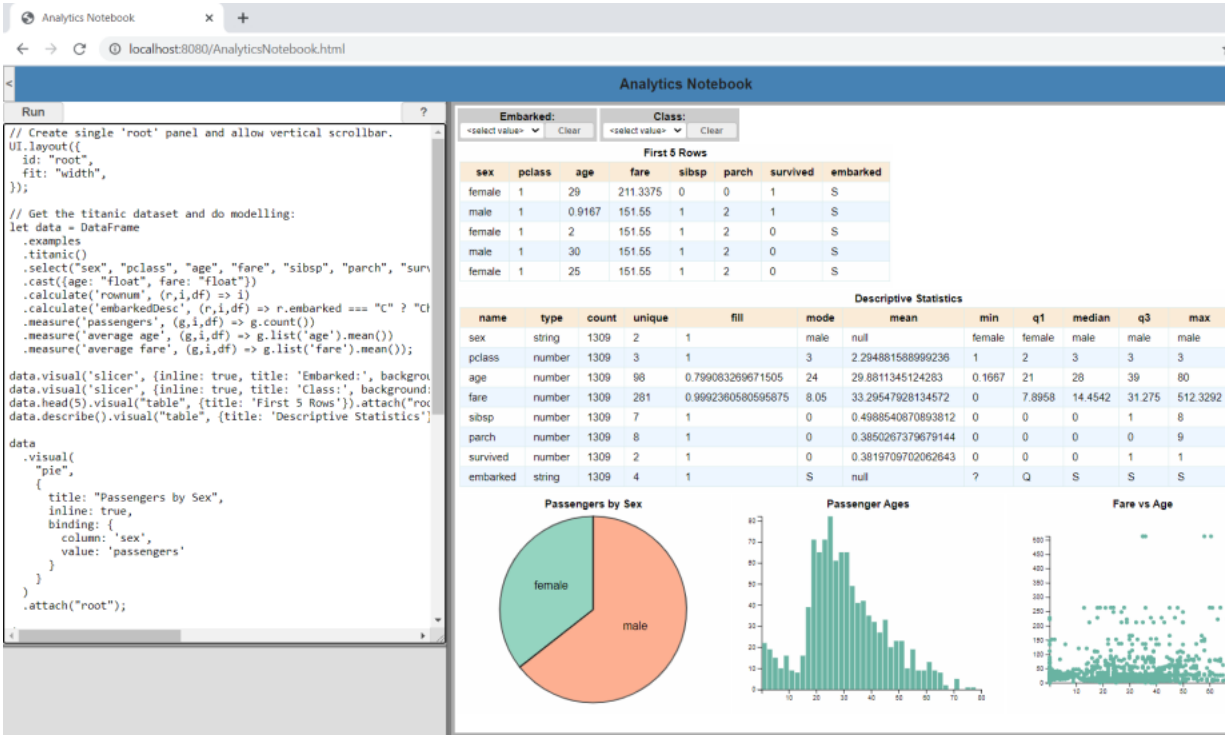
Analytics-Notebook

- Analytics-Notebook
- Application Layout
  - Code Section
  - Output Section
- Tutorial
- API
  - DataFrame
  - List
  - Visual
  - UI
- Documentation
- Examples
  - Titanic Exploratory Analysis
  - Anscombe's Quartet
  - Mtcars Analysis
- Development / Source Code
  - Scripts
- API Reference

The analytics-notebook is an analytical and statistical notebook application written in JavaScript, and is inspired by similar notebooks such as Jupyter notebooks, and https://observablehq.com/. Like the aforementioned applications, the intention for this application is to allow developers, analysts, researchers and statisticians to make sense out of data.

This application differs slightly from other notebooks in that instead of having multiple code blocks or cells entered inline within the output, analytics-notebook has a single code section and a single output section. The output section can be divided up into multiple visual 'panels' using the API, and visuals can be directed to these panels. A screenshot of the application is shown below:



Application Layout

The analytics-notebook application has 2 main sections.

- Code section
- Output section

Code Section

The code section is where the script is entered or loaded. Scripts are coded using Javascript, and there is an API available to help with common analytical tasks like data manipulation and with visuals rendering. To run a script simply click the run button, or press ctrl-enter. Any output of the script will be rendered to the output section.

At the base of the code section there is a console window. This is similar to the developer console windows found in browsers. This window will display any errors in your scripts, and output can also be directed to the console (you may not want all output from your scripts going to the output section, for example, debugging information) using the standard Javascript function console.log().

Output Section

The output section is used to render any output from the code. Output is typically

- Static text
- Raw unformatted data
- Simple scalar values
- Formatted tables
- Charts and other graphical visuals

Each distinct component displayed rendered to the output section is called a 'Visual'. There are broadly 2 kinds of visuals:

- Data bound visuals
- Static visuals

Data bound visuals are the most commonly used visual. These have a dataset (known as a DataFrame) attached to them. Furthermore, if the data changes (for example through some interactive visuals that can modify data - like the 'slicer' visual), then these visuals will automatically update. Static visuals are for things like headings, and free text. This information is normally static and does not have any data bound to it.

Tutorial

Notebooks are created using JavaScript. The script is entered in the left-hand script section. To run the script, click the **Run** button, or else press **ctrl-enter**. To run a simple hello world, paste the following JavaScript code into the code section, and press ctrl-enter to run:

```
alert("hello world!");
```

You will hopefully have received a hello world alert. Congratulations, you've just written your first (if not fairly pointless) notebook script.

You can include any valid JavaScript code in your script. You can use variables, create expressions, objects, functions - basically any valid JavaScript is allowed. For example, paste the following into the script section:

```
let a = 2;
let b = 3;
alert(`The sum of ${a} and ${b} is ${a + b}`);

// -> 'The sum of 2 and 3 is 5'.
```

Alerts are not particularly useful. You'll want to direct results, tables, and visuals to the output pane to create reports, dashboards and other useful analytical output. Each output component is called a 'Visual'. Visuals are either data bound or non-data-bound (static).

To create a static visual, we use the Visual.html method. Paste the following into the code section, and run:

```
Visual.html("hello world!").attach("root");
```

This should write 'hello world' to the output section. The string in the quotes of the html() call can be any valid html. For example, this is also perfectly allowed:

```
Visual.html(
  "<div style='padding: 12px; border: 1px solid #ccc; background: #eee;'>hello world!</div>"
).attach("root");
```

You may have noticed the above 2 examples included an attach() call. To actually place the visual into the output, you must include this function call. The value in brackets is the name of the panel to add the visual into. When you start a new notebook session, a default panel 'root' is always created for you.

Dashboards and other 'Business Intelligence' type reports often layout content in a grid fashion. You can to the same by using the UI.layout function. For example, we can create 4 panels, and direct output to each panels using the example below:

```
UI.layout({
  id: "root",
  direction: "horizontal",
  children: [
    {
      id: "left",
      direction: "vertical",
      children: ["top-left", "bottom-left"],
```

```
    },
    {
      id: "right",
      direction: "vertical",
      children: ["top-right", "bottom-right"],
    },
  ],
});

Visual.html("hello").attach("top-left");
Visual.html("world").attach("top-right");
Visual.html("from").attach("bottom-left");
Visual.html("Analytics Notebook").attach("bottom-right");
```

Each panel can contain multiple visuals, and you can make the layout grid as complex as you like. The UI.layout function supports various other features which are beyond the scope of this simple tutorial.

Static visuals are not going to take you very far though. The whole purpose of the analytics notebook is to enable analysts to obtain insights from data. For this, we need to use 'data bound' visuals.

To create data, the DataFrame class is generally used. External data can be obtained from the web using DataFrame.fetch(). However, for this tutorial, we're going to use a built-in dataset called 'iris'. Paste the following into the script section:

```
let data = DataFrame.examples.iris();
alert(data.count());
console.log(data);
```

When you run this script, you should receive an alert of '150'. This is the number of rows in the dataset. You may have also noticed the panel in the bottom left corner has some data. This panel is the 'console' panel, and information can be directed to this panel using the console.log() function. You will see in the above script, that we are writing the full DataFrame object to the console. The console is a useful tool for debugging your scripts.

Data on its own is not particularly useful. We need to visualise it. The simplest way to visualise data is via a table. Paste the following code into the script section:

```
let data = DataFrame.examples.iris().head(10).visual("table").attach("root");
```

This script introduces a few more concepts. Firstly, we can see a .head() function call. The .head() function returns the top 'n' rows from a DataFrame object. The .visual() function creates a data bound visual from the DataFrame object. Note how we need to provide the name of the visual type here. In this case we want to render a table, so use the type 'table'. There are a number of built-in visual types, and you can even customise the Analytics Notebook application and write your own custom visual types. Many visual types require an additional configuration argument to be provided, but the 'table' visual type renders fine without any special configuration.

You will also notice how in the above example, the function calls are all 'chained' together. The above syntax is the idiomatic style recommended. The above script could however, be re-written as:

```
let data = DataFrame.examples.iris();
let head = data.head(10);
let visual = head.visual("table");
visual.attach("root");
```

Finally, we can add a bit of interaction. Some visual types can be used to slice and dice the data. The 'slicer' visual is such a type. In general, if a number of visuals share the same DataFrame object in a script, then if this DataFrame object is sliced or diced (e.g. by a slicer visual), then all the other visuals bound to the same DataFrame object will automatically be sliced accordingly. Try running the following code:

```
UI.layout({
  id: "root",
  fit: "width",
});
let data = DataFrame.examples.iris();
data.visual("slicer", { binding: {column: "class" }}).attach("root");
data.visual("table").attach("root");
```

Here, you will be able to slice the table according to the 'class' field, and see the values changing. Note that for the slicer, we introduce a 'binding' option. The data connections to the visuals are defined through a bindings object. The bindings tells the visual which field(s) to add (and onto which axes for charts). The bindings will be feel similar to anyone who has used a business intelligence tool, and has dragged fields into a row / columns / details configuration area. In fact, the bindings names used in this tool are very similar: 'column','row','value','detail','color', and 'size'.

Hopefully, this tutorial has given you a basic grasp of how to run scripts. We recommend you now read the API documentation to get a deeper understanding of how to write more complex notebooks.

API

In order to work efficiently with the code and output sections, there is an API within the analytics-notebook. This API contains a number of classes and functions. The key classes are:

- DataFrame - for manipulating tabular data.
- List - for manipulating columnar data.
- Visual - for creating visuals.
- UI - for rendering to the output section.

DataFrame

The DataFrame class can be thought of as a 2-dimensional table. DataFrames are the work-horse of the analytics-notebook application. Json data can be read from a URL and is automatically returned as a DataFrame instance. Any transformation or filtering operations on a DataFrame instance generally return another DataFrame instance. In this way, the script can manipulate data by chaining calls together, to create a more natural-looking script. A DataFrame instance also supports cube-like features. Variables can be defined in the DataFrame instance, and these can are evaluated dynamically by any visuals bound to it. There are 2 types of variable:

- calculation: Calculations are evaluated row-by-row, A calculation behaves like a physical column, and can be used for slicing and dicing the data.
- measure: A measure is used for summarising data. Typically measures are used to calculate numeric aggregations (like 'total sales', or 'average performance').

The physical columns in the DataFrame instance, together with the calculations and measures, as known as a 'model' and an individual column, calculation or measure is known as a 'field'.

List

A List object can be considered as a 1 dimensional list of values or a single column from a DataFrame instance. Lists are typically used to process a column, often to aggregate the values in some way or to perform univariate analysis.

Visual

The Visual class is used to create and render visuals. As mentioned above, a key feature of data-bound visuals is that when the underlying data changes, the visual is automatically updated. This feature enables interactive dashboards to be built using this tool too.

UI

The UI class is used for manipulating the output section. Typically you don't need to use the UI class directly. It is called indirectly when you create visuals. For example, the normal pattern to render a visual is to create a Visual object from a DataFrame object using the visual() function, then attach to the DOM using the Visual.attach() function:

```
DataFrame.examples.mtcars().head(10).visual('table').attach('root');
```

The one method from the UI class which you will typically use (once per script) is the UI.layout() function. This function is used to design the grid layout for the output so that visuals can be positioned in the style of a dashboard.

Documentation

This documentation you're reading has been compiled using jsDoc. More information can be found from the home page: https://jsdoc.app/. The single-page version of the documentation also uses the following libraries:

- jsdoc-to-markdown: converts all documentation to markdown
- showdown: converts resulting markdown file to single .html file The full multifile version of the documentation has been styled using the ink-docstrap template (https://www.npmjs.com/package/ink-docstrap).

Examples

Titanic Exploratory Analysis

The following script performs some simple exploratory analysis on the 'Titanic' dataset:

```
// Create single 'root' panel and allow vertical scrollbar.
UI.layout({
  id: "root",
  fit: "width",
});

// Get the titanic dataset and do modelling:
let data = DataFrame
  .examples
  .titanic()
  .select("sex", "pclass", "age", "fare", "sibsp", "parch", "survived", "embarked")
  .cast({age: "float", fare: "float"})
  .calculate('rownum', (r,i,df) => i)
  .calculate('embarkedDesc', (r,i,df) => r.embarked === "C" ? "Cherbourg" : r.embarked === "S" ? "Southampton" : r.emb
```

```
  .measure('passengers', (g,i,df) => g.count())
  .measure('average age', (g,i,df) => g.list('age').mean())
  .measure('average fare', (g,i,df) => g.list('fare').mean());

data.visual('slicer', {inline: true, title: 'Embarked:', background: '#ccc', binding: {column: 'embarked'}}).attach('
data.visual('slicer', {inline: true, title: 'Class:', background: '#ccc', binding: {column: 'pclass'}}).attach('root')
data.head(5).visual("table", {title: 'First 5 Rows'}).attach("root");
data.describe().visual("table", {title: 'Descriptive Statistics'}).attach("root");

data
  .visual(
    "pie",
    {
      title: "Passengers by Sex",
      inline: true,
      binding: {
        column: 'sex',
        value: 'passengers'
      }
    }
  )
  .attach("root");

data
  .visual(
    "hist",
    {
      title: 'Passenger Ages',
      inline: true,
      binding: {
        column: 'age'
      }
    }
  )
  .attach("root");

data
  .visual(
    "scatter",
    {
      title: 'Fare vs Age',
      inline: true,
      binding: {
        column: 'average age',
        row: 'average fare',
        detail: 'rownum'
      }
    }
  )
  .attach("root");
```

Anscombe's Quartet

This notebook illustrates Anscombe's Quartet (https://en.wikipedia.org/wiki/Anscombe%27s_quartet):

```
UI.layout({
  id: "root",
  direction: "vertical",
  children: [
    {
      id: "top",
      direction: "horizontal",
      size: 10,
      children: [
        {
          id: "top-left",
        },
        {
          id: "top-right",
        },
      ],
    },
    {
      id: "bottom",
      direction: "horizontal",
      size: 10,
      children: [
        {
          id: "bottom-left",
        },
        {
          id: "bottom-right",
        },
      ],
```

```
    },
  ],
});
let anscombe = DataFrame
  .examples
  .anscombe();

let data = [
  { dataset: "1", panel: "top-left" },
  { dataset: "2", panel: "top-right" },
  { dataset: "3", panel: "bottom-left" },
  { dataset: "4", panel: "bottom-right" },
];

data.forEach((d) => {
  let dataset = anscombe
    .filter((r) => r.dataset === d.dataset)
    .calculate('rownum', (r,i,df) => i)
    .measure('x value', (g,i,df) => g.list('x').mean())
    .measure('y value', (g,i,df) => g.list('y').mean());

  dataset
    .visual(
      "scatter",
      {
        binding: {
          column: 'x value',
          row: 'y value',
          detail: 'rownum'
        },
        axes: {
          column: {
            min: 1,
            max: 20
          },
          row: {
            min: 1,
            max: 16
          }
        }
    }
  )
  .attach(d.panel);

  Visual.html(`mean(x): ${dataset.list("x").mean()}`).attach(d.panel);
  Visual.html(`var(x): ${dataset.list("x").var()}`).attach(d.panel);
  Visual.html(`mean(y): ${dataset.list("y").mean()}`).attach(d.panel);
  Visual.html(`var(y): ${dataset.list("y").var()}`).attach(d.panel);
  Visual.html(`corr(x,y): ${dataset.list("x").corr(dataset.list("y"))}`).attach(d.panel);
});
```

Mtcars Analysis

Another simple descriptive analytics notebook, this time featuring a heat-map to show correlation between variables.

```
let mt = DataFrame
  .examples
  .mtcars()
  .calculate('index', (r, i, df) => i)
  .measure('mpg value', (g, i, df) => g.list('mpg').mean())
  .measure('hp value', (g, i, df) => g.list('hp').mean())
  .measure('wt value', (g, i, df) => g.list('wt').mean())
  .measure('disp value', (g, i, df) => g.list('disp').mean())
  .measure('cyl value', (g, i, df) => g.list('cyl').mean())

mt.describe().visual('table').attach('root');

mt
  .corr()
  .measure('color', (g,i,df) => {
      // calculates the color attribute based on correlation (0 = red, +/-1 = green)
      let value = Math.abs(g.list('corr').mean());
      let green = 105 + Math.floor(150 * value);
      let red = 255 - Math.floor(150 * value);
      let blue = 100;
      return  `rgb(${red}, ${green}, ${blue})`;
    }
  )
  .visual(
    'crosstab',
    {
      binding: {
        column: 'x',
        row: 'y',
```

```
        value: 'corr',
       color: 'color'
      }
    }
  )
  .attach('root');

mt
  .visual(
    'scatter',
    {
      title: 'mpg vs hp (color: cyl)',
      inline: true,
      binding: {
        detail: 'index',
        column: 'mpg value',
        row: 'hp value',
        color: 'cyl value'
      }
    }
  )
  .attach('root');

mt
  .visual(
    'scatter',
    {
      title: 'wt vs disp (color: cyl)',
      inline: true,
      binding: {
        detail: 'index',
        column: 'wt value',
        row: 'disp value',
        color: 'cyl value'
      }
    }
  )
  .attach('root');
```

Development / Source Code

The source code for this project is available from: https://github.com/davidbarone/analytics-notebook. It's built using webpack, so you'll need the npm toolchain set up. You'll also need the following packages installed in the global namespace:

- jsDoc

Scripts

A number of scripts have been created for basic tasks:

- **docs**: Creates a full document web site using jsDoc and the in-docstrap template. Suitable for highest quality documentation.
- **docslite**: Creates a single-file documentation page via jsDoc-to-markdown and Showdown. This documentation is useful when you need a 'bundled' solution.
- **build**: Builds site in production mode.
- **build-dev**: Builds site in development mode.
- **serve**: runs the Webpack Dev Server with Hot Module Replacement (HMR).
- **test**: runs the test suite (using Jest) with code coverage metrics included

---

David Barone 04-Aug-2020

API Reference

Classes

DataFrame

Provides data extraction and manipulation services.

A DataFrame is similar to an Array object. It should be thought of as an array of objects, or a two dimensional array, similar to a table. Methods on the DataFrame class and DataFrame instances can be used for:

- Retrieving data
- Creating new data

- Cleansing & transforming data
- Summarising data
- Analysing data Many methods in a DataFrame instance return a new DataFrame instance. Therefore calls can be 'chained' together to form a data processing pipeline. Additionally, a DataFrame instance can have calculations and measures defined on it. These are formulae which are evaluated at runtime. Collectively, a DataFrame instance with its physical columns, calculations and measures is known as a 'model'. Each individual column, calculation or measure is know as a 'field'.

List

A List instance represents a single column from a DataFrame. Univariate analysis can be performed on a List instance. List instances are also used to summarise data.

UI

Controls rendering to the output section.

Visual

A visual represents any visual component rendered in the output, for example tables and charts.

Functions

isObject(item) â‡’ `boolean`

Simple object check.

validateBinding(dataFrame, binding, rules)

Validates the options.binding configuration passed into a Visual.

doBaseStyles()

Applies base styling on a visual.

Typedefs

ColumnCategory : `String`
JoinType : `String`
PanelAlignment : `String`
PanelFit : `String`

DataFrame

Provides data extraction and manipulation services.

A DataFrame is similar to an Array object. It should be thought of as an array of objects, or a two dimensional array, similar to a table. Methods on the DataFrame class and DataFrame instances can be used for:

- Retrieving data
- Creating new data
- Cleansing & transforming data
- Summarising data
- Analysing data Many methods in a DataFrame instance return a new DataFrame instance. Therefore calls can be 'chained' together to form a data processing pipeline. Additionally, a DataFrame instance can have calculations and measures defined on it. These are formulae which are evaluated at runtime. Collectively, a DataFrame instance with its physical columns, calculations and measures is known as a 'model'. Each individual column, calculation or measure is know as a 'field'.

Kind: global class

- DataFrame
  - *instance*
    - .getRowProxy(obj, i, dataFrame) â‡’ `object`
    - .model() â‡’ `Array`
    - .map(mapFunction) â‡’ `DataFrame`
    - .filter(filterFunction) â‡’ `DataFrame`
    - .group(groupingFunction, [aggregateFunction], [pivotFunction]) â‡’ `DataFrame`
    - .head(top) â‡’ `DataFrame`
    - .count() â‡’ `number`
    - .sort(sortFunction, descending) â‡’ `DataFrame`
    - .join(dataFrame, type, joinFunction, selectFunction) â‡’ `DataFrame`

- .list(column) â‡’ `List`
- .describe() â‡’ `DataFrame`
- .cast(types) â‡’ `DataFrame`
- .remove(â€¦columnNames) â‡’ `DataFrame`
- .select(â€¦columnNames) â‡’ `DataFrame`
- .setSlicer(visual, filterFunction)
- .unsetSlicer(visual)
- .resetSlicers()
- .visual(type, options, filterFunction) â‡’ `Visual`
- .clone() â‡’ `DataFrame`
- .corr() â‡’ `DataFrame`
- .forEach(forEachCallback)
- .calculate(name, calculation) â‡’ `DataFrame`
- .measure(name, measure) â‡’ `DataFrame`
- .columnCategory(columnName) â‡’ `ColumnCategory`
- .slicedData() â‡’ `DataFrame`
- .cube(â€¦columns)

static

- .examples
  - .iris()
  - .titanic()
  - .anscombe()
  - .mtcars()
- .create(arr) â‡’ `DataFrame`
- .fetch(url, options) â‡’ `DataFrame`

inner

- ~mapFunction â‡’ `object`
- ~filterFunction â‡’ `boolean`
- ~groupingFunction â‡’ `object`
- ~aggregateFunction â‡’ `object`
- ~pivotFunction â‡’ `string`
- ~sortFunction â‡’ `object`
- ~joinFunction â‡’ `boolean`
- ~mergeFunction â‡’ `object`
- ~forEachCallback : `function`
- ~measureFunction â‡’ `Number` | `String`

dataFrame.getRowProxy(obj, i, dataFrame) â‡’ `object`

Gets a proxy for a single row / object which can evaluate calculations in the model.

Kind: instance method of `DataFrame`
Returns: `object` - The returned object will be able to evaluate any calculations defined in the model.

Param	Type	Description
obj	`object`	The object to create a proxy for.
i	`Number`	The index of the row in the DataFrame instance.
dataFrame	`DataFrame`	The DataFrame instance with addional calculations & measures defined.

dataFrame.model() â‡’ `Array`

Gets all the columns, calculations, and measures in the model.

Kind: instance method of `DataFrame`
Returns: `Array` - A array of names in the model.

dataFrame.map(mapFunction) â‡’ `DataFrame`

Transforms a DataFrame instance using a mapping function.

Kind: instance method of `DataFrame`
Returns: `DataFrame` - The transformed DataFrame instance.

Param	Type	Description
mapFunction	`mapFunction`	The function to map the data..

Example *(Transforming a DataFrame instance)*

```
var people = [
  { name: "Tony", sex: "Male", age: 25 },
  { name: "Paul", sex: "Male", age: 17 },
  { name: "Sarah", sex: "Female", age: 42 },
  { name: "Debbie", sex: "Female", age: 62 },
  { name: "Michael", sex: "Male", age: 51 },
  { name: "Jenny", sex: "Female", age: 38 },
  { name: "Frank", sex: "Male", age: 32 },
  { name: "Amy", sex: "Female", age: 29 }
];

let df = DataFrame.create(people).map((p)=> { return { ageBand: Math.floor(p.age/10)*10, ...p }});
console.log(df);
```

dataFrame.filter(filterFunction) â‡’ DataFrame

Filters a DataFrame object using a filter function.

Kind: instance method of DataFrame
Returns: DataFrame - A filtered DataFrame instance.

Param	Type	Description
filterFunction	filterFunction	The function to filter the data. The function accepts a single parameter 'row' representing the current row. The function must return a boolean.

Example *(Filtering a DataFrame instance)*

```
var people = [
  { name: "Tony", sex: "Male", age: 25 },
  { name: "Paul", sex: "Male", age: 17 },
  { name: "Sarah", sex: "Female", age: 42 },
  { name: "Debbie", sex: "Female", age: 62 },
  { name: "Michael", sex: "Male", age: 51 },
  { name: "Jenny", sex: "Female", age: 38 },
  { name: "Frank", sex: "Male", age: 32 },
  { name: "Amy", sex: "Female", age: 29 }
];

let df = DataFrame.create(people).filter((p)=> { return p.sex==="Male" });
console.log(df);
```

dataFrame.group(groupingFunction, [aggregateFunction], [pivotFunction]) â‡’ DataFrame

Groups a DataFrame object using a grouping function and optional aggregation and pivot functions. The group function is mandatory and specifies the group values. If the aggregation function is ommitted, the result is simply the distinct group values. If an aggregation function is specified, then additional aggregated values for each group can be included. If a pivot function is specified, then the distinct string values returned by the pivot function are projected as column headers.

Kind: instance method of DataFrame
Returns: DataFrame - The grouped DataFrame instance.

Param	Type	Description
groupingFunction	groupingFunction	Callback function to group the DataFrame instance.
[aggregateFunction]	aggregateFunction	Optional callback to add aggregate data to the groups.
[pivotFunction]	pivotFunction	If specified, then the return value of the function ({string}) is used as a column header. Similar to pivoting in relational databases.

Example *(Grouping a DataFrame instance)*

```
var people = [
  { name: "Tony", sex: "Male", age: 25 },
  { name: "Paul", sex: "Male", age: 17 },
  { name: "Sarah", sex: "Female", age: 42 },
  { name: "Debbie", sex: "Female", age: 62 },
  { name: "Michael", sex: "Male", age: 51 },
  { name: "Jenny", sex: "Female", age: 38 },
  { name: "Frank", sex: "Male", age: 32 },
  { name: "Amy", sex: "Female", age: 29 }
];

let df = DataFrame
  .create(people)
  .group(
      (g)=> { return { sex: g.sex }},
      (a)=> { return { count: a.count() }}
    );
```

```
console.log(df);
```

Example *(Pivoting the Anscombe's Quartet built-in dataset)*

```
let df = DataFrame
  .examples
  .anscombe()
  .group(
    g => { return { observation: g.observation }},
    a => { return JSON.stringify({ x: a.list('x').mean(), y: a.list('y').mean() })},
    p => p.dataset
  )
  .remove('observation')
  .visual('table')
  .attach('root');
```

dataFrame.head(top) â‡’ DataFrame

Gets the top 'n' rows of a DataFrame object.

Kind: instance method of DataFrame
Returns: DataFrame - A DataFrame instance with top 'n' rows only.

Param	Type	Description
top	Number	Top 'n' rows to select.

Example *(Getting the top 'n' rows from a DataFrame Instance)*

```
let titanic = DataFrame.examples.titanic().head(5);
console.log(titanic);
```

dataFrame.count() â‡’ number

Returns the number of rows in the DataFrame.

Kind: instance method of DataFrame
Returns: number - The number of rows in the DataFrame instance.
Example *(Getting the row count of a DataFrame Instance)*

```
let count = DataFrame.examples.titanic.count();
console.log(count);
```

dataFrame.sort(sortFunction, descending) â‡’ DataFrame

Sorts the rows in a DataFrame object based on a sort function.

Kind: instance method of DataFrame
Returns: DataFrame - The sorted DataFrame object

Param	Type
sortFunction	sortFunction
descending	*

Example *(Ranking a dataset using Sort)*

```
let oldest5 = DataFrame
  .examples
  .titanic()
  .map((t) => { return { name: t.name, age: parseFloat(t.age) }})
  .filter((t) => { return !Number.isNaN(t.age) })
  .sort((t) => { return t["age"] }, true)
  .head(5);

console.log(oldest5);
```

dataFrame.join(dataFrame, type, joinFunction, selectFunction) â‡’ DataFrame

Joins the current DataFrame instance (left) to another DataFrame instance (right). Supports left, right, inner and outer join types.

Kind: instance method of DataFrame

Param	Type	Description
dataFrame	DataFrame	The right hand DataFrame instance

Param	Type	Description
type	JoinType	The join type
joinFunction	joinFunction	The join function to compare rows from the left and right DataFrame instances.
selectFunction	mergeFunction	The function to select the reqired values from the 2 joined tables.

Example *(Joining 2 DataFrame Instances)*

```
UI.layout({
  id: 'root',
  direction: 'horizontal',
  children: [
    {id: 'left'},
    {id: 'centre'},
    {id: 'right'}
  ]
});

let sales = DataFrame.create([
    {customer:'A1495', sku: 'BH41', qty: 10, unitPrice: 1.45},
    {customer:'G234', sku: 'HF42', qty: 1, unitPrice: 2.00},
    {customer:'F4824', sku: 'AH52', qty: 5, unitPrice: 1.00},
    {customer:'E472', sku: 'IF14', qty: 20, unitPrice: 1.20},
    {customer:'A2235', sku: 'FI42', qty: 5, unitPrice: 1.80},
    {customer:'J942', sku: 'AV91', qty: 2, unitPrice: 2.50},
    {customer:'B1244', sku: 'FY14', qty: 1, unitPrice: 3},
    {customer:'S95', sku: 'FE56', qty: 5, unitPrice: 5},
    {customer:'D424', sku: 'FE39', qty: 1, unitPrice: 2.50},
    {customer:'P1254', sku: 'DD67', qty: 2, unitPrice: 3.00}
]);

let customers = DataFrame.create([
    {customer:'A1495', name: 'Paul Allen'},
    {customer:'G234', name: 'Tony George'},
    {customer:'F4824', name: 'Dave Farthing'},
    {customer:'E472', name: 'Simone Earl'},
    {customer:'A2235', name: 'Fiona Abbot' },
    {customer:'J942', name: 'Tracy Jones'},
    {customer:'B1244', name: 'Stan Brown'},
    {customer:'S345', name: 'Michael Smith'},
    {customer:'F254', name: 'Dave Firth'},
    {customer:'J1344', name: 'Stuart Jones' }
]);

let join = sales.join(
    customers,
    'outer',
    (left,right) => { return left.customer === right.customer },
    (left,right) => { return {
        customer: right.customer ?? left.customer ?? null,
        name: right.name ?? null,
        sku: left.customer ?? null,
        qty: left.qty ?? null,
        unitPrice: left.unitPrice ?? null
    }}
);

sales.visual('table').attach('left');
customers.visual('table').attach('centre');
join.visual('table').attach('right');
```

dataFrame.list(column) â‡’ List

Returns a List instance based on a single column or calculation from a DataFrame instance.

Kind: instance method of DataFrame

Param	Type	Description
column	string	The column or calculation to return a list for.

Example *(Getting a list of unique values in a column)*

```
let list = DataFrame.examples.titanic().list('pclass');
console.log(list.unique());
```

dataFrame.describe() â‡’ DataFrame

Returns descriptive statistics about the current DataFrame instance.

Kind: instance method of DataFrame

dataFrame.cast(types) â‡’ DataFrame

Changes the types of columns in a DataFrame object.

Kind: instance method of DataFrame

Param	Type
types	object

dataFrame.remove(â€¦columnNames) â‡’ DataFrame

Removes selected columns from a DataFrame object.

Kind: instance method of DataFrame

Param	Type	Description
â€¦columnNames	string	list of columns to remove.

dataFrame.select(â€¦columnNames) â‡’ DataFrame

Selects columns to keep in a dataset. Columns not specified are removed.

Kind: instance method of DataFrame

Param	Type	Description
â€¦columnNames	string	List of columns to keep.

dataFrame.setSlicer(visual, filterFunction)

Adds a visual slicer to the DataFrame slicer context. Each visual can add a single slicer function to this context.

Kind: instance method of DataFrame

Param	Type	Description
visual	Visual	The visual providing the slicer context.
filterFunction	filterFunction	The filter function applied to the DataFrame object.

dataFrame.unsetSlicer(visual)

Removes a slicer originating from a Visual object.

Kind: instance method of DataFrame

Param	Type	Description
visual	Visual	The visual providing the slicer context.

dataFrame.resetSlicers()

Removes all slicer filter functions

Kind: instance method of DataFrame

dataFrame.visual(type, options, filterFunction) â‡’ Visual

Creates a Visual object from a DataFrame object.

Kind: instance method of DataFrame

Param	Type	Description
type	string	The visual type. This visual type must exist in the Visual.library toolbox.
options	*	The configuration for the renderer. The configuration is renderer-specific.
filterFunction	filterFunction	Optional function to filter the data. The filter is only applied to this visual.

dataFrame.clone() â‡’ DataFrame

Clones the DataFrame object.

Kind: instance method of DataFrame

dataFrame.corr() â‡’ DataFrame

Creates a correlation table for all numeric pairs in the DataFrame object.

Kind: instance method of DataFrame
Returns: DataFrame - Correlation for all numerical variable pairs in the DataFrame object.
Example *(Generating the correlation pairs for a DataFrame object)*

```
let iris = DataFrame.examples.iris();
let corr = iris.corr();
console.log(corr);

// Visualise
corr.group(
  g => { return { x: g.x }},
  a => a.list('corr').mean(),
  p => p.y
).visual('table').attach('root');
```

dataFrame.forEach(forEachCallback)

Executes a callback function for each row in the DataFrame object.

Kind: instance method of DataFrame

Param	Type
forEachCallback	forEachCallback

dataFrame.calculate(name, calculation) â‡’ DataFrame

Defines a calculation on the DataFrame object. Unlike the map() function, Calculations are not physically materialised in the DataFrame instance. Instead, they are dynamically evaluated at runtime.

Kind: instance method of DataFrame

Param	Type	Description
name	string	The name of the calculation.
calculation	mapFunction	The calculation callback function.

dataFrame.measure(name, measure) â‡’ DataFrame

Defines a measure on the DataFrame object. Measures aggregate a group of data down to a single value. Measures are normally used to sum, count and otherwise summarise numeric data.

Kind: instance method of DataFrame

Param	Type	Description
name	string	The name of the measure.
measure	mapFunction	The measure callback function.

dataFrame.columnCategory(columnName) â‡’ ColumnCategory

Returns the column category.

Kind: instance method of DataFrame

Param	Type
columnName	*

dataFrame.slicedData() â‡’ DataFrame

Returns the data after slicers have been applied to it. A new DataFrame object is returned with the same calculations and measures as the original DataFrame instance.

Kind: instance method of DataFrame

dataFrame.cube(â€¦columns)

Evaluates a subcube from the DataFrame object. The cube method evaluates any calculations and measures defined in the model

Kind: instance method of `DataFrame`

Param	Type	Description
â€¦columns	`Array`	The columns to include in the subcube.

DataFrame.examples

Built-in example datasets.

Kind: static property of `DataFrame`

- .examples
  - .iris()
  - .titanic()
  - .anscombe()
  - .mtcars()

examples.iris()

The Iris flower data set or Fisher's Iris data set is a multivariate data set introduced by the British statistician, eugenicist, and biologist Ronald Fisher in his 1936 paper The use of multiple measurements in taxonomic problems as an example of linear discriminant analysis. It is sometimes called Anderson's Iris data set because Edgar Anderson collected the data to quantify the morphologic variation of Iris flowers of three related species. Two of the three species were collected in the GaspÃ© Peninsula "all from the same pasture, and picked on the same day and measured at the same time by the same person with the same apparatus". Fisher's paper was published in the journal, the Annals of Eugenics, creating controversy about the continued use of the Iris dataset for teaching statistical techniques today. The data set consists of 50 samples from each of three species of Iris (Iris setosa, Iris virginica and Iris versicolor). Four features were measured from each sample: the length and the width of the sepals and petals, in centimeters. Based on the combination of these four features, Fisher developed a linear discriminant model to distinguish the species from each other.

Kind: static method of `examples`
Example

```
let df = DataFrame.examples.iris();
```

examples.titanic()

The original Titanic dataset, describing the survival status of individual passengers on the Titanic. The titanic data does not contain information from the crew, but it does contain actual ages of half of the passengers. The principal source for data about Titanic passengers is the Encyclopedia Titanica. The datasets used here were begun by a variety of researchers. One of the original sources is Eaton & Haas (1994) Titanic: Triumph and Tragedy, Patrick Stephens Ltd, which includes a passenger list created by many researchers and edited by Michael A. Findlay.

Kind: static method of `examples`
Example

```
let df = DataFrame.examples.titanic();
```

examples.anscombe()

Anscombe's quartet comprises four data sets that have nearly identical simple descriptive statistics, yet have very different distributions and appear very different when graphed. Each dataset consists of eleven (x,y) points. They were constructed in 1973 by the statistician Francis Anscombe to demonstrate both the importance of graphing data before analyzing it and the effect of outliers and other influential observations on statistical properties. He described the article as being intended to counter the impression among statisticians that "numerical calculations are exact, but graphs are rough.

Kind: static method of `examples`
Example

```
let df = DataFrame.examples.anscombe();
```

examples.mtcars()

Mtcars is a built-in dataset included with R. It features 11 features for 32 automobiles (1973-74 models). The data was extracted from the 1974 Motor Trend US magazine.

Kind: static method of `examples`

DataFrame.create(arr) â‡’ DataFrame

Creates a new DataFrame object from a plain Javascript Array object.

Kind: static method of DataFrame
Returns: DataFrame - A DataFrame instance.

Param	Type	Description
arr	Array	The array to create a DataFrame object from.

DataFrame.fetch(url, options) â‡’ DataFrame

Fetches data from a Url. The data must be JSON data.

Kind: static method of DataFrame

Param	Type	Description
url	string	The Url to fetch data from.
options	object	Options used for the fetch.

Example *(Fetching data from an external API)*

```
UI.layout({
  id: 'root',
  direction: 'horizontal',
  children: [
    {
      id: 'left',
      direction: 'vertical',
      children: [
        {id: 'top-left'}, {id: 'bottom-left'}
      ]
    },
    {
      id: 'right'
    }
  ]
});

// Get country population / area data from https://restcountries

let data = await DataFrame.fetch('https://restcountries.eu/rest/v2/all');
let countries = data.select('name','capital','region','subregion','population','area');
Visual.html("<h1>First 10 countries</h1>").attach('right');
countries.head(10).visual('table').attach('right');

// Get 10 largest countries and display in bar chart

let largest = countries.sort(c=>c.area, true).head(10);
largest.visual('bar', {
  border: {
    width: 2,
    color: "gray",
    background: "#e0e8ef",
    radius: 8
  },
  margin: {
    top: 40,
    left: 80,
    right: 20,
    bottom: 60
  },
  title: "Population by Continent",
  fnCategories: c=> { return {region: c.region}},
  fnValues: (c)=> { return {
    population: c.list("population").mean(),
    area: c.list("area").sum()
  }}
}).attach('top-left');

// Pie chart showing population by region.

Visual.html("<h1>Population by Region</h1>").attach('bottom-left');
countries.visual('pie', {
  border: {
    width: 2,
    color: "gray",
    background: "#ddd",
    radius: 8
```

```
  },
  title: "Population by Continent",
  fnCategories: c=> { return { region: c.region }},
  fnValues: (c)=> { return { population: c.list("population").sum() }}
}).attach('bottom-left');
```

DataFrame~mapFunction â‡’ `object`

Callback function to map or transform an object to another object.

Kind: inner typedef of `DataFrame`
Returns: `object` - The transformed object.

Param	Type	Description
currentValue	`object`	The current row in the DataFrame instance.
[index]	`number`	The index of the current row in the DataFrame instance.
[dataFrame]	`DataFrame`	The DataFrame instance that the function was called on.

DataFrame~filterFunction â‡’ `boolean`

Callback function to filter (include / exclude) an object.

Kind: inner typedef of `DataFrame`
Returns: `boolean` - Returns true to keep the object in the DataFrame instance, and false to remove it.

Param	Type	Description
currentValue	`object`	The current row in the DataFrame instance.
[index]	`number`	The index of the current row in the DataFrame instance.
[dataFrame]	`DataFrame`	The DataFrame instance that the function was called on.

DataFrame~groupingFunction â‡’ `object`

Callback function which assigns an object to a group.

Kind: inner typedef of `DataFrame`
Returns: `object` - The callback should return an object representing the properties of the row that should be considered as the 'group' for the row. All the unique values returned for all rows in the DataFrame objects will form the grouping rows of the resulting DataFrame instance.

Param	Type	Description
currentValue	`object`	The current row in the DataFrame instance.
[index]	`number`	The index of the current row in the DataFrame instance.
[dataFrame]	`DataFrame`	The DataFrame instance that the function was called on.

DataFrame~aggregateFunction â‡’ `object`

Callback function which aggregates a group of objects.

Kind: inner typedef of `DataFrame`
Returns: `object` - The callback should return an object representing any aggregrated values of the group. A single object must be returned with one or more properties.

Param	Type	Description
group	`DataFrame`	The current group in the DataFrame.
dataFrame	`DataFrame`	The original DataFrame object that the grouping was performed on.

DataFrame~pivotFunction â‡’ `string`

Callback function which defines column headings for each object.

Kind: inner typedef of `DataFrame`
Returns: `string` - The pivot function should return back a string value. This value will be projected as a column header.

Param	Type	Description
currentValue	`object`	The current row in the DataFrame instance.
[dataFrame]	`DataFrame`	The DataFrame instance that the function was called on.

DataFrame~sortFunction â‡’ `object`

Callback function which defines the value to sort on. Note that this callback function differs from JavaScript's sort callback in that it doesn't actually calculate the sort, but instead simply returns the value to be sorted.

Kind: inner typedef of `DataFrame`
Returns: `object` - The callback should return an object representing the value to sort on.

Param	Type	Description
currentValue	`object`	The current row in the DataFrame instance.
[index]	`number`	The index of the current row in the DataFrame instance.
[dataFrame]	`DataFrame`	The DataFrame instance that the function was called on.

DataFrame~joinFunction â‡’ `boolean`

Callback function which compares 2 objects for equality.

Kind: inner typedef of `DataFrame`
Returns: `boolean` - The function should return true if the objects are considered equal based on the function.

Param	Type	Description
objA	`object`	The first object.
objb	`object`	The second object.

DataFrame~mergeFunction â‡’ `object`

Callback function which merges / returns an object from 2 input objects.

Kind: inner typedef of `DataFrame`
Returns: `object` - The merged / returned object.

Param	Type	Description
objA	`object`	The first object.
objb	`object`	The second object.

DataFrame~forEachCallback : `function`

This callback is a required parameter of the DataFrame map method.

Kind: inner typedef of `DataFrame`

Param	Type	Description
currentValue	`object`	The current row in the DataFrame.
[index]	`number`	The index of the current row in the DataFrame.
[array]	`Array`	The array that the forEach was called on.

DataFrame~measureFunction â‡’ `Number` | `String`

Callback function to calculate a measure based on a group of data.

Kind: inner typedef of `DataFrame`
Returns: `Number` | `String` - The aggregated value.

Param	Type	Description
currentGroup	`DataFrame`	The current group in the DataFrame instance.
[index]	`number`	The index of the current group in the DataFrame instance.
[dataFrame]	`DataFrame`	The DataFrame instance that the function was called on.

List

A List instance represents a single column from a DataFrame. Univariate analysis can be performed on a List instance. List instances are also used to summarise data.

Kind: global class

- List
  - .count() â‡’ `number`
  - .sum() â‡’ `number`
  - .min() â‡’ `number`
  - .max() â‡’ `number`
  - .mean() â‡’ `number`
  - .percentile(percentile) â‡’ `number`
  - .values() â‡’ `List`
  - .unique() â‡’ `List`
  - .type() â‡’ `string`
  - .mode() â‡’ `Array`
  - .var() â‡’ `number`
  - .std() â‡’ `number`
  - .corr(list) â‡’ `number`

list.count() â‡’ `number`

Returns the number of items in the list including null values.

Kind: instance method of `List`
Returns: `number` - The count of items in the list.
Example *(Getting the number of records in a dataset)*

```
let titanic = DataFrame.examples.titanic();
alert(titanic.column("age").count());
```

list.sum() â‡’ `number`

Returns the sum of items in a list. Nulls are ignored.

Kind: instance method of `List`
Returns: `number` - Returns the sum of a list.
Example *(Getting the sum of a list)*

```
let titanic = DataFrame.examples.titanic();
alert(titanic.column("survived").sum());
```

list.min() â‡’ `number`

Returns the minimum value of items in a list. Nulls are ignored.

Kind: instance method of `List`
Returns: `number` - Returns the minimum value in a list.
Example *(Getting the minimum value of a list)*

```
let titanic = DataFrame.examples.titanic();
alert(titanic.column("survived").min());
```

list.max() â‡’ `number`

Returns the maximum value of items in a list. Nulls are ignored.

Kind: instance method of `List`
Returns: `number` - Returns the maximum value in a list.
Example *(Getting the maximum value of a list)*

```
let titanic = DataFrame.examples.titanic();
alert(titanic.column("survived").max());
```

list.mean() â‡’ `number`

Returns the mean value of items in a list. Nulls are ignored.

Kind: instance method of `List`
Returns: `number` - Returns the mean value in a list.
Example *(Getting the mean value of a list)*

```
let titanic = DataFrame.examples.titanic();
alert(titanic.column("survived").mean());
```

list.percentile(percentile) â‡’ `number`

Returns the specified percentile of a list of numbers. Nulls are ignored.

Kind: instance method of `List`
Returns: `number` - Returns the corresponding percentile value.

Param	Type	Description
percentile	`number`	The percentile value between 0 and 100.

Example *(Getting the Q1 value of a list of values)*

```
let titanic = DataFrame.examples.titanic().cast({age: 'float'});
alert(titanic.list("age").percentile(25));
```

list.values() â‡’ `List`

Returns a list of non-null values. Duplicates are included.

Kind: instance method of `List`
Returns: `List` - All non-null values (duplicates included).
Example *(Getting a list of non-null values)*

```
let titanic = DataFrame.examples.titanic().cast({age: 'float'});
alert(titanic.list("age").values());
```

list.unique() â‡’ `List`

Returns a unique list of non-null values. Duplicates are excluded.

Kind: instance method of `List`
Returns: `List` - Unique list of non-null values.
Example *(Getting a unique list of non-null values)*

```
let titanic = DataFrame.examples.titanic().cast({age: 'float'});
alert(titanic.list("age").unique());
```

list.type() â‡’ `string`

Gets the type of the list. The Javascript type of the first row is used.

Kind: instance method of `List`
Returns: `string` - The type of the list.
Example *(Getting the type of a list)*

```
let titanic = DataFrame.examples.titanic().cast({age: 'float'});
alert(titanic.list("age").unique());
```

list.mode() â‡’ `Array`

Returns the most frequent value(s). Up to 5 mode values are permitted.

Kind: instance method of `List`
Returns: `Array` - The list of most frequently occuring value(s).
Example *(Getting the mode of a list)*

```
let values = new List([1,5,3,7,3,7,8,12,15]);
alert(values.mode());
```

list.var() â‡’ `number`

Calculates the variance of a list of values.

Kind: instance method of `List`
Returns: `number` - The non-biased variance.
Example *(Getting the variance of ages on the titanic)*

```
let variance = DataFrame
  .examples
  .titanic()
  .cast({age: 'float'})
  .list('age')
  .var();
alert(variance);
```

list.std() â‡’ `number`

Calculates the standard deviation of a list of values.

Kind: instance method of `List`
Returns: `number` - The non-biased standard deviation.
Example *(Getting the variance of ages on the titanic)*

```
let std = DataFrame
  .examples
  .titanic()
  .cast({age: 'float'})
  .list('age')
  .std();
alert(std);
```

list.corr(list) â‡’ `number`

Calculates the correlation to another List object

Kind: instance method of `List`
Returns: `number` - The correlation value between -1 and 1.

Param	Type	Description
list	`List`	List object with which to calculate the correlation.

Example *(Calculating the correlation between 2 List objects)*

```
let iris = DataFrame.examples.iris();
let sepal_length_cm = iris.list('sepal_length_cm');
let petal_length_cm =  iris.list('petal_length_cm');
alert(sepal_length_cm.corr(petal_length_cm));
```

UI

Controls rendering to the output section.

Kind: global class

- UI
  - *static*
    - .panels
    - .reset()
    - .clear(id)
    - .layout(panels, parentId)
    - .content(content, id)
  - *inner*
    - ~Panel : `Object`

UI.panels

The collection of panels in the current output. Each panel can hold zero or more visuals.

Kind: static property of `UI`

UI.reset()

Clears / resets the output completely.

Kind: static method of `UI`
Example *(Clearing the UI output pane)*

```
UI.reset();
```

UI.clear(id)

Clears a single panel. Typically used internally to redraw parts of the output when data is being interactively sliced.

Kind: static method of `UI`

Param	Type	Description

Param	Type	Description
id	`String`	The id of the panel to clear.

UI.layout(panels, parentId)

Creates a layout using panels. Panels are containers for visuals. A panel layout is typically used to create a dashboard or report.

Kind: static method of `UI`

Param	Type	Default	Description
panels	`Array.<Panel>`		An array of panels. A single panel can also be specified.
parentId	`String`		The parent id of the element to place the panels in.

Example *(Creating a simple layout)*

```
UI.layout({
  id: 'root',
  direction: 'vertical',
  children: [
    {
      id: 'top',
      size: 1
    },
    {
      id: 'middle',
      size: 1
    },
    {
      id: 'bottom',
      size: 2,
      children: [
        {
          id: 'bottom-left'
        },
        {
          id: 'bottom-right'
        }
      ]
    }
  ]
});
```

UI.content(content, id)

Writes content and visuals to an output panel. Note this method should not be used by users. To display visuals, use the Visual.attach() method to attach the visual to the DOM.

Kind: static method of `UI`

Param	Type	Default
content	`object`	
id	`string`	`null`

UI~Panel : `Object`

A definition of a single panel in the output pane.

Kind: inner typedef of `UI`
Properties

Name	Type	Description
id	`String`	The id of the panel. Must be globally unique.
size	`Number`	The relative size of the panel. The number has no dimensions and is relative to its sibling sizes.
alignment	`PanelAlignment`	The alignment of child panels within this panel.
fit	`PanelFit`	Specifies how child items / visuals are fitted within this panel.
children	`Array.<Panel>`	Optional array of child panel objects

Visual

A visual represents any visual component rendered in the output, for example tables and charts.

Kind: global class

- Visual
  - new Visual(type, dataFrame, options, filterFunction)
  - *instance*
    - .attach(panelId)
    - .slicedData() â‡’ `Array`
    - .setState(key, value)
    - .node() â‡’ `Node`
    - .render()
  - *static*
    - .library
      - .box(visual) â‡’ `Node`
      - .column(visual) â‡’ `Node`
      - .crosstab(visual) â‡’ `Node`
      - .hist(visual) â‡’ `Node`
      - .html(visual) â‡’ `Node`
      - .pairs(visual) â‡’ `Node`
      - .pie(visual) â‡’ `Node`
      - .scatter(visual) â‡’ `Node`
      - .slicer(visual) â‡’ `Node`
      - .table(visual) â‡’ `Node`
    - .nextId
    - .html()
  - *inner*
    - ~OptionsMargin : `object`
    - ~OptionsBorder : `object`
    - ~OptionsBase : `object`
    - ~OptionsAxes : `object`
    - ~OptionsBindingRule : `object`
    - ~ColumnOptions : `OptionsBase`
    - ~OptionsCrosstab : `OptionsBase`
    - ~ScatterOptions : `OptionsBase`
    - ~slicerOptions : `Object`

new Visual(type, dataFrame, options, filterFunction)

Creates a new visual. If the visual is created with a dataFrame object, the visual is 'data-bound'. Otherwise the visual is 'non-data-bound' or static. Data-bound visuals automatically update whenver the underlying data changes. Visuals are configured using an options object.

Param	Type	Description
type	`string`	The type of visual from the visual library.
dataFrame	`DataFrame`	The DataFrame instance that is bound to the visual.
options	`OptionsBase`	Configuration for the visual. This is visual-type specific, but there are standard configuration options that all visuals have.
filterFunction	`filterFunction`	The function to filter the data. The filter will be applied to this visual only.

visual.attach(panelId)

Attaches the visual to a panel in the output.

Kind: instance method of `Visual`

Param	Type	Description
panelId	`String`	The id of the panel to attach the visual to.

visual.slicedData() â‡’ `Array`

Returns the data after all appropriate filters / contexts have been applied. All visuals should get data from this function only.

Kind: instance method of `Visual`
Returns: `Array` - Data is returned in native Javascript Array format, which is more suited to libraries like D3.

visual.setState(key, value)

Sets the internal state of the visual.

Kind: instance method of `Visual`

Param	Type
key	
value	*

visual.node() â‡’ Node

Returns an orphaned Html Node element which can be manually placed into the DOM.

Kind: instance method of Visual

visual.render()

Redraws the current visual and all other visuals in the same panel. Re-rendering typically occurs when data is sliced via interactive visuals.

Kind: instance method of Visual

Visual.library

Registry of visual types rendering functions. These include data-bound and non-data bound renderers. Render functions are generally not called directly, but are used by the internal framework when rendering visuals to the output panel. The visual type required is typically defined when using the DataFrame.prototype.visual method. Render functions take no parameters. However, they automatically bind 'this' to the Visual, so all data and configuration can be obtained from the parent Visual object.

- To get the data, call this.slicedData()
- To get the configuration options, call this.options
- To add a slicer, call this.dataFrame.setSlicer(this,)
- To remove a slicer, call this.dataFrame.unsetSlicer(this) The format of the options object is renderer-specific.

Kind: static property of Visual

- .library
  - .box(visual) â‡’ Node
  - .column(visual) â‡’ Node
  - .crosstab(visual) â‡’ Node
  - .hist(visual) â‡’ Node
  - .html(visual) â‡’ Node
  - .pairs(visual) â‡’ Node
  - .pie(visual) â‡’ Node
  - .scatter(visual) â‡’ Node
  - .slicer(visual) â‡’ Node
  - .table(visual) â‡’ Node

library.box(visual) â‡’ Node

Draws a boxplot diagram. Boxplot diagrams are useful for showing the 5-number summary of a continuous variable. The function will automatically include a boxplot for every numeric variable in the DataFrame object.

Kind: static method of library

Param	Type	Description
visual	Visual	The Visual object used for rendering.

Example *(Displaying a boxplot diagram for continuous variables in the iris dataset)*

```
DataFrame
  .examples
  .iris()
  .visual(
    'box',
    {
      binding: {
        column: [
          'sepal_width_cm',
          'sepal_length_cm',
          'petal_width_cm',
          'petal_length_cm'
        ]
      }
    }
  )
  .attach('root');
```

library.column(visual) â‡’ Node

Renders a column chart or grouped column chart. Column charts should be used to show a distribution of data points, or show comparisons between different categories of data. Bars are vertically aligned. For horizontally-aligned bars, refer to the bar visual type. For configuration, refer to: ColumnOptions.

Kind: static method of library

Param	Type	Description
visual	Visual	The Visual object used for rendering.

Example *(Creating a grouped column chart)*

```
let model = DataFrame
  .examples
  .titanic()
  .measure('passengers', (g, i, df) => g.count());

model
  .visual(
    'column',
    {
      title: 'Passengers on the Titanic by Embarked & Sex',
      background: '#abcdef',
      binding: {
        column: 'embarked',
        row: 'sex',
        value: 'passengers'
      }
    }
  )
  .attach('root');
```

Example *(Law of Large Numbers)*

```
let roll = () => Math.floor(Math.random() * 6) + 1;
let attempts = prompt("Enter number of dice throws (1 - 1,000,000):");
let data = [];

for (let i = 0; i < attempts; i++) {
  data.push({roll: roll()});
}

let df = DataFrame
  .create(data)
  .measure('count', (g, i, df) => g.count());

df
  .visual(
    'column'
    , {
      title: `Results of ${attempts} Throws:`,
      binding: {
        column: 'roll',
        value: 'count'
      }
    }
  )
  .attach('root');
```

library.crosstab(visual) â‡’ Node

Generates a crosstab, matrix or contingency table allowing relationships between multiple categorical variables to be viewed. For configuration, refer to: OptionsCrosstab.

Kind: static method of library

Param	Type	Description
visual	Visual	The Visual object used for rendering.

Example *(Creating a contingency table of Titanic survival rates)*

```
let data = DataFrame.examples.titanic();

data.measure('passengers', (g, i, df) => g.count());
data.measure('survival', (g, i, df) => g.filter(r => r.survived===1).count() / g.count());

data
  .visual('crosstab', {
```

```
    binding: {
      columns: ['pclass'],
      rows: ['sex'],
      values: ['passengers', 'survival']
    }
}).attach('root');
```

library.hist(visual) â‡’ `Node`

Renders a histogram. Histograms display the display the frequency distribution of a continuous variable using bins.

Kind: static method of `library`

Param	Type	Description
visual	`Visual`	The Visual object used for rendering.

Example *(Displaying a histogram)*

```
DataFrame
  .examples
  .titanic()
  .cast({age: 'float'})
  .visual(
    'hist',
    {
      binding: {
        column: 'age'
      }
    }
  )
  .attach("root");
```

library.html(visual) â‡’ `Node`

Renderer that renders static HTML content.

Kind: static method of `library`

Param	Type	Description
visual	`Visual`	The Visual object used for rendering.

Example *(Displaying HTML content)*

```
Visual.html('hello world').attach('root');
```

library.pairs(visual) â‡’ `Node`

Creates scatterplot matrix showing relationship between numerical variables in a DataFrame object.

Kind: static method of `library`

Param	Type	Description
visual	`Visual`	The Visual object used for rendering.

Example *(Creating a scatterplot matrix)*

```
DataFrame
  .examples
  .iris()
  .visual('pairs')
  .attach('root');
```

library.pie(visual) â‡’ `Node`

Renders a pie chart.

Kind: static method of `library`

Param	Type	Description
visual	`Visual`	The Visual object used for rendering.

Example *(Displaying a pie chart)*

```
DataFrame
  .examples
  .titanic()
  .measure('passengers',(g,i,df) => g.count())
  .visual(
    'pie',
    {
      title: 'Titanic Passengers' Sex',
      background: '#999',
      border: {width: 1, color: '#333'},
      binding: {
        column: 'sex',
        value: 'passengers'
      }
    }
  )
  .attach("root");
```

library.scatter(visual) â‡’ `Node`

Displays a scatter plot allowing 2 continuous variables to be compared. For configuration, refer to: ScatterOptions.

Kind: static method of `library`

Param	Type	Description
visual	`Visual`	The Visual object used for rendering.

Example *(Displaying a scatterplot)*

```
DataFrame
  .examples
  .titanic()
  .cast({age: 'float', fare: 'float'})
  .calculate('rownum', (r, i, df) => i)
  .measure('age values', (g, i, df) => g.list('age').mean())
  .measure('fare values', (g, i, df) => g.list('fare').mean())
  .visual(
    'scatter',
    {
      binding: {
        column: 'age values',
        row: 'fare values',
        detail: 'rownum'
      }
    }
  )
  .attach("root");
```

Example *(Incorporating a 3rd dimension using color)*

```
let iris = DataFrame
  .examples
  .iris()
  .calculate('rownum', (r, i, df) => i)
  .measure('sepal length values', (g, i, df) => g.list('sepal_length_cm').mean())
  .measure('sepal width values', (g, i, df) => g.list('sepal_width_cm').mean())
  .measure('color value', (g, i, df) => g.list('class').min());

iris
  .visual(
    'scatter',
    {
      binding: {
        column: 'sepal length values',
        row: 'sepal width values',
        detail: 'rownum',
        color: 'color value'
      }
    }
  )
  .attach("root");
```

library.slicer(visual) â‡’ `Node`

Creates an interactive slicer which can slice the bound DataFrame object. Any visuals sharing the DataFrame object will be automatically filtered. For configuration, refer to: slicerOptions

Kind: static method of `library`

Param	Type	Description

Param	Type	Description
visual	Visual	The Visual object used for rendering.

Example *(Adding a slicer for interactive slicing)*

```
UI.layout({
  id: 'root',
  fit: 'width'
});
let data = DataFrame.examples.iris();
data.visual('slicer', {binding: {column: 'class'}}).attach('root');
data.visual('table').attach('root');
```

library.table(visual) â‡’ `Node`

Default table renderer function.

Kind: static method of library

Param	Type	Description
visual	Visual	The Visual object used for rendering.

Example *(Creating a table visual)*

```
let data = DataFrame
  .examples
  .iris()
  .head(20)
  .visual('table')
  .attach('root');
```

Visual.nextId

Stores the next id value. Ensures all visuals are allocated a unique id in the DOM.

Kind: static property of Visual

Visual.html()

Creates a static html-rendered visual. This visual is not bound to any data. Use html visuals for static content like text and abstract shapes which does not change

Kind: static method of Visual

Visual~OptionsMargin : `object`

Configuration for the margins of a visual. Typically used for visuals with x and y axes. The margins refer to the space where the axes labels and title go.

Kind: inner typedef of Visual
Properties

Name	Type	Description
top	`number`	The top margin
right	`number`	The right margin
bottom	`number`	The bottom margin
left	`number`	The left margin

Visual~OptionsBorder : `object`

Configuration of the border around a visual.

Kind: inner typedef of Visual
Properties

Name	Type	Description
width	`number`	The width of the border
color	`string`	The border color
radius	`number`	The border radius

Visual~OptionsBase : `object`

Configuration applicable to all visuals.

Kind: inner typedef of `Visual`
Properties

Name	Type	Description
title	`number`	The title for the visual
width	`number`	The width of the visual
height	`number`	The height of the visual
inline	`boolean`	Set to true for an inline visual. The default is false (block visual). Inline visuals align horizontally within a panel and block visuals align vertically.
margin	`OptionsMargin`	The margin for the visual
background	`string`	The background color for the visual
border	`OptionsBorder`	The border style for the visual
binding	`object`	Data bindings for the visual. Bindings are visual type specific.

Visual~OptionsAxes : `object`

Configuration of a single axis of a visual. Used for visuals with axes (e.g. x & y).

Kind: inner typedef of `Visual`
Properties

Name	Type	Description
display	`boolean`	Toggles the display of the column (x) axis.
title	`string`	The column (x) axis title.
min	`string`	The min value of the column (x) axis.
max	`string`	The max value of the column (x) axis. / /** Configuration of the axes of a visual. Used for visuals with axes (e.g. x & y).
column	`Visual~OptionsAxis`	The column (x) axis configuration.
row	`Visual~OptionsAxis`	The row (y) axis configuration.

Visual~OptionsBindingRule : `object`

Defines a single binding rule. Each binding rule is comprised of a set of properties with each storing a 2 character value defining the binding rule.

The first character specifies the number of fields allowed in the binding:

Value	Meaning
1	One field binding only allowed
?	Zero or one field binding allowed
*****	Zero or more field bindings allowed
+	One or more field bindings allowed

The second character defines the type of binding allowed:

Value	Meaning
m	Measure required
c	Column or calculated field required
a	Any (Measure or Column allowed)

The required bindings on the built-in visuals are:

Type	Column	Row	Value	Color	Size	Detail
bar	X	X	X			
column	X	X				
pie	X		X			
table	X					
crosstag	X	X	X			
hist	X					

Type	Column	Row	Value	Color	Size	Detail
box	X					
scatter	X	X		X	X	X
slicer	X					
pairs	?					
html						

The 6 properties in the binding rule are not all required for all visual types. Typically, a visual may only require one or two of these bindings.

Kind: inner typedef of `Visual`
Properties

Name	Type	Description
column	`string`	The column binding.
row	`string`	The row binding.
value	`string`	The value binding.
detail	`string`	The detail binding.
color	`string`	The color binding.
size	`string`	The size binding.

Visual~ColumnOptions : `OptionsBase`

Additional options for configuring a column visual.

Kind: inner typedef of `Visual`
Properties

Name	Type	Description
binding.column	`string`	The DataFrame field name to project onto the main categories axis of the column chart.
binding.row	`string`	The DataFrame field name to project onto the groupings of the column chart.
binding.value	`Array`	The DataFrame field name(s) to projects onto the values / cells of the crosstab. If the binding.row value is specified, only 1 value field can be entered here.
axes	`object`	The axes configuration.
axes.column	`Visual~OptionsAxis`	The x-axis configuration.
axes.row	`Visual~OptionsAxis`	The y-axis configuration.

Visual~OptionsCrosstab : `OptionsBase`

Additional options object for configuring a crosstab visual

Kind: inner typedef of `Visual`
Properties

Name	Type	Description
binding.columns	`Array`	The DataFrame field names to project onto the columns of the crosstab.
binding.rows	`Array`	The DataFrame field names to projects onto the rows of the crosstab.
binding.values	`Array`	The DataFrame field names to projects onto the values / cells of the crosstab.

Visual~ScatterOptions : `OptionsBase`

Additional options for configuring a scatterplot visual.

Kind: inner typedef of `Visual`
Properties

Name	Type	Description
binding.column	`string`	The DataFrame field name to project onto the x axis of the scatter plot.
binding.row	`string`	The DataFrame field name to project onto the y axis of the scatter plot.
binding.detail	`string`	If specified, the data frame will be grouped by this field. Otherwise, the raw data will be used.
binding.size	`string`	The DataFrame field name to use for the size of the marks on the scatter plot.
binding.color	`string`	The DataFrame field name to use for the color of the marks on the scatter plot.
axes	`object`	The axes configuration.
axes.column	`object`	The x-axis configuration.

Name	Type	Description
axes.column.display	`boolean`	Toggles the display of the column (x) axis.
axes.column.title	`string`	The column (x) axis title.
axes.column.min	`string`	The min value of the column (x) axis.
axes.column.max	`string`	The max value of the column (x) axis.
axes.row	`object`	The y-axis configuration.
axes.row.display	`boolean`	Toggles the display of the row (y) axis.
axes.row.title	`string`	The row (y) axis title.
axes.row.min	`string`	The min value of the row (y) axis.
axes.row.max	`string`	The max value of the row (y) axis.

Visual~slicerOptions : `Object`

Options object for configuring a slicer visual

Kind: inner typedef of `Visual`
Properties

Name	Type	Description
title	`string`	Optional title.
column	`string`	The column used to populate the slicer.

COLUMN_CATEGORY : `enum`

The type of a column in a DataFrame instance.

Kind: global enum
Properties

Name	Type	Default	Description
COLUMN	`ColumnCategory`	`column`	A physical column in the data frame instance.
CALCULATION	`ColumnCategory`	`calculation`	A calculated column in the model. Behaves like a real column.
MEASURE	`ColumnCategory`	`measure`	A measure defined in the model. Measures are used for aggregating data.

JOIN_TYPE : `enum`

Kind: global enum
Properties

Name	Type	Default	Description
LEFT	`JoinType`	`left`	Includes all matching rows, plus unmatched rows from the left-hand DataFrame instance.
RIGHT	`JoinType`	`right`	Includes all matching rows, plus unmatched rows from the right-hand DataFrame instance.
INNER	`JoinType`	`inner`	Includes all matching rows, and exludes all unmatched rows from the left and right DataFrame instances.
OUTER	`JoinType`	`outer`	Includes all matching and unmatched rows from both left and right DataFrame insteances.

PANEL_ALIGNMENT : `enum`

Kind: global enum
Properties

Name	Type	Default	Description
HORIZONTAL	`PanelAlignment`	`horizontal`	Child panels will be aligned horizontally within the parent.
VERTICAL	`PanelAlignment`	`vertical`	Child panels will be aligned vertically within the parent.

PANEL_FIT : `enum`

Kind: global enum
Properties

Name	Type	Default	Description
NONE	`PanelFit`	`none`	Child visuals will not be resized. Scroll bars will be displayed if content overflows the panel.
WIDTH	`PanelFit`	`width`	Child visuals will be resized so the width fits the panel.

Name	Type	Default	Description
HEIGHT	PanelFit	height	Child visuals will be resized so the height fits the panel.
BOTH	PanelFit	both	Child visuals will be resized so both the width and height fits the panel.

isObject(item) â‡’ boolean

Simple object check.

Kind: global function

Param
item

validateBinding(dataFrame, binding, rules)

Validates the options.binding configuration passed into a Visual.

Kind: global function

Param	Type	Description
dataFrame	DataFrame	The DataFrame instance containing the model.
binding	object	The binding object to validate.
rules	Array.<OptionsBindingRule>	The binding validation rules.

doBaseStyles()

Applies base styling on a visual.

Kind: global function

ColumnCategory : String

Kind: global typedef

JoinType : String

Kind: global typedef

PanelAlignment : String

Kind: global typedef

PanelFit : String

Kind: global typedef