Candela

Kitware, Inc.

Jul 12, 2018

Using Candela

Candela is an open-source suite of interoperable web visualization components for Kitware's Resonant platform. Candela focuses on making scalable, rich visualizations available with a normalized API for use in real-world data science applications. Integrated components include:

- *LineUp* component: LineUp dynamic ranking by the Harvard University Visual Computing Group and the Caleydo project.
- *UpSet* component: UpSet set visualization by the Harvard University Visual Computing Group and the Caleydo project.
- *OnSet* component: OnSet set visualization by the Georgia Institute of Technology Information Interfaces Group.
- Vega visualizations by the University of Washington Interactive Data Lab. Example component: *ScatterPlot*.
- GeoJS geospatial visualizations by Kitware's Resonant platform. Example component: *GeoDots*.

CHAPTER 1

Getting started

1.1 Quick start - JavaScript

1. Enter the following in a text file named `index.html`:

```
<body>
<div id="vis"></div>
<script src="//unpkg.com/candela/dist/candela.min.js"></script>
<script>

  var data = [
    {x: 1, y: 3},
    {x: 2, y: 4},
    {x: 2, y: 3},
    {x: 0, y: 1}
  ];

  var el = document.getElementById('vis');
  var vis = new candela.components.ScatterPlot(el, {
    data: data,
    x: 'x',
    y: 'y'
  });
  vis.render();

</script>
</body>
```

2. Open `index.html` in your browser to display the resulting visualization.

1.2 Quick start - Python

1. Make sure you have Python 2.7 and pip installed (on Linux and OS X systems, your local package manager should do the trick; for Windows, see here).
2. Open a shell (e.g. Terminal on OS X; Bash on Linux; or Command Prompt on Windows) and issue this command to install the Candela package and the Requests library for obtaining sample data from the web:

```
pip install pycandela requests
```

   (On UNIX systems you may need to do this as root, or with `sudo`.)

3. Issue this command to start Jupyter notebook server in your browser:

```
jupyter-notebook
```

4. Create a notebook from the New menu and enter the following in a cell, followed by Shift-Enter to execute the cell and display the visualization:

```
import requests
data = requests.get(
    'https://raw.githubusercontent.com/vega/vega-datasets/gh-pages/data/iris.json'
).json()

import pycandela
pycandela.components.ScatterPlot(
    data=data, color='species', x='sepalLength', y='sepalWidth')
```

1.3 Quick start - R

1. Download and install RStudio.
2. Run the following commands to install Candela:

```
install.packages('devtools')
devtools::install_github('Kitware/candela', subdir='R/candela')
```

3. Issue these commands to display a scatter plot of the `mtcars` dataset:

```
library(candela)
candela('ScatterPlot', data=mtcars, x='mpg', y='wt', color='disp')
```

1.3.1 Installation

There are two ways to install Candela: from standard package repositories systems such as npm and the Python Package Index (PyPI). Installing from a package repository is simpler, but limited to public release versions; installing from source is slightly more complicated but allows you to run cutting-edge development versions.

Installing from package management systems

JavaScript

To install the Candela JavaScript library to `node_modules/candela` in your current directory, run:

```
npm install candela
```

To install the Candela JavaScript library as a dependency of your web application and add it to your `package.json` file, run:

```
npm install --save candela
```

A self-contained JavaScript bundle for Candela will be found at `node_modules/candela/dist/candela[.min].js`.

Using Webpack

If your project uses a Webpack build process, you can use Candela's bundled Webpack helper function to include Candela easily in your project without having to use the fullsize bundle file. The idea is to directly include Candela source files as needed in your project, relying on the Webpack helper to arrange for the proper loaders and other configuration options to be used. For example, without Webpack, your source file might include lines like

```
var candela = require('candela/dist/candela.min.js');
var ScatterPlot = candela.components.ScatterPlot;
```

This will result in your application loading the entire Candela bundle at runtime, which may not be optimal if you just wish to use the `ScaterPlot` component. Instead, using Webpack, you could cast this code as follows:

```
var ScatterPlot = require('candela/components/ScatterPlot');
```

To make sure that your build process uses the correct loaders for this file, you should make sure to use the Candela webpack helper function in your project's Webpack configuration:

```
var candelaWebpack = require('candela/webpack');

module.exports = candelaWebpack({
  // Your original webpack configuration goes here
});
```

This approach lets you keep your code more concise and meaningful, while also avoiding unnecessarily large application bundles.

Python

The latest release version of the Python bindings for Candela can always be found in the Python Package Index. The easiest way to install Candela is via Pip, a package manager for Python.

1. Install software dependencies

Install the following software:

- Python 2.7
- Pip

On Linux and OS X computers, your local package manager should be sufficient for installing these. On Windows, please consult this guide for advice about Python and Pip.

2. Install the Candela Python package

Use this command in a shell to install the Candela package and its dependencies:

```
pip install pycandela
```

You may need to run this command as the superuser, using `sudo` or similar.

Building and installing from source

Before any of these source installations, you will need to issue this Git command to clone the Candela repository:

```
git clone git://github.com/Kitware/candela.git
```

This will create a directory named `candela` containing the source code. Use `cd` to move into this directory:

```
cd candela
```

JavaScript

Candela is developed on GitHub. If you wish to contribute code, or simply want the very latest development version, you can download, build, and install from GitHub, following these steps:

1. Install software dependencies

To build Candela from source, you will need to install the following software:

- Git
- Node.js
- npm
- cairo (`brew install cairo` on macOS)

2. Install Node dependencies

Issue this command to install the necessary Node dependencies via the Node Package Manager (NPM):

```
npm install
```

The packages will be installed to a directory named `node_modules`.

3. Begin the build process

Issue this command to kick off the build process:

```
npm run build
```

The output will create a built Candela package in `build/candela/candela.js`.

Watch the output for any errors. In most cases, an error will halt the process, displaying a message to indicate what happened. If you need any help deciphering any such errors, drop us a note on GitHub issues or on Gitter chat.

4. View the examples

Candela contains several examples used for testing, which also may be useful for learning the variety of visualizations available in Candela. To build the examples, run:

```
npm run build:examples
```

To view the examples, run:

```
npm run examples
```

5. Run the test suites

Candela comes with a battery of tests. To run these, you can invoke the test task as follows:

```
npm run test:all
```

This runs both the unit and image tests. Each test suite can be run on its own, with:

```
npm run test:unit
```

and:

```
npm run test:image
```

Each of these produces a summary report on the command line.

6. Build the documentation

Candela uses Sphinx documentation hosted on ReadTheDocs. To build the documentation locally, first install the required Python dependencies:

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

When the installation completes, issue this command:

```
npm run docs
```

The documentation will be hosted at http://localhost:3000.

Python

1. Install software dependencies

To use Candela from Python you will need Python 2.7 and `pip`.

2. Install the library locally

```
pip install -e .
```

3. Test the installation

Issue this command to start Jupyter notebook server in your browser:

```
jupyter-notebook
```

Create a notebook from the New menu and enter the following in a cell, followed by Shift-Enter to execute the cell and display the visualization:

```
import requests
data = requests.get(
    'https://raw.githubusercontent.com/vega/vega-datasets/gh-pages/data/iris.json'
).json()

import pycandela
pycandela.components.ScatterPlot(
    data=data, color='species', x='sepalLength', y='sepalWidth')
```

R - using `install_github` or Git checkout

This procedure will install Candela either directly from GitHub or from a local Git checkout of Candela.

1. Install R **, and optionally** RStudio

2. Install the Candela package

To install directly from GitHub:

```
install.packages('devtools')
devtools::install_github('Kitware/candela', subdir='R/candela', dependencies = TRUE)
```

To install from a Git checkout, set your working directory to the Git checkout then install and check the installation. `check()` will run tests and perform other package checks.

```
setwd('/path/to/candela/R/candela')
install.packages('devtools')
devtools::install(dependencies = TRUE)
devtools::check()
```

3. Test the installation

The following will create a scatter plot of the `mtcars` dataset and save it to `out.html`:

```
library(candela)
w <- candela('ScatterPlot', data=mtcars, x='mpg', y='wt', color='disp')
htmlwidgets::saveWidget(w, 'out.html')
```

From RStudio, the visualization will appear in the application when you refer to a visualization without assigning it to a variable:

```
w
```

Note: `saveWidget` requires an installation of Pandoc when run outside of RStudio. See the installation instructions to install.

1.3.2 Versioning

Candela uses semantic versioning for its version numbers, meaning that each release's version number establishes a promise about the levels of functionality and backwards compatibility present in that release. Candela's version numbers come in two forms: *x.y* and *x.y.z*. *x* is a *major version number*, *y* is a *minor version number*, and *z* is a *patch level*.

Following the semantic versioning approach, major versions represent a stable API for the software as a whole. If the major version number is incremented, it means you can expect a discontinuity in backwards compatibility. That is to say, a setup that works for, e.g., version 1.3 will work for versions 1.4, 1.5, and 1.10, but should not be expected to work with version 2.0.

The minor versions indicate new features or functionality added to the previous version. So, version 1.1 can be expected to contain some feature not found in version 1.0, but backwards compatibility is ensured.

The patch level is incremented when a bug fix or other correction to the software occurs.

Major version 0 is special: essentially, there are no guarantees about compatibility in the 0.*y* series. The stability of APIs and behaviors begins with version 1.0.

In addition to the standard semantic versioning practices, Candela also tags the current version number with "dev" in the Git repository, resulting in version numbers like "1.1dev" for the Candela package that is built from source. The release protocol deletes this tag from the version number before uploading a package to the Python Package Index.

1.3.3 BarChart

A bar chart. The **x** field should contain a distinct value for each bar, while the **y** field will correspond to the height of each bar. The **color** field may be used to color each bar. In the case where there are multiple records for a single **x** value, **aggregate** may be used to combine values into a single bar.

This component can be found in the `candela/plugins/vega` plugin.

Example

JavaScript

```
<body>
<script src="//unpkg.com/candela/dist/candela.min.js"></script>
<script>
  var el = document.createElement('div')
  document.body.appendChild(el);

  var data = [];
  for (var d = 0; d < 10; d += 1) {
    data.push({
      a: d,
      b: d
    });
  }

  var vis = new candela.components.BarChart(el, {
    data: data,
    x: 'a',
    y: 'b'
  });
  vis.render();
</script>
</body>
```

Python

```
import pycandela

data = [{'a': d, 'b': d} for d in range(10)]

pycandela.components.BarChart(data=data, x='a', y='b')
```

R

```
library(candela)

candela('BarChart', data=mtcars, x='mpg', y='wt', color='disp')
```

Options

data (*Table*) The data table.

x (String) The x axis (bar position) field. Must contain numeric data. See *Axis scales*.

xType (String) The data type for the `x` field. The default is `"nominal"`.

y (String) The y axis (bar height) field. Must contain numeric data. See *Axis scales*.

yType (String) The data type for the `y` field. The default is `"quantitative"`.

color (String) The field used to color the bars.

colorType (String) The data type for the `color` field. The default is `"nominal"`.

aggregate (String) The aggregation mode for `y` values when the `x` value is the same in multiple records. The default is `"sum"`.

width (Number) Width of the chart in pixels. See *Sizing*.

height (Number) Height of the chart in pixels. See *Sizing*.

renderer (String) Whether to render in `"svg"` or `"canvas"` mode (default `"canvas"`).

1.3.4 BoxPlot

A boxplot. The visualization takes a set of measures (**fields**) and produces a boxplot of each one. The optional **group** field will partition the data into groups with matching value and make a boxplot (or set of boxplots) for each group.

This component can be found in the `candela/plugins/vega` plugin.

Example

JavaScript

```
<body>
<script src="//unpkg.com/candela/dist/candela.min.js"></script>
<script>
  var el = document.createElement('div')
  document.body.appendChild(el);

  var data = [];
  for (var d = 0; d < 10; d += 1) {
    data.push({
      a: d,
      b: d/2 + 7
    });
  }

  var vis = new candela.components.BoxPlot(el, {
    data: data,
    fields: ['a', 'b']
  });
  vis.render();
</script>
</body>
```

Python

```
import pycandela

data = [{'a': d, 'b': d/2 + 7} for d in range(10)]

pycandela.components.BoxPlot(data=data, fields=['a', 'b'])
```

R

```
library(candela)

candela('BoxPlot', data=mtcars, fields=c('mpg', 'wt', 'disp'))
```

Options

data (*Table*) The data table.

fields (Array of String) The fields to use as measurements. The visualization will produce a boxplot for each field. Must contain numeric or temporal data. See *Axis scales*. Axis type will be chosen by the inferred value of the first field in the array.

x (String) The optional field to group by. Defaults to all records being placed in a single group. See *Axis scales*.

xType (String) The data type for the `x` field. The default is `"nominal"`.

color (String) The field used to color the box plots.

colorType (String) The data type for the `color` field. The default is `"nominal"`.

width (Number) Width of the chart in pixels. See *Sizing*.

height (Number) Height of the chart in pixels. See *Sizing*.

renderer (String) Whether to render in `"svg"` or `"canvas"` mode (default `"canvas"`).

1.3.5 GanttChart

A Gantt chart. The **data** table must contain two numeric fields, **start** and **end**, which specify the start and end of horizontal bars. A **label** field can specify the name of each item.

This component can be found in the `candela/plugins/vega` plugin.

Example

JavaScript

```
<body>
<script src="//unpkg.com/candela/dist/candela.min.js"></script>
<script>
  var el = document.createElement('div')
  document.body.appendChild(el);

  var data = [
    {name: 'Do this', level: 1, start: 0, end: 5},
    {name: 'This part 1', level: 2, start: 0, end: 3},
    {name: 'This part 2', level: 2, start: 3, end: 5},
    {name: 'Then that', level: 1, start: 5, end: 15},
```

(continues on next page)

(continued from previous page)

```
    {name: 'That part 1', level: 2, start: 5, end: 10},
    {name: 'That part 2', level: 2, start: 10, end: 15}
  ];
  var vis = new candela.components.GanttChart(el, {
    data: data, label: 'name',
    start: 'start', end: 'end', level: 'level',
    width: 700, height: 200
  });
  vis.render();
</script>
</body>
```

Python

```
import pycandela

data = [
    dict(name='Do this', level=1, start=0, end=5),
    dict(name='This part 1', level=2, start=0, end=3),
    dict(name='This part 2', level=2, start=3, end=5),
    dict(name='Then that', level=1, start=5, end=15),
    dict(name='That part 1', level=2, start=5, end=10),
    dict(name='That part 2', level=2, start=10, end=15)
];
pycandela.components.GanttChart(
    data=data, label='name',
    start='start', end='end', level='level',
    width=700, height=200
)
```

R

```
library(candela)

data <- list(
    list(name='Do this', level=1, start=0, end=5),
    list(name='This part 1', level=2, start=0, end=3),
    list(name='This part 2', level=2, start=3, end=5),
    list(name='Then that', level=1, start=5, end=15),
    list(name='That part 1', level=2, start=5, end=10),
    list(name='That part 2', level=2, start=10, end=15))

candela('GanttChart',
    data=data, label='name',
    start='start', end='end', level='level',
    width=700, height=200)
```

Options

data (*Table*) The data table.

label (String) The field used to label each task.

start (String) The field representing the start of each task. Must be numeric.

end (String) The field representing the end of each task. Must be numeric.

level (String) The string used as the level for hierarchical items. Currently supports two unique values, the first value encountered will be level 1 which is rendered more prominently, and the second value will be level 2.

type (String) The data type for the `start` and `end` fields. The default is `"quantitative"`.

tickCount (String) The suggested number of tick marks to place along the x axis.

axisTitle (String) The title of the x axis.

width (Number) Width of the chart in pixels. See *Sizing*.

height (Number) Height of the chart in pixels. See *Sizing*.

renderer (String) Whether to render in `"svg"` or `"canvas"` mode (default `"canvas"`).

1.3.6 Geo

A geospatial chart using GeoJS.

This component can be found in the `candela/plugins/geojs` plugin.

Example

JavaScript

```
<body>
<script src="//unpkg.com/candela/dist/candela.min.js"></script>
<script>
  var el = document.createElement('div')
  el.style.width = '500px';
  el.style.height = '500px';
  document.body.appendChild(el);

  var data = [
    {lat: 41.702, lng: -87.644},
    {lat: 41.617, lng: -87.693},
    {lat: 41.715, lng: -87.712}
  ];
  var vis = new candela.components.Geo(el, {
    map: {
      zoom: 10,
      center: {
        x: -87.6194,
        y: 41.867516
      }
    },
    layers: [
      {
        type: 'osm'
      },
      {
        type: 'feature',
        features: [
          {
            type: 'point',
            data: data,
            x: 'lng',
            y: 'lat'
```

(continues on next page)

(continued from previous page)

```
          }
        ]
      }
    ]
  });
  vis.render();
</script>
</body>
```

Python

```
import pycandela

data = [
    dict(lat=41.702, lng=-87.644),
    dict(lat=41.617, lng=-87.693),
    dict(lat=41.715, lng=-87.712)
]

pycandela.components.Geo(
    map=dict(
        zoom=10,
        center=dict(x=-87.6194, y=41.867516)
    ),
    layers=[
        dict(type='osm'),
        dict(
            type='feature',
            features=[
                dict(type='point', data=data, x='lng', y='lat')
            ]
        )
    ]
)
```

R

```
library(candela)

data = list(
  list(lat=41.702, lng=-87.644),
  list(lat=41.617, lng=-87.693),
  list(lat=41.715, lng=-87.712))

candela('Geo',
  map=list(
    zoom=10,
    center=list(x=-87.6194, y=41.867516)
  ),
  layers=list(
    list(type='osm'),
    list(
      type='feature',
      features=list(
        list(type='point', data=data, x='lng', y='lat')
      )
    )
```

(continues on next page)

(continued from previous page)

```
  )
)
```

Options

map (Object) Key-value pairs describing GeoJS map options.

layers (Array of *Layer*) The layers of the map.

Layer specification

A layer contains key-value pairs describing GeoJS layer options. These options are passed through to GeoJS, with the exception of the `"features"` option for a layer with `type` set to `"feature"`. In this case, the `"features"` option is an array of *Feature specifications*.

Feature specification

Each feature is an object with the following properties:

name (String) The name of the feature.

type (String) The feature type (currently supported: `"point"`).

data (*Table*) The data table.

x (String) The field to use for the feature's `x` coordinate.

y (String) The field to use for the feature's `y` coordinate.

1.3.7 GeoDots

A geospatial view with locations marked by dots, using GeoJS. The **latitude** and **longitude** fields should contain lat/long values for each location in the data.

This component can be found in the `candela/plugins/geojs` plugin.

Example

JavaScript

```
<body>
<script src="//unpkg.com/candela/dist/candela.min.js"></script>
<script>
 var el = document.createElement('div')
 el.style.width = '500px';
 el.style.height = '500px';
 document.body.appendChild(el);

 var data = [
   {lat: 41.702, lng: -87.644, a: 5},
   {lat: 41.617, lng: -87.693, a: 15},
   {lat: 41.715, lng: -87.712, a: 25}
 ];
```

(continues on next page)

(continued from previous page)

```
 var vis = new candela.components.GeoDots(el, {
   zoom: 10,
   center: {
     longitude: -87.6194,
     latitude: 41.867516
   },
   data: data,
   latitude: 'lat',
   longitude: 'lng',
   size: 'a',
   color: 'a'
 });
 vis.render();
</script>
</body>
```

Python

```
import pycandela

data = [
    dict(lat=41.702, lng=-87.644, a=5),
    dict(lat=41.617, lng=-87.693, a=15),
    dict(lat=41.715, lng=-87.712, a=25)
]

pycandela.components.GeoDots(
    zoom=10,
    center=dict(longitude=-87.6194, latitude=41.867516),
    data=data,
    latitude='lat',
    longitude='lng',
    size='a',
    color='a'
)
```

R

```
library(candela)

data = list(
  list(lat=41.702, lng=-87.644, a=5),
  list(lat=41.617, lng=-87.693, a=15),
  list(lat=41.715, lng=-87.712, a=25))

candela('GeoDots',
  zoom=10,
  center=list(longitude=-87.6194, latitude=41.867516),
  data=data,
  latitude='lat',
  longitude='lng',
  size='a',
  color='a')
```

Options

data (*Table*) The data table.

longitude (String) The longitude field.

latitude (String) The latitude field.

color (String) The field to color the points by.

size (String) The field to size the points by. The field must contain numeric values.

zoom (Integer) The initial zoom level.

center (Object) An object with `longitude` and `latitude` properties specifying the initial center of the map.

tileUrl (String) A tile URL template (see GeoJS OSM layer options). Set to `null` to disable the OSM layer completely.

1.3.8 GLO (Graph-Level Operations)

A visualization framework that treats the data like nodes of a graph, and uses positioning and visual commands to arrange them into different formats to implement different visualizations.

The **nodes** table contains a list of objects, each with an `id` field containing a unique identifier, along with any other data attributes needed. The **edges** table contains `source` and `target` fields referring to `id` values from the **nodes** table, an optional `type` field reading either `Undirected` or `Directed`, an `id` field identifying each edge, and an optional `weight` value. **width** and **height** control the size of the canvas used for rendering the visualization.

This component can be found in the `candela/plugins/glo` plugin.

Example

JavaScript

```
<body>
<script src="//unpkg.com/candela/dist/candela.min.js"></script>
<script>
  var el = document.createElement('div')
  el.setAttribute('width', 700);
  el.setAttribute('width', 700);

  document.body.appendChild(el);

  var alphabet = 'abcdefghijklmnopqrstuvwxyz';
  var vowels = 'aeiou'.split('');

  var nodes = [];
  for (var i = 0; i < 26; i++) {
    var letter = {
      id: i,
      label: alphabet[i],
      vowel: vowels.indexOf(alphabet[i]) > 0 ? 'vowel' : 'consonant'
    };

    for (var j = 0; j < 26; j++) {
      letter[alphabet[j]] = Math.abs(j - i);
    }
```

(continues on next page)

(continued from previous page)

```
    nodes.push(letter);
  }

  var edges = [];
  var counter = 0;
  for (var i = 0; i < 26; i++) {
    for (var j = i + 1; j < 26; j++) {
      if (nodes[i][alphabet[j]] > 20) {
        edges.push({
          source: i,
          target: j,
          type: 'Undirected',
          id: counter++,
          weight: 1
        });
      }
    }
  }

  var vis = new candela.components.Glo(el, {
    nodes: nodes,
    edges: edges,
    width: 700,
    height: 200
  });

  vis.render();

  vis.distributeNodes('x');
  vis.colorNodesDiscrete('vowel');
  vis.curvedEdges();
</script>
</body>
```

Python

```
import pycandela

data = [
  {'id': 0, 'label': 'A', 'class': 0},
  {'id': 1, 'label': 'B', 'class': 1},
  {'id': 2, 'label': 'C', 'class': 1}
]

edges = [
  {'id': 0, 'source': 0, 'target': 1},
  {'id': 1, 'source': 0, 'target': 2},
  {'id': 2, 'source': 2, 'target': 1}
]

glo = pycandela.components.Glo(nodes=nodes, edges=edges)
glo.render()
glo.distributeNodes('x');
glo.colorNodesDiscrete('class');
glo.curvedEdges();
```

R

```r
library(candela)

id = c(0, 1, 2)
label = c('A', 'B', 'C')
class = c(0, 1, 1)
nodes = data.frame(id, label, class)

source = c(0, 0, 2)
target = c(1, 2, 1)
edges = data.frame(id, source, target)

glo = candela('SimilarityGraph', nodes=nodes, edges=edges)
glo.render()
glo.distributeNodes('x')
glo.colorNodesDiscrete('class')
glo.curvedEdges()
```

Options

nodes (*Table*) The node table.

edges (*Table*) The edge table.

width (number) The width of the drawing area.

height (number) The height of the drawing area.

Methods

`colorNodesDiscrete` (*field*)

Arguments

- **`field`** (*`string`*) – The field to color by

Use a categorical colormap to color the nodes by the values in `field`.

`colorNodesContinuous` (*field*)

Arguments

- **`field`** (*`string`*) – The field to color by

Use a continuous colormap to color the nodes by the values in `field`.

`colorNodesDefault` ()

Revert the node color to the default state (no colormap).

`sizeNodes` (*field*)

Arguments

- **`field`** (*`string`*) – The field to size by

Size the nodes according to the values in `field`.

`sizeNodesDefault` ()

Revert the node size to the default state (constant sized).

`distributeNodes` (*axis*[, *attr*])

Arguments

- **string** (*attr*) – The axis on which to distribute the nodes
- **string** – The field to use for grouping the nodes

Position the nodes evenly along `axis`, which must be one of `"x"`, `"y"`, `"rho"` (radial axis), or `"theta"` (angle axis). If `attr` is given, the nodes will be partitioned and grouped according to it.

positionNodes(*axis*, *value*)

Arguments

- **axis** (*string*) – The axis on which to distribute the nodes
- **value** (*string|number*) – The field to draw position data from, or a constant

Position the nodes along `axis` (see *distributeNodes()*) according to the data in `value`. If `value` is a string, it refers to a column of data frome the **nodes** table; if it is a number, then all nodes will be positioned at that location.

forceDirected()
Apply a force-directed positioning algorithm to the nodes.

showEdges()
Display all edges between nodes.

hideEdges()
Hide all edges between nodes.

fadeEdges()
Render edges using a transparent gray color.

solidEdges()
Render edges using black.

incidentEdges()
Only render edges incident on a node when the mouse pointer is hovering over that node.

curvedEdges()
Render edges using curved lines.

straightEdges()
Render edges using straight lines.

1.3.9 Histogram

A histogram. The **bin** option specifies which field to summarize. By default, each record in the **data** table will contribute 1 to the bin's total. Specifying an **aggregate** field will instead add up that field's value for the each bin.

This component can be found in the `candela/plugins/vega` plugin.

Example

JavaScript

```
<body>
<script src="//unpkg.com/candela/dist/candela.min.js"></script>
<script>
  var el = document.createElement('div')
  document.body.appendChild(el);
```

(continues on next page)

(continued from previous page)

```
  var data = [];
  for (var d = 0; d < 1000; d += 1) {
    data.push({
      a: Math.sqrt(-2*Math.log(Math.random()))*Math.cos(2*Math.PI*Math.random())
    });
  }

  var vis = new candela.components.Histogram(el, {
    data: data,
    x: 'a',
    width: 700,
    height: 400
  });
  vis.render();
</script>
</body>
```

Python

```
import pycandela
from random import normalvariate as nv

data = [{'a': nv(0, 1)} for d in range(1000)]

pycandela.components.Histogram(data=data, x='a', width=700, height=400)
```

R

```
library(candela)

candela('Histogram', data=mtcars, x='mpg')
```

Options

data (*Table*) The data table.

x (String) The x axis field, which is binned into a histogram.

xType (String) The data type for the `x` field. The default is `"nominal"`.

aggregate (String) The aggregation mode for `y` values within each histogram bin. The default is `"count"`, which does not use the `y` values but will count the number of records in the bin.

y (String) The y axis field, which is used to determine the height of the histogram bar when `aggregate` is not set to `"count"`.

yType (String) The data type for the `y` field. The default is `"quantitative"`.

color (String) The field used to color the bars.

colorType (String) The data type for the `color` field. The default is `"nominal"`.

width (Number) Width of the chart in pixels. See *Sizing*.

height (Number) Height of the chart in pixels. See *Sizing*.

renderer (String) Whether to render in `"svg"` or `"canvas"` mode (default `"canvas"`).

1.3.10 LineChart

A line chart. The chart plots a line for the **y** field against a single **x** field, optionally splitting into multiple lines using the **series** field.

This component can be found in the `candela/plugins/vega` plugin.

Example

JavaScript

```
<body>
<script src="//unpkg.com/candela/dist/candela.min.js"></script>
<script>
  var el = document.createElement('div')
  document.body.appendChild(el);

  var data = [];
  for (var d = 0; d < 10; d += 1) {
    data.push({
      a: d,
      b: d
    });
  }

  var vis = new candela.components.LineChart(el, {
    data: data,
    x: 'a',
    y: 'b',
    width: 700,
    height: 400
  });
  vis.render();
</script>
</body>
```

Python

```
import pycandela

data = [{'a': d, 'b': d} for d in range(10)]

pycandela.components.LineChart(
    data=data, x='a', y='b', width=700, height=400)
```

R

```
library(candela)

candela('LineChart', data=mtcars, x='mpg', y='wt', color='disp')
```

Options

data (*Table*) The data table.

x (String) The x axis field.

xType (String) The data type for the `x` field. The default is `"quantitative"`.

y (String) The y axis field.

yType (String) The data type for the `y` field. The default is `"quantitative"`.

series (String) The optional field used to separate the data into multiple lines.

seriesType (String) The data type for the `series` field. The default is `"nominal"`.

colorSeries (Boolean) Whether to color the different series and show a legend. The default is `true`.

showPoints (Boolean) Whether to overlay points on the lines. The default is `false`.

width (Number) Width of the chart in pixels. See *Sizing*.

height (Number) Height of the chart in pixels. See *Sizing*.

renderer (String) Whether to render in `"svg"` or `"canvas"` mode (default `"canvas"`).

1.3.11 LineUp

A LineUp table ranking visualization.

This component can be found in the `candela/plugins/lineup` plugin.

Example

JavaScript

```
<body>
<script src="//unpkg.com/candela/dist/candela.min.js"></script>
<script>
  var el = document.createElement('div')
  document.body.appendChild(el);

  var data = [];
  for (var d = 0; d < 10; d += 1) {
    data.push({
      a: d,
      b: 10 - d,
      name: d
    });
  }

  var vis = new candela.components.LineUp(el, {
    data: data,
    fields: ['a', 'b']
  });
  vis.render();
</script>
</body>
```

Python

```
import pycandela

data = [{'a': d, 'b': 10 - d, 'name': d} for d in range(10)]

pycandela.components.LineUp(data=data, fields=['a', 'b'])
```

R

```
library(candela)

candela('LineUp', data=mtcars, fields=c('_row', 'mpg', 'wt', 'disp'))
```

Options

data (*Table*) The data table.

fields (Array of String) A list of fields that will be shown on the LineUp view. The list determines the order of the fields. If not supplied, all fields from the data are shown.

stacked (Boolean) Whether to display grouped measures as a stacked bar (default false).

histograms (Boolean) Whether to display histograms in the headers of each measure (default true).

animation (Boolean) Whether to animate transitions when the scoring metric changes (default true).

1.3.12 OnSet

An OnSet set visualization.

OnSet interprets binary columns (i.e. columns with a literal `"0"` or `"1"`, `"true"` or `"false"`, `"yes"` or `"no"` in every row) as sets. Any field in the **sets** option will be interpreted in this way. Since most data is not arranged in binary columns, the visualization also supports arbitrary categorical fields with the **fields** option. Each field specified in this list will first be preprocessed into a collection sets, one for each field value, with the name `"<fieldName> <value>"`.

For example, suppose the data table is:

```
[
  {"id": "n1", "f1": 1, "f2": "x"},
  {"id": "n2", "f1": 0, "f2": "x"},
  {"id": "n3", "f1": 0, "f2": "y"}
]
```

You could create an OnSet visualization with the following options:

```
new OnSet({
  data: data,
  id: 'id',
  sets: ['f1'],
  fields: ['f2']
});
```

This would preprocess the `f2` field into `f2 x` and `f2 y` sets as follows and make them available to OnSet:

```
f1: n1
f2 x: n1, n2
f2 y: n3
```

If the `rowSets` option is set to `true`, the set membership is transposed and the sets become:

```
n1: f1, f2 x
n2: f2 x
n3: f2 y
```

This component can be found in the `candela/plugins/onset` plugin.

Options

data (*Table*) The data table.

id (String) A field containing unique ids for each record.

sets (Array of String) A list of fields containing 0/1 set membership information which will be populated in the OnSet view.

fields (Array of String) A list of categorical fields that will be translated into collections of 0/1 sets for every distinct value in each field and populated in the OnSet view.

rowSets (Boolean) If `false`, treat the columns as sets, if `true`, treat the rows as sets. Default is `false`.

1.3.13 ScatterPlot

A scatterplot. This visualization will plot values at specified **x** and **y** positions. Additional fields may determine the **color**, **size**, and **shape** of the plotted points.

This component can be found in the `candela/plugins/vega` plugin.

Example

JavaScript

```
<body>
<script src="//unpkg.com/candela/dist/candela.min.js"></script>
<script>
  var el = document.createElement('div')
  document.body.appendChild(el);

  var data = [];
  for (var d = 0; d < 10; d += 1) {
    data.push({
      a: d,
      b: d
    });
  }

  var vis = new candela.components.ScatterPlot(el, {
    data: data,
    x: 'a',
    y: 'b',
    width: 700,
    height: 400
  });
  vis.render();
</script>
</body>
```

Python

```
import pycandela

data = [{'a': d, 'b': d} for d in range(10)]

pycandela.components.ScatterPlot(
    data=data, x='a', y='b', width=700, height=400)
```

R

```
library(candela)

candela('ScatterPlot', data=mtcars, x='mpg', y='wt', color='disp')
```

Options

data (*Table*) The data table.

x (String) The x axis field.

xType (String) The data type for the `x` field. The default is `"quantitative"`.

y (String) The y axis field.

yType (String) The data type for the `y` field. The default is `"quantitative"`.

color (String) The field used to color the points.

colorType (String) The data type for the `color` field. The default is `"nominal"`.

size (String) The field used to size the points.

sizeType (String) The data type for the `size` field. The default is `"quantitative"`.

shape (String) The field used to set the shape of the points.

shapeType (String) The data type for the `shape` field. The default is `"nominal"`.

filled (String) Whether to fill the points or just draw the outline. The default is `true`.

width (Number) Width of the chart in pixels. See *Sizing*.

height (Number) Height of the chart in pixels. See *Sizing*.

renderer (String) Whether to render in `"svg"` or `"canvas"` mode (default `"canvas"`).

1.3.14 ScatterPlotMatrix

A scatterplot matrix. This visualization will display a scatterplot for every pair of specified **fields**, arranged in a grid. Additional fields may determine the **size**, **color**, and **shape** of the points.

This component can be found in the `candela/plugins/vega` plugin.

Example

JavaScript

```
<body>
<script src="//unpkg.com/candela/dist/candela.min.js"></script>
<script>
  var el = document.createElement('div')
  document.body.appendChild(el);

  var data = [];
  for (var d = 0; d < 10; d += 1) {
    data.push({
      a: d,
      b: 10 - d,
      name: d
    });
  }

  var vis = new candela.components.ScatterPlotMatrix(el, {
    data: data,
    fields: ['a', 'b']
  });
  vis.render();
</script>
</body>
```

Python

```
import pycandela

data = [{'a': d, 'b': 10 - d, 'name': d} for d in range(10)]

pycandela.components.ScatterPlotMatrix(data=data, fields=['a', 'b'])
```

R

```
library(candela)

candela('ScatterPlotMatrix', data=mtcars, fields=c('mpg', 'wt', 'disp'))
```

Options

data (*Table*) The data table.

fields (Array of String) The fields to use as axes in the scatterplot matrix. Specifying N fields will generate an N-by-N matrix of scatterplots.

color (String) The field used to color the points.

colorType (String) The data type for the `color` field. The default is `"nominal"`.

size (String) The field used to size the points.

sizeType (String) The data type for the `size` field. The default is `"quantitative"`.

shape (String) The field used to set the shape of the points.

shapeType (String) The data type for the `shape` field. The default is `"nominal"`.

filled (String) Whether to fill the points or just draw the outline. The default is `true`.

width (Number) Width of the chart in pixels. See *Sizing*.

height (Number) Height of the chart in pixels. See *Sizing*.

renderer (String) Whether to render in `"svg"` or `"canvas"` mode (default `"canvas"`).

1.3.15 SentenTree

A [SentenTree](https://github.com/twitter/SentenTree) sentence visualization. Given a table of text samples about some topic, SentenTree attempts to abstract out the common expressions between them, visualizing them as a flowing "sentence tree".

The **data** table contains a list of objects, each with an **id** field containing a unique identifier for each row, a **text** field containing the text sample, and a **count** field expressing the strength of that sample, or number of times it occurred in the corpus, etc.

This component can be found in the `candela/plugins/sententree` plugin.

Example

JavaScript

```
<body>
<script src="//unpkg.com/candela/dist/candela.min.js"></script>
<script>
  var el = document.createElement('div')
  el.setAttribute('width', 1200);
  el.setAttribute('width', 700);

  document.body.appendChild(el);

  var data = [
    {id: 0, count: 3787, text: 'brazil\'s marcelo scores the first goal of the world␣
↪cup ... against brazil.'},
    {id: 1, count: 2878, text: 'at least brazil have scored the first goal of the␣
↪world cup'},
    {id: 2, count: 1702, text: 'first game of the world cup tonight! can\'t wait!'},
    {id: 3, count: 1689, text: 'the first goal of the world cup is an own goal!␣
↪marcelo accidentally knocks it into his own net past julio cesar! croatia leads 1-0.
↪'},
    {id: 4, count: 1582, text: 'goal: brazil 0-1 croatia marcelo scores an own goal␣
↪in the 11th minute'},
    {id: 5, count: 1525, text: 'just like we predicted, a brazilian scored the first␣
↪goal in the world cup'},
    {id: 6, count: 1405, text: 'whoever bet that the first goal of the world cup was␣
↪going to be an own goal just made a lot of money.'},
    {id: 7, count: 1016, text: '736 players 64 matches 32 teams 12 stadiums 4 years␣
↪of waiting 1 winning country the 2014 world cup has started .'},
    {id: 9, count: 996, text: 'watching the world cup tonight! with the tour fam'},
    {id: 10, count: 960, text: 'the first goal of the world cup was almost as bad as␣
↪the opening ceremony.'},
    {id: 11, count: 935, text: 'live from the 2014 fifa world cup in brazil, the␣
↪unveiling of the happiness flag.'},
    {id: 13, count: 915, text: 'world cup starts today!!!!!! amazing!!!'},
    {id: 14, count: 818, text: 'the first goal scored of the world cup 2014... was an␣
↪own goal!'},
    {id: 15, count: 805, text: 'after 4 years, the wait is finally over.'},
    {id: 16, count: 803, text: 'that\'s not in the script! own goal from marcelo puts␣
↪croatia up 0-1.'},
```

(continues on next page)

(continued from previous page)

```
    {id: 17, count: 746, text: 'that moment when you score an own goal in the opening␣
↪game of the world cup.'},
    {id: 18, count: 745, text: 'scoring on themselves in the world cup'},
    {id: 19, count: 719, text: 'world cup 2014 first goal is own-goal by marcelo'}
  ];

  var vis = new candela.components.SentenTree(el, {
    data: data,
    graphs: 3
  });
  vis.render();
</script>
</body>
```

Python

```
import pycandela

data = [
  {'id': 0, 'count': 3787, 'text': 'brazil\'s marcelo scores the first goal of the␣
↪world cup ... against brazil.'},
  {'id': 1, 'count': 2878, 'text': 'at least brazil have scored the first goal of the␣
↪world cup'},
  {'id': 2, 'count': 1702, 'text': 'first game of the world cup tonight! can\'t wait!
↪'},
  {'id': 3, 'count': 1689, 'text': 'the first goal of the world cup is an own goal!␣
↪marcelo accidentally knocks it into his own net past julio cesar! croatia leads 1-0.
↪'},
  {'id': 4, 'count': 1582, 'text': 'goal: brazil 0-1 croatia marcelo scores an own␣
↪goal in the 11th minute'},
  {'id': 5, 'count': 1525, 'text': 'just like we predicted, a brazilian scored the␣
↪first goal in the world cup'},
  {'id': 6, 'count': 1405, 'text': 'whoever bet that the first goal of the world cup␣
↪was going to be an own goal just made a lot of money.'},
  {'id': 7, 'count': 1016, 'text': '736 players 64 matches 32 teams 12 stadiums 4␣
↪years of waiting 1 winning country the 2014 world cup has started .'},
  {'id': 9, 'count': 996, 'text': 'watching the world cup tonight! with the tour fam'}
↪,
  {'id': 10, 'count': 960, 'text': 'the first goal of the world cup was almost as bad␣
↪as the opening ceremony.'},
  {'id': 11, 'count': 935, 'text': 'live from the 2014 fifa world cup in brazil, the␣
↪unveiling of the happiness flag.'},
  {'id': 13, 'count': 915, 'text': 'world cup starts today!!!!!! amazing!!!'},
  {'id': 14, 'count': 818, 'text': 'the first goal scored of the world cup 2014...␣
↪was an own goal!'},
  {'id': 15, 'count': 805, 'text': 'after 4 years, the wait is finally over.'},
  {'id': 16, 'count': 803, 'text': 'that\'s not in the script! own goal from marcelo␣
↪puts croatia up 0-1.'},
  {'id': 17, 'count': 746, 'text': 'that moment when you score an own goal in the␣
↪opening game of the world cup.'},
  {'id': 18, 'count': 745, 'text': 'scoring on themselves in the world cup'},
  {'id': 19, 'count': 719, 'text': 'world cup 2014 first goal is own-goal by marcelo'}
]

pycandela.components.SentenTree(data=data, id='id', count='count', text='text')
```

R

```
library(candela)

id = c(0, 1, 2, 3, 4, 5, 6, 7, 8, 9, 10, 11, 12, 13, 14, 15, 16, 17, 18, 19)
count = c(3787, 2878, 1702, 1689, 1582, 1525, 1405, 1016, 996, 960, 935, 915, 818,␣
↪805, 803, 746, 745, 719)
text = c('brazil\'s marcelo scores the first goal of the world cup ... against brazil.
↪', 'at least brazil have scored the first goal of the world cup', 'first game of␣
↪the world cup tonight! can\'t wait!', 'the first goal of the world cup is an own␣
↪goal! marcelo accidentally knocks it into his own net past julio cesar! croatia␣
↪leads 1-0.', 'goal: brazil 0-1 croatia marcelo scores an own goal in the 11th minute
↪', 'just like we predicted, a brazilian scored the first goal in the world cup',
↪'whoever bet that the first goal of the world cup was going to be an own goal just␣
↪made a lot of money.', '736 players 64 matches 32 teams 12 stadiums 4 years of␣
↪waiting 1 winning country the 2014 world cup has started .', 'watching the world␣
↪cup tonight! with the tour fam', 'the first goal of the world cup was almost as bad␣
↪as the opening ceremony.', 'live from the 2014 fifa world cup in brazil, the␣
↪unveiling of the happiness flag.', 'world cup starts today!!!!!! amazing!!!', 'the␣
↪first goal scored of the world cup 2014... was an own goal!', 'after 4 years, the␣
↪wait is finally over.', 'that\'s not in the script! own goal from marcelo puts␣
↪croatia up 0-1.', 'that moment when you score an own goal in the opening game of␣
↪the world cup.', 'scoring on themselves in the world cup', 'world cup 2014 first␣
↪goal is own-goal by marcelo')

data = data.frame(id, count, text)

candela('SentenTree', data=data, id='id', color='class', threshold=0.4)
```

Options

data (*Table*) The data table.

id (String) The ID field. Can contain any data type, but the value should be unique to each data record.

text (String) The text sample field.

count (Integer) The field expressing the count or strength of each text sample.

1.3.16 SimilarityGraph

An interactive similarity graph. Given a list of entities that encode a connection strength to the other entities, this component creates a graph with the entities as the nodes, and a link appearing between nodes whose connection strength exceeds some threshold.

The **data** table contains a list of objects, each with an **id** field containing a unique identifier for each entity. Each object should also have a numeric fields named by the IDs of the other entities, containing a link strength to each entity. If any entity's link strength is missing, it is presumed to be 0. Each object may optionally contain a **color** field, containing a value identifying its color, and a **size** field, which is either a number (in pixels) for the radius of each node, or a string identifying a field in **data** that contains a number that will be mapped to the radius for each node. **threshold** is a numeric value specifying the minimum value for a link strength to appear in the graph. **linkDistance** sets the desired length of the links in pixels.

This component can be found in the `candela/plugins/similaritygraph` plugin.

Example

JavaScript

```
<body>
<script src="//unpkg.com/candela/dist/candela.min.js"></script>
<script>
  var el = document.createElement('div')
  el.setAttribute('width', 700);
  el.setAttribute('width', 700);

  document.body.appendChild(el);

  var alphabet = 'abcdefghijklmnopqrstuvwxyz';
  var vowels = 'aeiou'.split('');

  var data = [];
  for (var i = 0; i < 26; i++) {
    var letter = {
      id: alphabet[i],
      size: 10 + i,
      color: vowels.indexOf(alphabet[i]) > 0 ? 'vowel' : 'consonant'
    };

    for (var j = 0; j < 26; j++) {
      letter[alphabet[j]] = Math.abs(j - i);
    }

    data.push(letter);
  }

  var vis = new candela.components.SimilarityGraph(el, {
    data: data,
    size: 'size',
    threshold: 20
  });
  vis.render();
</script>
</body>
```

Python

```
import pycandela

data = [
  {'id': 'A', 'class': 0, 'A': 1.0, 'B': 0.5, 'C': 0.3},
  {'id': 'B', 'class': 1, 'A': 0.5, 'B': 1.0, 'C': 0.2},
  {'id': 'C', 'class': 1, 'A': 0.3, 'B': 0.2, 'C': 1.0}
]

pycandela.components.SimilarityGraph(data=data, id='id', color='class', threshold=0.4)
```

R

```
library(candela)

id = c('A', 'B', 'C')
class = c(0, 1, 1)
```

(continues on next page)

(continued from previous page)

```
A = c(1.0, 0.5, 0.3)
B = c(0.5, 1.0, 0.2)
C = c(0.3, 0.2, 1.0)
data = data.frame(id, class, A, B, C)

candela('SimilarityGraph', data=data, id='id', color='class', threshold=0.4)
```

Options

data (*Table*) The data table.

id (String) The ID field. Can contain any data type, but the value should be unique to each data record.

color (String) The field used to color the nodes. See *Color scales*.

size (String or Number) If a string, the field used to provide the radius for each node; if a number, the radius to use for *all* nodes.

threshold (Number) The link strength above which a link will appear in the graph.

linkDistance (Number) The desired length of each link in pixels.

1.3.17 TreeHeatmap

A heatmap with optional hierarchies attached to the rows and columns.

This component can be found in the `candela/plugins/treeheatmap` plugin.

Example

The examples below assume you have downloaded the example data.

JavaScript

```
<body>
<script src="//unpkg.com/candela/dist/candela.min.js"></script>
<div id="vis" style="width:600px;height:600px"></div>
<script type="text/javascript" >
  var el = document.getElementById('vis');

  d3.json('heatmap.json', function (error, data) {
    var vis = new candela.components.TreeHeatmap(el, {
      data: data,
      scale: 'column'
    });
    vis.render();
  });
</script>
</body>
```

Python

```
import pycandela
import json
```

(continues on next page)

(continued from previous page)

```
data = json.load(open('heatmap.json'))

pycandela.components.TreeHeatmap(data=data, scale='column')
```

R

```
library(candela)
library(jsonlite)

fname <- 'heatmap.json'
s <- readChar(fname, file.info(fname)$size)
data <- fromJSON(s)

candela('TreeHeatmap', data=data, scale='column')
```

Options

data (*Table*) The data table.

idColumn (String) A column with unique identifiers. If not set, the visualization will use a column with an empty name, or a column named "_" or "_id" if it exists.

scale (String) Specify whether to color the data values with a global scale ("global"), scale each row or column separately ("row" or "column"), or use a -1 to 1 color scale suitable for a correlation matrix ("correlation"). The view uses a global scale if this parameter is not specified.

clusterRows (Boolean) If set to true, orders the rows by hierarchical cluster linkage. This option requires specially-defined columns named "_cluster", "_child1", "_child2", "_distance", and "_size" to define the clustering of the rows. See the heatmap analysis in pysciencedock for an example of how to create the appropriate hierarchy columns.

clusterColumns (Boolean) If set to true, orders the columns by hierarchical cluster linkage. this option requires specially-defined rows named "_cluster", "_child1", "_child2", "_distance", and "_size" to define the clustering of the columns. See the heatmap analysis in pysciencedock for an example of how to create the appropriate hierarchy rows.

threshold (Number) The value to threshold by according to the threshold mode.

thresholdMode (String) If set, uses the threshold value to display only certain cells in the table. Valid values are "none" (no thresholding), "greater than" (show values greater than the threshold), "less than" (show values less than the threshold), or "absolute value greater than" (show only values whose absolute value is greater than the threshold. If set to anything other than "none", the threshold parameter must also be set.

removeEmpty (Boolean) If true, removes rows and columns that are entirely filtered out by the threshold. Clustering by rows and columns will not be used if this flag is set.

1.3.18 UpSet

An UpSet set visualization.

UpSet interprets binary columns (i.e. columns with a literal `"0"` or `"1"`, `"true"` or `"false"`, `"yes"` or `"no"` in every row) as sets. Any field in the **sets** option will be interpreted in this way. Since most data is not arranged in binary columns, the visualization also supports arbitrary categorical fields with the **fields** option. Each field specified in this list will first be preprocessed into a collection of 0/1 columns that are then passed to UpSet.

For example, suppose the data table is:

```
[
  {"id": "n1", "f1": 1, "f2": "x"},
  {"id": "n2", "f1": 0, "f2": "x"},
  {"id": "n3", "f1": 0, "f2": "y"}
]
```

You could create an UpSet visualization with the following options:

```
new UpSet({
  data: data,
  id: 'id',
  sets: ['f1'],
  fields: ['f2']
});
```

This would preprocess the `f2` field into `f2 x` and `f2 y` sets as follows and make them available to UpSet:

```
[
  {"id": "n1", "f1": 1, "f2 x": 1, "f2 y": 0},
  {"id": "n2", "f1": 0, "f2 x": 1, "f2 y": 0},
  {"id": "n3", "f1": 0, "f2 x": 0, "f2 y": 1}
]
```

This component can be found in the `candela/plugins/upset` plugin bundle.

Options

data (*Table*) The data table.

id (String) A field containing a unique identifier for each record.

fields (Array of String) A list of fields that will be shown on the UpSet view. Each value in each field will be converted to a set membership 0/1 field before being passed to UpSet.

sets (Array of String) A list of fields that will be shown on the UpSet view. Each field is assumed to already be a 0/1 set membership field.

metadata (Array of String) A list of fields that will be shown as metadata when drilling down to individual records. Numeric data will also be shown in summary box plots to the right of each set.

1.3.19 Candela JavaScript API

- *candela.components* - The built-in Candela components.
- *Sizing*
- *Field matchings*
- *Data types*
- *Visualization components* - The base class for Candela components.
- *candela.mixins* - The built-in Candela component mixins.
- *Utilities* - Candela utility functions.

Components

Candela comes with several visualization components ready to use. To make it easier to include these components in your project, they are partitioned into several built-in *plugins*. Each plugin exports its contents through its `index.js` file:

```
import * as candelaVega from 'candela/plugins/vega';

let vis = new candelaVega.BarChart(...);
```

and can load its contents into the `candela.components` object through its `load.js` file as follows:

```
import candela from 'candela';
import 'candela/plugins/vega/load.js';

let vis = new candela.components.BarChart(...);
```

You can also import a component directly:

```
import BarChart from 'candela/plugins/vega/BarChart';

let vis = new BarChart(...);
```

And as a last resort, you can also import the *candela* bundle, which is built to contain every component, preloaded into `candela.components`:

```
import candela from 'candela/dist/candela';

let vis = new candela.components.BarChart(...);
```

However, the *candela* bundle is very large; using one of the other methods of building your application will result in a smaller, more manageable bundle size.

The current list of plugins is:

- `candela/plugins/vega` - Charts based on Vega, including basic chart types such as bar charts, scatter plots, and histograms.
- `candela/plugins/geojs` - Components based on GeoJS for geospatial data visualization.
- `candela/plugins/glo` - A component based on GLO - "graph-level operations".
- `candela/plugins/lineup` - A component based on LineUp for visualizing rankings.
- `candela/plugins/onset` - A component based on OnSet for visualizing subset relationships.
- `candela/plugins/sententree` - A component based on SentenTree for visualizing the grammatical structure of a corpus of text.
- `candela/plugins/similaritygraph` - A specialized interactive graph visualization component for investigating degrees of similarity between nodes in a data table.
- `candela/plugins/trackerdash` - A component based on the TrackerDash algorithm metric tracking dashboard.
- `candela/plugins/treeheatmap` - A heatmap combined with hierarchical clustering.
- `candela/plugins/upset` - A component based on UpSet, also for visualizing subset relationships.

For more details about each component (including how to import these bundles into your project), see the *full list* of component documentation.

Sizing

Components often have a **width** and **height** option, which specify the width and height of the component in pixels.

Field matchings

Axis scales

Several components have options that relate to the axes of the visualization. These are commonly called **x** and **y** but may also have more descriptive names. The component will often automatically detect the type of values in the field being mapped to an axis and will create an appropriate axis type, such as evenly-spaced values for string fields and a continuous-ranged axis for numeric and date fields. Visualizations showing continuous-ranged axes often allow pan and zoom of the axis by dragging and scrolling in the visualization area.

Color scales

Many Candela components contain a **color** option, which will color the visual elements by the field specified. When possible, **color** will detect the type of the column and use an appropriate color scale. For fields containing string/text values, the visualization will use a color scale with distinct colors for each unique value. For fields containing numeric or date values, the visualization will use a smooth color gradient from low to high values.

Data types

Table

A Candela table is an array of records of the form:

```
[
  {
    "a": 1,
    "b": "Mark",
    "c": "Jun 1, 2010"
  },
  {
    "a": 2,
    "b": "Andy",
    "c": "Feb 6, 2010"
  },
  {
    "a": 3,
    "b": "Joe",
    "c": "Nov 27, 2010"
  }
]
```

Visualization components

`VisComponent` is the base class for Candela visualization components. This class is intentionally minimal, because there are only a few common features of all Candela components:

1. Candela components work on the web, so the constructor looks like `new VisComponent(el)`, where `el` is (usually) a DOM element. The `VisComponent` constructor attaches `el` to the object, so you can always refer to it using `this.el`.
2. Candela components perform some type of visualization, so they have a *render* method. The base class *render* simply raises an exception.
3. Sometimes you need to change an aspect of the visualization at runtime, such as the color map, which columns of data are being visualized, or even the data itself; to support such changes, Candela components have an *update* method. The base class *update* returns a promise object that delivers the component itself.
4. When a visualization component reaches the end of its lifecycle, it may need to clean up after itself, which can be done in the component's *destroy* method. The base class *destroy* simply removes all content from *this.el*.

You can create a concrete visualization component by extending `VisComponent`. The following best practices maximize clarity, reusability, and interoperability of your components (in the rest of this document, imagine that `Component` is declared as an extension of `VisComponent`, such as `BarChart`):

1. The *constructor* should take an additional parameter `options` encapsulating all runtime options for the component.
2. The component should report its expected inputs in *`Component.options`*.

`var component = new Component`(*el*, *options*)
Constructs a new instance of the Candela component.

- **el** is a valid container for the visualization. The container will often be a DOM element such as `<div>`, but may need to be another type for certain visualizations.
- **options** is an object containing the initial options for the visualization. This includes any data, visual matchings, or other settings pertinent to the visualization. Options are specified in the form `{name: value}`.

Note: The constructor for the abstract superclass is empty. You should use the constructor for specific subclasses of `VisComponent`.

`component.`**`render`**`()`
Renders the component to its container using the current set of options.

Note: The `VisComponent` `render()` method simply throws an exception; if you truly want your component to do nothing when it renders, simply redefine the method to be a no-op.

`component.`**`update`**`(`*options*`)`
Changes the component state to reflect *options*. This method allows for incremental changes to the component state. The form of *options* should be the same as what the *constructor* takes. The difference is, only the options given to this method should change, while any left unspecified should remain as they are.

Note: The `VisComponent` `update()` method returns a promise object that delivers the component itself without changing it, since the semantics of updating will be different for every component.

`component.`**`destroy`**`()`
Performs any cleanup required of the component when it is no longer needed. This may be as simple as emptying the container element the component has been using, or it may involve unregistering event listeners, etc.

Note: The `VisComponent` `destroy()` method just empties the top-level container, since this is a common "cleanup" operation.

`component.`**`empty`**`()`
Convenience method that empties the component's container element. This can be used in the constructor to prepare the container element, or in the *destroy* method to clean up after the component.

`component.`**`serializationFormats`**
The `serializationFormats` field is a list of strings of supported formats. Formats include:

- `'png'`: A base64-encoded string for a PNG image. The string may be placed in the `src` attribute of an `<img>` element to show the image.
- `'svg'`: A base64-encoded string for an SVG scene. The string may be placed in the `src` attribute of an `<img>` element to show the image.

`component.`**`serialize`**(*format*)
Serializes the component into the specified **format**.

`Component.`**`options`**
This static property is an array of *Option specifications*, containing a description of the options this visualization accepts. This may be used to introspect the component to implement features such as automatic UI building.

`Component.`**`container`**
A static field containing the type of container this visualization can be added to. The most common is DOMElement.

Mixins

Candela uses mixins to add functionality to `VisComponent` when creating a new component. To use a mixin, the pattern is as follows:

```
class MyCoolVisualization extends Mixin(candela.VisComponent) {
  .
  .
  .
}
```

The class `Mixin` is defined using this pattern:

```
const Mixin = Base => class extends Base {
  mixinMethod() {
    .
    .
    .
  }
};
```

This is a function expression that maps a base class to a new, unnamed class - in other words, mixins are functions that can be applied to `VisComponent` (or any existing component class) to yield a new class with extra functionality.

Candela comes with several mixins, which are available in the plugin `candela/plugins/mixin`.

`Events``()`

Adds basic event handling to the component. The component gains an `.on()` method that takes a string naming an event type, and a callback to invoke when that event occurs, and a `.trigger()` method that takes an event type and optional arguments to fire that event type with those arguments.

`InitSize``()`

Causes a `width` and `height` property to be written to the component, based on the size of `this.el` at the time the component is instantiated.

`Resize``()`

Uses the `Events` mixin to trigger a `resize` event whenever the containing element's size changes. The event fires with the new width and height of the element, and a reference to the component itself.

`AutoResize``()`

Combines the `InitSize` and `Resize` mixins, and automatically responds to the `resize` event by updating the `this.width` and `this.height` properties.

`VegaView``()`

Implements a Vega or Vega-Lite visualization component. Subclasses should implement a `generateSpec()` method which returns an appropriate Vega/Vega-Lite specification based on the view options.

Option specification

An option specification describes an input to a visualization as part of the *`Component.options`* array. It is an object containing the following properties:

name (String) The name of the option.

type (String) The type of the option. Type and format follow Girder Worker types/formats.

format (String) The format (specific encoding within a type) of the option. Type and format follow Girder Worker types/formats.

domain (*Domain*) Optional. A restriction on this option's set of allowed values.

Domain specification

The domain of an option restricts the set of allowed values for an option. It is an object with the following properties:

mode (String) The domain mode, one of `'choice'` or `'field'`. The `'choice'` mode will allow a fixed set of options set in the `'from'` field. The `'field'` mode will allow a field or list of fields from another input. If the option type is `'string'`, the option is a single field, and if the option type is `'string_list'`, the option accepts a list of fields.

from (Array or String) If the mode is `'choice'`, it is an array of strings to use as a dropdown. If the mode is `'field'`, it is the name of the input from which to extract fields.

fieldTypes (Array) If mode is `'field'`, this specifies the types of fields to support. This array may contain any combination of datalib's supported field types which include `'string'`, `'date'`, `'number'`, `'integer'`, and `'boolean'`.

Utilities

Candela utility functions.

`util.`**`getElementSize`**`(`*el*`)`
Returns an object with the fields `width` and `height` containing the current width and height of the DOM element **el** in pixels.

`util.`**`vega`**
`Utilities for generating Vega specifications.`

`util.vega.`**`chart`**`(`*template*, *el*, *options*, *done*`)`
Generates a Vega chart based on a **template** instantiated with **options**.

template is the [Vega template](#vega-templates) representing the chart.

el is the DOM element in which to place the Vega visualization.

options is an object of `{key: value}` pairs, containing the options to use while compiling the template. The options may contain arbitrarily nested objects and arrays.

done is a callback function to called when the Vega chart is generated. The function takes one argument that is the resulting Vega chart.

`util.vega.`**`transform`**(*template*, *options*)

Returns the instantiation of a **template** with the given **options**. This is the underlying function used by :js:func‘util.vega.chart‘ to instantiate its template before rendering with the Vega library.

template is the Vega template.

options is an object of `{key: value}` pairs, containing the options to use while compiling the template. The options may contain arbitrarily nested objects and arrays.

1.3.20 Candela Python API

The Candela Python library enables the use of interactive Candela visualizations within Jupyter notebooks.

`candela.components.`**`ComponentName`**(***options*)

Creates an object representing the Candela visualization specified by the given *options*. *ComponentName* is the name of the Candela component, such as ScatterPlot. For a full list of components and their options, see *Components*.

If a pandas DataFrame is passed as an option, it is automatically converted to a list of records of the form `[{"a": 1, "b": "foo"}, {"a": 2, "b": "baz"}]` before being sent to the Candela visualization.

To display a component, simply refer to the visualization, without assignment, as the last statement in a notebook cell. You may also explicitly display the visualization from anywhere within a cell using `vis.display()`.

1.3.21 Candela R API

The Candela R library enables the use of interactive Candela visualizations from R Studio by exposing Candela as htmlwidgets.

`candela`(*name*, ...)

Creates a widget representing the Candela visualization specified by the given options. *name* is the name of the Candela component, such as `"ScatterPlot"`. For a full list of components and their options, see *Components*.

If a data frame is passed as an option, it is automatically converted to a list of records of the form `[{"a": 1, "b": "foo"}, {"a": 2, "b": "baz"}]` before being sent to the Candela visualization.

1.3.22 Coding style guidelines

We follow semistandard for all JavaScript code.

1.3.23 Creating Candela releases

To perform a new release of Candela, please follow these steps. This assumes that the code on `master` is ready to be turned into a new release (i.e., it passes all tests and includes all new functionality desired for the new release). In this example, we will pretend that the new release version number will be 1.2.0.

1. Create a new release branch, named `release-1.2.0`:

```
git checkout -b release-1.2.0 master
```

2. Bump the version number to 1.2.0 by editing `package.json`. Make a commit with the commit message "Bump version number for release" and push the branch:

```
vim package.json
git commit -am 'Bump version number for release'
git push -u origin release-1.2.0
```

3. Make a new local branch to save your spot in the commit tree here. Be sure your checkout remains on `release-1.2.0`. You can do this with:

```
git branch save-point
```

4. Build the distribution files by using the "production" NPM script:

```
npm run build:production
```

   This will create a `dist` directory containing two JavaScript files, a regular and a minified version.

5. Commit the production files and push again.

```
git add dist
git commit -m 'Add production files for release'
git push
```

6. Create a pull request from the release-1.2.0 branch. **Make sure you base the PR against the `release` branch, not against `master`.**

7. Wait for an "LGTM" message, and then merge the pull request and delete the release-1.2.0 branch.

8. Check out the `release` branch, pull, tag a release, push, and then delete the `release-1.2.0` branch.

```
git checkout release
git pull
git tag v1.2.0
git push --tags
git branch -d release-1.2.0
```

9. Publish the new package to NPM. You will need to log in with your NPM credentials first.

```
npm login
npm publish
```

10. Merge the `save-point` branch into `master` (do not use a fast-forward merge, since this is a special type of commit to prepare master for development with a new version number, rather than adding any new functionality), push, then delete `save-point`. **Be sure you are not merging `release-1.2` or `release` into master; we do not want the distribution files to enter the mainline development branch.**

```
git checkout master
git merge save-point
git branch -d save-point
git push
```

This concludes the release process. You will have a new, tagged release published, with a corresponding commit on the `release` branch, while `master` will have the package version number updated, ready for further development.

1.3.24 Testing

Image testing

One of Candela's testing phases is **image testing**, in which images of visualization components are created programmatically and compared to established baseline images. These images are automatically uploaded to Kitware's Girder instance where they are catalogued by Travis build number and can be viewed by anyone.

Index