Candela

Release None

Kitware, Inc.

February 14, 2017

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"></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=df, 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.

Example

JavaScript

```
<body>
<script src="//unpkg.com/candela"></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*.

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

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

hover (Array of String) The fields to display on hover.

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.

Example

JavaScript

```
<body>
<script src="//unpkg.com/candela"></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.

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

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 BulletChart

A bullet chart based on the description by Perceptual Edge. The visualization takes a numeric **value** and plots it on a one-dimensional plot against comparison **markers** and background color **ranges**.

Example

JavaScript

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

  var vis = new candela.components.BulletChart(el, {
    value: 0.8,
    title: 'My measurement',
    subtitle: '... it is really important',
    ranges: [
      { min: 0, max: 0.2, foreground: 'gray', background: 'red' },
      { min: 0.2, max: 0.7, foreground: 'gray', background: 'yellow' },
      { min: 0.7, max: 1, foreground: 'gray', background: 'green' }
    ],
    width: 700,
    height: 100
  });
  vis.render();
</script>
</body>
```

Python

```
import pycandela

pycandela.components.BulletChart(
    value=0.8,
    title='My measurement',
    subtitle='... it is really important',
    ranges=[
        dict(min=0, max=0.2, foreground='gray', background='red'),
        dict(min=0.2, max=0.7, foreground='gray', background='yellow'),
        dict(min=0.7, max=1, foreground='gray', background='green')
    ],
    width=700,
    height=100
)
```

R

```
library(candela)

candela('BulletChart',
  value=0.8,
  title='My measurement',
  subtitle='... it is really important',
  ranges=data.frame(
    min=c(0, .2, .7), max=c(.2, .7, 1),
    foreground=c('gray', 'gray', 'gray'),
    background=c('red', 'yellow', 'green')),
  width=700,
  height=100
)
```

Options

value (Number) The value to plot in the bullet chart.

title (String) The title to show to the left of the chart.

subtitle (String) An optional subtitle to display below the title.

markers (Array of Number) Comparative markers to display as vertical lines.

ranges (Array of *Range*) Background ranges to display under the chart.

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"`).

Range specification

A range represents a visual range of an axis with background and foreground colors. It consists of an object with the following fields:

min (Number) The minimum value of the range.

max (Number) The maximum value of the range.

background (String) The background color of the range.

foreground (String) The color of values and markers that fall in this range (default: `"black"`).

1.3.6 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.

Example

JavaScript

```
<body>
<script src="//unpkg.com/candela"></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},
    {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.

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"`).

1.3.7 Geo

A geospatial chart using GeoJS.

Example

JavaScript

```
<body>
<script src="//unpkg.com/candela"></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'
          }
        ]
      }
    ]
  });
  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')
      )
    )
  )
)
```

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.

1.3.8 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.

Example

JavaScript

```
<body>
<script src="//unpkg.com/candela"></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}
 ];
 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.

1.3.9 Heatmap

A heatmap of a table. The heatmap displays a grid of rectangular patches for a set of **fields** in the **data** table using a separate color scale for each field. An optional **id** field is used to name the records, and the records can be ordered using a **sort** field.

Example

JavaScript

```
<body>
<script src="//unpkg.com/candela"></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.Heatmap(el, {
    data: data,
    fields: ['a', 'b'],
    id: 'name',
    width: 700,
    height: 400
  });
  vis.render();
</script>
</body>
```

Python

```
import pycandela

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

pycandela.components.Heatmap(
    data=data, fields=['a', 'b'], id='name', width=700, height=400)
```

R

```
library(candela)

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

Options

data (*Table*) The data table.

fields (Array of String) The fields to display in the heatmap. Numeric and date fields are colored with gradient color scales, while string fields are colored with categorical color scales.

sort (String) An optional field used to sort the records.

id (String) An optional field used to label the records. Must be a unique value for each record. If unset, uses an auto-generated *_id* field.

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 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.

Example

JavaScript

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

  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,
    bin: '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, bin='a', width=700, height=400)
```

R

```
library(candela)

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

Options

data (*Table*) The data table.

bin (String) The field to summarize. See *Axis scales*.

aggregate (String) An optional field to aggregate per bin. Must contain numeric data. See *Axis scales*.

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 LineChart

A line chart. The chart plots a line for each specified **y** field against a single **x** field.

Example

JavaScript

```
<body>
<script src="//unpkg.com/candela"></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=list('wt'), color='disp')
```

Options

data (*Table*) The data table.

x (String) The field containing x-coordinates for the lines. The field must contain date or numeric data. See *Axis scales*.

y (Array of String) The fields containing y-coordinates for the lines. The fields must contain date or numeric data. See *Axis scales*.

hover (Array of String) The fields to display on hover.

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

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

hoverSize (Number) Displays the hover value when the pointer is within this number of pixels (default 20).

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

1.3.12 LineUp

A LineUp table ranking visualization.

Example

JavaScript

```
<body>
<script src="//unpkg.com/candela"></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).

1.3.13 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
```

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`.

1.3.14 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.

Example

JavaScript

```
<body>
<script src="//unpkg.com/candela"></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. Must contain numeric data. See *Axis scales*.

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

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

shape (String) The field used to determine the shape of each point.

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

hover (Array of String) The fields to display on hover.

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 ScatterPlotMatrix

A scatterplot matrix. This visualization will display a scatterplot for every pair of specified **fields**, arranged in a grid. An additional field may determine the **color** of the points.

Example

JavaScript

```
<body>
<script src="//unpkg.com/candela"></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. The fields must contain numeric data. See *Axis scales*.

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

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"`).

1.3.16 SurvivalPlot

A survival plot, used to visualize mortality rates of, e.g., patients with a disease undergoing experimental treatments. This visualization expects data with a column specifying **time** as an offset from the start of the study indicating the time of observed deaths or withdrawal from the study (aka "censoring"), a column specifying **censorship status** (`0` if the patient has left the study, aka been "right censored"; `1` if the patient has died), and an optional column by which to group patients by, e.g., treatment type.

The number of surviving patients is plotted as a decreasing step function, with crosses overplotted whenever a patient is censored from the study.

Options

data (*Table*) The data table.

time (String) The observation event time field. Must contain numeric data. See *Axis scales*.

censor (String) The censorship status field. Must contain `0` or `1` only.

group (String) The field used to split patients into groups. *Optional*

xAxis.title (String) The label to use for the x axis. Default: `Time`.

yAxis.title (String) The label to use for the y axis. Default: `Survivors`.

legend (Boolean) Whether to enable the legend. Default: `false`.

legendTitle (String) The title to use for the legend. Default: `Legend`.

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"`).

1.3.17 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}
]
```

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.

1.3.18 Candela JavaScript API

- *candela.components* - The built-in Candela components.
- *Sizing*

- *Field matchings*
- *Data types*
- *Visualization components* - The base class and mixins for Candela components.
- *Utilities* - Candela utility functions.

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.

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.`**`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.

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.

1.3.19 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()`.

1.3.20 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.

1.3.21 Coding style guidelines

We follow semistandard for all JavaScript code.

1.3.22 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.

1.3.23 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.

1.3.24 Vega templates

A Vega template is a JSON-serializable JavaScript object which translates into a Vega specification when instantiated with a given set of options. This enables a single specification to build Vega visualizations with a variety of data sources and other customization options.

The central mechanic to a Vega template is `@`-prefixed expressions that signal the template engine to perform custom logic. The result of that logic is injected into the result at the place that the expression occurs. An `@`-expression takes the following form:

```
["@op", arg, arg, ...]
```

This is not unlike a basic Scheme-like language. In general, the arguments are first evaluated, then custom logic is performed using built-in JavaScript functions implementing the various operations. The returned values of these functions determine what data is injected into the template at that location.

get

Syntax:

```
["@get", name]
```

Retrieves the value of the option `name`. The `name` may refer to a nested value using dot-notation.

Example:

```
transform(
  {"value": ["@get", "a.b"]},
  {a: {b: 4}}
)
===
{"value": 4}
```

let

Syntax:

```
["@let", [[name, value], [name, value], ...], body]
```

Evaluates `body` with a set of defined values. This overrides any prior values of the specified names.

Example:

```
transform(
  [
    "@let",
    [["a", 1], ["b", 2]],
    {"a": ["@get", "a"], "b": ["@get", "b"]}
  ],
  {b: 5}
)
===
{a: 1, b: 2}
```

defaults

Syntax:

```
["@defaults", [[name, value], [name, value], ...], body]
```

Evaluates `body` with a set of default option values. This behaves exactly like *let*, except that if `name` was provided as an option, the provided value is not overridden.

Example:

```
transform(
  [
    "@defaults",
    [["a", 1], ["b", 2]],
    {"a": ["@get", "a"], "b": ["@get", "b"]}
  ],
  {b: 5}
)
===
{a: 1, b: 5}
```

map

Syntax:

```
["@map", array, item_name, body]
```

Evaluates `body` once for each item in `array`, binding the current item to `item_name`. All non-null resulting values are concatenated into the output array.

Example:

```
transform(["@map", [1, 2, 3], "d", {a: ["@get", "d"]}])
===
[{"a": 1}, {"a": 2}, {"a": 3}]
```

if

Syntax:

```
["@if", condition, then_clause, else_clause]
```

Evaluates the `condition`. If the result is truthy in the JavaScript sense, the `then_clause` is evaluated and the result is placed in the template, otherwise the `else_clause` is evaluated and used.

Example:

```
transform(["@if", false, 10, 20])
===
20
```

eq, lt, gt

Syntax:

```
["@eq", a, b]
["@lt", a, b]
["@gt", a, b]
```

Evaluates to `true` or `false` depending on the result of JavaScript equality (`a === b`), less than (`a < b`), or greater than (`a > b`).

Example:

```
transform(["@lt", 10, 20])
===
true
```

and, or

Syntax:

```
["@and", arg, arg, ...]
["@or", arg, arg, ...]
```

Evaluates the logical AND or OR of the arguments.

Example:

```
transform(["@and", true, true, false])
===
false
```

length

Syntax:

```
["@length", obj]
```

Retrieves the `.length` property of `obj` if it is an Array or String.

Example:

```
transform([["@length", "abc"], ["@length", [1, 2, 3, 4]])
===
[3, 4]
```

mult, add

Syntax:

```
["@mult", arg, arg, ...]
["@add", arg, arg, ...]
```

Computes the product or sum of a set of numbers.

Example:

```
transform(["@mult", 3, 2, 4])
===
24
```

join

Syntax:

```
["@join", separator, array]
```

Joins the `array` of Strings into a single string separated by a `separator`.

Example:

```
transform(["@join", ":", ["a", "b", "c"]])
===
"a:b:c"
```

merge

Syntax:

```
["@merge", a, b]
```

Merges the value of `b` into the value of `a`. If `a` and `b` (or corresponding sub-objects of `a` and `b`) are objects, the keys in the result will be the union of keys in `a` and `b`. If a matching key is found in both `a` and `b`, the values are recursively merged. If `a` and `b` (or corresponding sub-objects of `a` and `b`) are arrays, the arrays are concatenated with `b`‘s values following `a`‘s.

Example:

```
transform(
  {
    "concatenate": ["@merge", [1, 2], [3]],
    "merge_keys": ["@merge", {"a": 1}, {"b": 2}]
  }
)
===
{
  "concatenate": [1, 2, 3],
  "merge_keys": {"a": 1, "b": 2}
}
```

colorScale

Syntax:

```
["@colorScale", options]
```

Creates a Vega Scale appropriate for coloring data elements by values in a certain field. The `options` argument is an object with the following properties:

name (String) The name to give the scale.

values (*Table*) The data table to be colored by the scale.

field (String) The field to be colored by the scale.

axis

Syntax:

```
["@axis", options]
```

Creates a horizontal or vertical axis for a Vega visualization. This includes a Vega Scale component and Axis component, as well as a set of Signals if the axis is to enable interactive pan and zoom. The `options` argument is an object with the following properties:

axis (String) The type of axis, either `"x"` (horizontal) or `"y"` (vertical).

data (*Table*) The data table represented by the axis.

field (String) The field of the data represented by the axis.

pan (Boolean) If `true`, enables axis panning through drag events.

zoom (Boolean) If `true`, enables axis zooming through scroll events.

The following options are passed through to the Vega Scale: points, zero, nice, round, padding, domain. All are optional except the domain. The following options are passed through to the Vega Axis: grid, title, properties. All of these are optional.

isStringField

Syntax:

```
["@isStringField", array, field]
```

Returns `true` if the first element of `array` is an object with a key named `field` which holds a string. This is a convenience for writing conditional code in Vega specifications depending on whether data contains string or numeric values.

Example:

```
transform(
  [
    "@isStringField",
    [{"a": "hi", "b": 1}, {"a": "there", "b": 2}],
    "a"
  ]
)
===
true
```

orient

Syntax:

```
["@orient", direction, obj]
```

Orients a Vega Mark object to a direction. This operation assumes the incoming object is oriented horizontally.

direction Either `"horizontal"` (the object is returned unchanged), or `"vertical"` (the object's x, y, width, and height properties are swapped).

obj A Vega Mark object.

Example:

```
transform(
  [
    "@orient",
    "vertical",
    {
      "x": {"value": 10},
      "y": {"value": 20},
      "width": {"value": 40}
    }
```

```
  ]
)
===
{
  "y": {"value": 10},
  "x": {"value": 20},
  "height": {"value": 40}
}
```

C

U

V