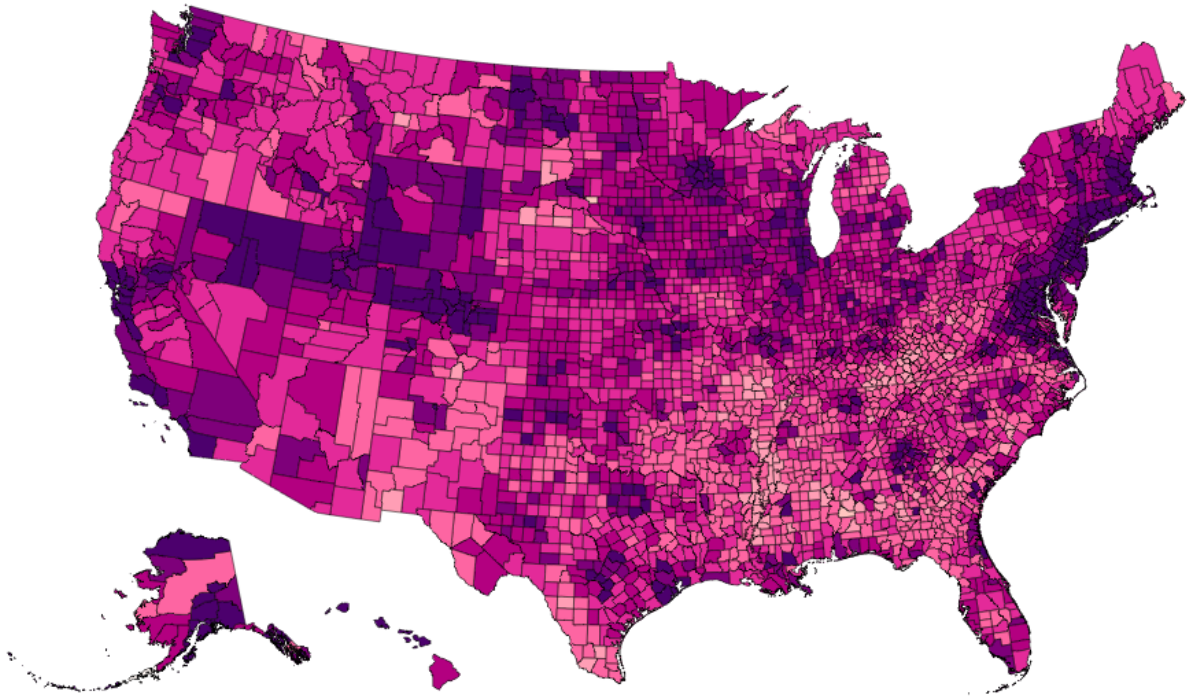


The data can be rebound for new columns with different color brewer scales on the fly:

```
vis.rebind(column='Unemployment_rate_2011', brew='YlGnBu')
vis.to_json('vega.json')
```



```
vis.rebind(column='Median_Household_Income_2011', brew='RdPu')
vis.to_json('vega.json')
```



1.3 Building Vega with Vincent

The Vincent API attempts to map 1:1 to Vega grammar through a set of object relational classes. You can build complex Vega grammar directly with Vincent via simple getters and setters.

1.3.1 Grammar

Here is an example of a simple set of marks for a bar chart in Vega JSON.

```
    {
  "type": "rect",
  "from": {"data": "table"},
  "properties": {
    "enter": {
      "x": {"scale": "x", "field": "data.idx"},
      "width": {"scale": "x", "band": true, "offset": -1},
      "y": {"scale": "y", "field": "data.val"},
      "y2": {"scale": "y", "value": 0}
    },
    "update": {
      "fill": {"value": "steelblue"}
    },
    "hover": {
      "fill": {"value": "red"}
    }
  }
}
```

Here's the same thing being built with Vincent's API:

```
from vincent import *

enter_props = PropertySet(x=ValueRef(scale='x', field="data.idx"),
                          y=ValueRef(scale='y', field="data.val"),
                          width=ValueRef(scale='x', band=True, offset=-1),
                          y2=ValueRef(scale='y', value=0))

update_props = PropertySet(fill=ValueRef(value='steelblue'))

mark = Mark(type='rect', from_=MarkRef(data='table'),
            properties=MarkProperties(enter=enter_props,update=update_props))
```

If you wanted to transform this into a line chart, Vincent makes spec changes simple. Let's change the fill color. Assuming that your Vincent object is named `vis`:

```
vis.marks[0].properties.update.fill.value = 'red'
```

If you want to check on the grammar, you can call `grammar()` to return a Python data structure representation of the Vega grammar at almost any level of nesting depth:

```
>>>vis.marks[0].properties.grammar()
{u'enter': {u'width': {u'band': True, u'offset': -1, u'scale': u'x'},
 u'x': {u'field': u'data.idx', u'scale': u'x'},
 u'y': {u'field': u'data.val', u'scale': u'y'},
 u'y2': {u'scale': u'y', u'value': 0}},
u'update': {u'fill': {u'value': u'steelblue'}}}
```

Vincent also performs type-checking on grammar elements to try and avoid grammar errors:

```
>>>vis.marks[0].properties.enter.y2.scale = 1

ValueError: scale must be str
```

The best way to get an idea of how to build Vega grammar with Vincent is to see the examples in the Github Repo . Building a bar chart from scratch using the Vincent API looks as follows:

```
from vincent import *

vis = Visualization(width=500, height=300)
vis.scales['x'] = Scale(name='x', type='ordinal', range='width',
                        domain=DataRef(data='table', field="data.idx"))
vis.scales['y'] = Scale(name='y', range='height', nice=True,
                        domain=DataRef(data='table', field="data.val"))
vis.axes.extend([Axis(type='x', scale='x'),
                 Axis(type='y', scale='y')])

#Marks
enter_props = PropertySet(x=ValueRef(scale='x', field="data.idx"),
                          y=ValueRef(scale='y', field="data.val"),
                          width=ValueRef(scale='x', band=True, offset=-1),
                          y2=ValueRef(scale='y', value=0))

update_props = PropertySet(fill=ValueRef(value='steelblue'))

mark = Mark(type='rect', from_=MarkRef(data='table'),
            properties=MarkProperties(enter=enter_props,
                                      update=update_props))
vis.marks.append(mark)

data = Data.from_iter([10, 20, 30, 40, 50])
#Using a Vincent KeyedList here
vis.data['table'] = data
```

You'll notice two interesting pieces here: `Data` , and the `KeyedList`

1.3.2 Keyed List

The Vega specification consists of high level attributes such as scales, marks, axes, legends, etc, each with an array containing any number of individual marks, scales, etc. It's useful to be able to index these arrays by name, so Vincent has introduced a Python List that can also be indexed by a name parameter.

So, for example, if we want to introduce a new color scale:

```
scale = vincent.Scale(name='color', type='ordinal',
                      domain=DataRef(data='table', field='data.col'),
                      range='category20')
```

We can add it to the scales list and index it by name:

```
vis.scales['color'] = scale
```

Note that the name must match the index you inserting:

```
vis.scales['newscale'] = scale

ValidationError: key must be equal to 'name' attribute
```

1.3.3 Data Importing

The Vincent Data class has a number of conveniences for import Python data types.

First, `from_iter`, which will take lists, tuples, or key/value dicts:

```
list_dat = [10, 20, 30, 40, 50]
data = vincent.Data.from_iter(list_dat)

data.values
[{'col': 'data', 'idx': 0, 'val': 10},
 {'col': 'data', 'idx': 1, 'val': 20},
 {'col': 'data', 'idx': 2, 'val': 30},
 {'col': 'data', 'idx': 3, 'val': 40},
 {'col': 'data', 'idx': 4, 'val': 50}]

dict_dat = {'x': 10, 'y': 20, 'z': 30}
data = vincent.Data.from_iter(dict_dat)

data.values
[{'col': 'data', 'idx': 'y', 'val': 20},
 {'col': 'data', 'idx': 'x', 'val': 10},
 {'col': 'data', 'idx': 'z', 'val': 30}]
```

There's also a `from_mult_iters` convenience method, in which you must provide a common index:

```
x = [0, 1, 2, 3, 4]
y = [10, 20, 30, 40, 50]
z = [70, 80, 90, 100, 110]
data = vincent.Data.from_mult_iters(index=x, values1=y, values2=z, idx='index')

data.values
[{'col': 'values1', 'idx': 0, 'val': 10},
 {'col': 'values1', 'idx': 1, 'val': 20},
 {'col': 'values1', 'idx': 2, 'val': 30},
 {'col': 'values1', 'idx': 3, 'val': 40},
 {'col': 'values1', 'idx': 4, 'val': 50},
 {'col': 'values2', 'idx': 0, 'val': 70},
 {'col': 'values2', 'idx': 1, 'val': 80},
 {'col': 'values2', 'idx': 2, 'val': 90},
 {'col': 'values2', 'idx': 3, 'val': 100},
 {'col': 'values2', 'idx': 4, 'val': 110}]
```

This indexing structure allows for faceting on `col` or `idx` for charts like stacked or grouped bars.

The best way to get data into Vincent is with the Pandas Series and DataFrame. These provide built-in indexing and index sorting, and will generally make your charts appear nicer. We'll start with Series:

```
series = pd.Series([10, 20, 30, 40, 50])
data = vincent.Data.from_pandas(series)

data.values
[{'col': 'data', 'idx': 0, 'val': 10},
 {'col': 'data', 'idx': 1, 'val': 20},
 {'col': 'data', 'idx': 2, 'val': 30},
 {'col': 'data', 'idx': 3, 'val': 40},
 {'col': 'data', 'idx': 4, 'val': 50}]
```

If the series has a name, this will be your `col` value:

```
series.name = 'metric'
data = vincent.Data.from_pandas(series)

data.values
[{'col': 'metric', 'idx': 0, 'val': 10},
 {'col': 'metric', 'idx': 1, 'val': 20},
 {'col': 'metric', 'idx': 2, 'val': 30},
 {'col': 'metric', 'idx': 3, 'val': 40},
 {'col': 'metric', 'idx': 4, 'val': 50}]
```

DataFrames are just as simple:

```
farm_1 = {'apples': 10, 'berries': 32, 'squash': 21}
farm_2 = {'apples': 15, 'berries': 43, 'squash': 17}
farm_3 = {'apples': 6, 'berries': 24, 'squash': 22}

farm_data = [farm_1, farm_2, farm_3]
farm_index = ['Farm 1', 'Farm 2', 'Farm 3']
df = pd.DataFrame(farm_data, index=farm_index)
data = vincent.Data.from_pandas(df)

data.values
[{'col': 'apples', 'idx': 'Farm 1', 'val': 10},
 {'col': 'berries', 'idx': 'Farm 1', 'val': 32},
 {'col': 'squash', 'idx': 'Farm 1', 'val': 21},
 {'col': 'apples', 'idx': 'Farm 2', 'val': 15},
 {'col': 'berries', 'idx': 'Farm 2', 'val': 43},
 {'col': 'squash', 'idx': 'Farm 2', 'val': 17},
 {'col': 'apples', 'idx': 'Farm 3', 'val': 6},
 {'col': 'berries', 'idx': 'Farm 3', 'val': 24},
 {'col': 'squash', 'idx': 'Farm 3', 'val': 22}]
```

You can also key on a column, rather than the index:

```
data = vincent.Data.from_pandas(df, key_on='apples')

data.values
[{'col': 'apples', 'idx': 10, 'val': 10},
 {'col': 'berries', 'idx': 10, 'val': 32},
 {'col': 'squash', 'idx': 10, 'val': 21},
 {'col': 'apples', 'idx': 15, 'val': 15},
 {'col': 'berries', 'idx': 15, 'val': 43},
 {'col': 'squash', 'idx': 15, 'val': 17},
 {'col': 'apples', 'idx': 6, 'val': 6},
 {'col': 'berries', 'idx': 6, 'val': 24},
 {'col': 'squash', 'idx': 6, 'val': 22}]
```

Finally, if you turn on `grouped`, it will add an additional iterative key for Vega grouping that groups on the column values:

```
data = vincent.Data.from_pandas(df, grouped=True)]

data.values
[{'col': 'apples', 'group': 0, 'idx': 'Farm 1', 'val': 10},
 {'col': 'berries', 'group': 1, 'idx': 'Farm 1', 'val': 32},
 {'col': 'squash', 'group': 2, 'idx': 'Farm 1', 'val': 21},
 {'col': 'apples', 'group': 0, 'idx': 'Farm 2', 'val': 15},
 {'col': 'berries', 'group': 1, 'idx': 'Farm 2', 'val': 43},
 {'col': 'squash', 'group': 2, 'idx': 'Farm 2', 'val': 17},
```

```
 {'col': 'apples', 'group': 0, 'idx': 'Farm 3', 'val': 6},
 {'col': 'berries', 'group': 1, 'idx': 'Farm 3', 'val': 24},
 {'col': 'squash', 'group': 2, 'idx': 'Farm 3', 'val': 22}]
```

1.4 Vega API Reference

The full Vega specification is exposed through a set of object-relational JSON classes. Vincent also pr

1.4.1 Visualization

Field properties for `Visualization`:

1.4.2 Data

Field properties for `Data`:

1.4.3 Scale

Field properties for `Scale`:

1.4.4 DataRef

Field properties for `DataRef`:

ValueRef

Field properties for `ValueRef`:

1.4.5 Mark

Field properties for `Mark`:

1.4.6 Mark Properties

Field properties for `MarkProperties`:

1.4.7 PropertySet

Field properties for `PropertySet`:

1.4.8 Transforms

Field properties for `Transform`:

1.5 Development

Want to help make Vincent better? Here's how to get started:

First, fork Vincent on Github. Then clone your fork into a local folder:

```
$git clone https://github.com/your_username/vincent
```

Set up your virtualenv:

```
$virtualenv .env
```

Pip install the dependencies:

```
$pip install -r requirements.txt
```

Set up the package for development:

```
$python setup.py develop
```

Now you're set. Here are some area where Vincent could use contribution:

1.5.1 Charts.py

Go take a look at charts.py within the main Vincent package and you'll see that we're using Vincent to build Vincent! This module is the home for convenience chart builds, such that we can do things like `bar = vincent.Bar([10, 20, 30])`. This package still needs maps, pie charts, donut charts, treemaps, faceted charts, etc. If you use Vincent to build a new chart, make a pull request for this module. Make sure you add a test with valid grammar to `test_charts.py`

1.5.2 Transforms.py

There are still a number of Vega Transform types that have yet to be implemented. Start adding them to `transforms.py` and make a pull request.

1.5.3 Testing

Vincent uses the Nose library for testing, and aims for reasonable test coverage. Take a look at `test_charts.py` and `test_vega.py` to get an idea of what our tests look like, and please add coverage for anything you add.

- search