Jupyter-Viz

Jupyter Notebook Visualization Tools

1.0.0

26 September 2018

Nathan Carter

Nathan Carter
Email: ncarter@bentley.edu
Homepage: http://nathancarter.github.io
Address: 175 Forest St.
Waltham, MA 02452
USA

Contents

Chapter 1

How to use this package

1.1 Purpose

Since 2017, it has been possible to use GAP in Jupyter through the JupyterKernel package. Output was limited to the ordinary text output GAP produces; charts and graphs were not possible.

In 2018, Martins and Pfeiffer released francy (repository, article), which lets users create graphs of a few types (vertices and edges, line chart, bar chart, scatter chart). It also allows the user to attach actions to the elements of these charts, which result in callbacks to GAP that can update the visualization.

This package aims to make a wider variety of visualizations accessible to GAP users, but does not provide tools for conveniently making such visualizations interactive. Where the francy package excels at interactive visualizations, this package instead gives a broader scope of visualization tools.

This is achieved by importing several existing JavaScript visualization toolkits and exposing them to GAP code, as described later in this manual.

The toolkits currently exposed by this package are listed here.

- AnyChart
- CanvasJS
- ChartJS
- Cytoscape
- D3
- Plotly
- Native HTML `canvas` element
- Plain HTML

1.2 Loading the package into a Jupyter notebook

To import the package into a Jupyter notebook, do so just as with any other GAP package: Ensure that the kernel of the notebook is a GAP kernel, then execute the following code in one of the notebook cells.

Example
```
LoadPackage( "jupyter-viz" );
```

1.3 Creating a visualization

Visualizations of any kind supported by this package are created through one function, `CreateVisualization` (2.1.3). You can view its complete documentation in for details, but examples are given in this section.

Nearly all visualizations in this package are created by passing data to the `CreateVisualization` (2.1.3) function as records describing what to draw. These records are converted into JSON form by the json package, and handed to whichever JavaScript toolkit you have chosen to use for creating the visualization.

1.3.1 Example: AnyChart

The AnyChart website contains documentation on how to create visualizations from JSON data. Following those conventions, we could give AnyChart the following JSON to produce a pie chart.

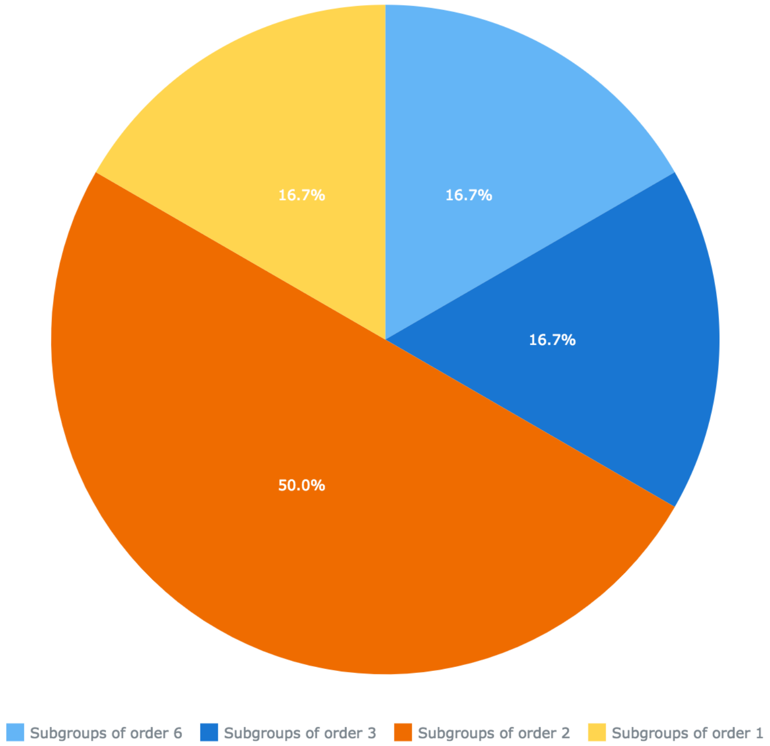
Example
```
{
    "chart" : {
        "type" : "pie",
        "data" : [
            { "x" : "Subgroups of order 6", "value" : 1 },
            { "x" : "Subgroups of order 3", "value" : 1 },
            { "x" : "Subgroups of order 2", "value" : 3 },
            { "x" : "Subgroups of order 1", "value" : 1 }
        ]
    }
}
```

In GAP, the same data would be represented with a record, as follows.

Example
```
myChartData := rec(
    chart := rec(
        type := "pie",
        data := [
            rec( x := "Subgroups of order 6", value := 1 ),
            rec( x := "Subgroups of order 3", value := 1 ),
            rec( x := "Subgroups of order 2", value := 3 ),
            rec( x := "Subgroups of order 1", value := 1 )
        ]
    )
);
```

We can ask GAP, running in a Jupyter notebook, to create a visualization from this data by passing that data directly to `CreateVisualization` (2.1.3). We wrap it in a record that must specify the tool to use (in this case `"anychart"`) and optionally some other details not relevant here.

Example
```
CreateVisualization( rec( tool := "anychart", data := myChartData ) );
```



You can see the results of several examples in the files `in-noteboook-test.pdf` and `in-noteboook-test.ipynb`, in the `tst/` subfolder of this package's installation directory.

If you have the data defining a visualization stored in a `.json` file on disk, you can use the following code rather than rewriting the JSON code into GAP code yourself.

Example
```
CreateVisualization( rec(
    tool := "anychart",
    data := JsonStringToGap( ReadAll( InputTextFile( "your-file.json" ) ) )
) );
```

AnyChart can make a wide variety of charts (area, bar, box, bubble, bullet, column, doughnut, and so on, for over 125 different types and subtypes). Other JavaScript libraries available also have similarly broad capabilities, but we do not include here examples of CanvasJS, ChartJS, or Plotly, because their capabilities and purpose are somewhat similar to that of AnyChart. Though their data formats are different, you can find links to those formats' documentation in the documentation for the function `CreateVisualization` (2.1.3). So instead future sections focus on four other examples that are unlike AnyChart.

1.3.2 Post-processing visualizations

Note that `CreateVisualization` (2.1.3) takes an optional second parameter, a string of JavaScript code to be run once the visualization is complete. For example, if the visualization library did not support a solid black border, but you wanted to add one, you could do so in subsequent code.

Example

```
CreateVisualization(
    sameDataAsAbove, # plus this new second parameter:
    "visualization.style.border = '5px solid black'"
)
```

This holds for any visualization tool, not just AnyChart. In the code given in the second parameter, two variables will be defined for your use: `element` refers to the output cell element in the notebook and `visualization` refers to the visualization that the toolkit you chose created within that output cell (also an HTML element).

1.3.3 Example: Cytoscape

Unlike AnyChart, Cytoscape is for the vertices-and-edges type of graph, not the x-and-y-axes type. A tiny Cytoscape graph (just $A \to B$) is represented by the following JSON.

Example

```
{
    elements : [
        { data : { id : "A" } },
        { data : { id : "B" } },
        { data : { id : "edge", source : "A", target : "B" } }
    ],
    layout : { name : "grid", rows : 1 }
}
```

Cytoscape graphs can also have style attributes not shown here.

Rather than copy this data directly into GAP, let's generate graph data in the same format using GAP code. Here we make a graph of the first 50 positive integers, with $n \to m$ iff $n \mid m$ (ordinary integer divisibility).

Example

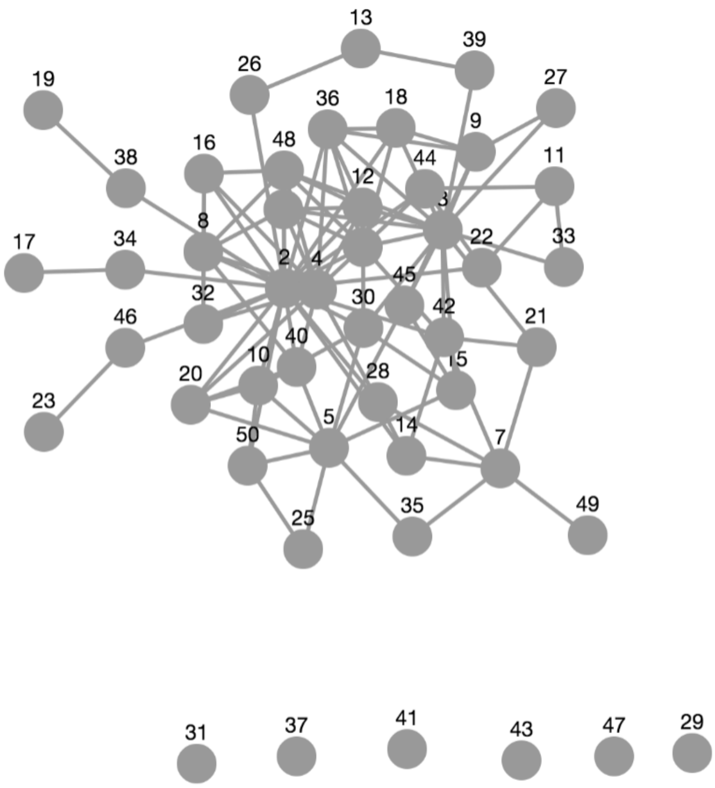
```
N := 50;
elements := [ ];
roots := [ ];
for i in [2..N] do
    Add( elements, rec( data := rec( id := String( i ) ) ) );
    if IsPrime( i ) then
        Add( roots, i );
    fi;
    for j in [2..i-1] do
        if i mod j = 0 then
            Add( elements, rec( data := rec(
                source := String( j ),
                target := String( i ) ) ) );
        fi;
    od;
od;
```

We then need to choose a layout algorithm. The Cytoscape documentation suggests that the "cose" layout works well. Here, we do choose a height (in pixels) for the result, because Cytoscape does not automaticlly resize visualizations to fit their contents. We also set the style for each node to display its ID (which is the integer associated with it).

All the code below comes directly from translating the Cytoscape documentation from JSON form to GAP record form. See that documentation for more details; it is cited in the documentation for the CreateVisualization (2.1.3) function.

Example

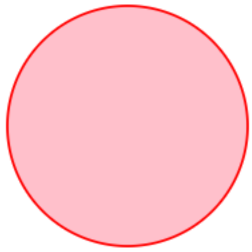
```
CreateVisualization( rec(
    tool := "cytoscape",
    height := 600,
    data := rec(
        elements := elements, # computed in the code above
        layout := rec( name := "cose" ),
        style := [
            rec( selector := "node", style := rec( content := "data(id)" ) )
        ]
    )
) );
```



1.3.4 Example: D3

While D3 is one of the most famous and powerful JavaScript visualization libraries, it does not have a JSON interface. Consequently, we can interact with D3 only through the JavaScript code passed in the second parameter to CreateVisualization (2.1.3). This makes it much less convenient, but we include it in this package for those who need it.

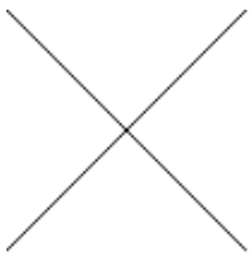
Example
```
CreateVisualization(
    rec( tool := "d3" ),
    """
    // arbitrary JavaScript code can go here to interact with D3, like so:
    d3.select( visualization ).append( "circle" )
        .attr( "r", 50 ).attr( "cx", 55 ).attr( "cy", 55 )
        .style( "stroke", "red" ).style( "fill", "pink" );
    """
);
```



1.3.5 Example: Native HTML Canvas

You can create a blank canvas, then use the existing JavaScript canvas API to draw on it.

Example
```
CreateVisualization(
    rec( tool := "canvas", height := 300 ),
    """
    // visualization is the canvas element
    var context = visualization.getContext( '2d' );
    // draw an X
    context.moveTo( 0, 0 );
    context.lineTo( 100, 100 );
    context.moveTo( 100, 0 );
    context.lineTo( 0, 100 );
    context.stroke();
    """
);
```



1.3.6 Example: Plain HTML

This is the degenerate example of a visualization. It does not create any visualization, but lets you specify arbitrary HTML content instead. It is provided here merely as a convenient way to insert

HTML into the notebook.

Example

```
CreateVisualiation( rec(
    tool := "html",
    data := rec(
        html := "<i>Any</i> HTML can go here.  Tables, buttons, whatever."
    )
) );
```

Chapter 2

Function reference

2.1 Public API

2.1.1 RunJavaScript

▷ `RunJavaScript(`*`script`*`)` (function)

Returns: an object that, if rendered in a Jupyter notebook, will run *`script`* as JavaScript

If evaluated in a Jupyter notebook, its result, when rendered by that notebook, will run the JavaScript code in *`script`*.

When the given code is run, the varible `element` will be defined in its environment, and will contain the output element in the Jupyter notebook corresponding to the code that was just evaluated. The script is free to write to that output element.

This function is not intended for use in the GAP REPL.

2.1.2 LoadJavaScriptFile

▷ `LoadJavaScriptFile(`*`filename`*`)` (function)

Returns: the string contents of the file whose name is given

Interprets the given *`filename`* relative to the `lib/js/` path in the Jupyter-Viz package's installation folder, because that is where this package stores its JavaScript libraries. A `.js` extension will be added to *`filename`* iff needed. A `.min.js` extension will be added iff such a file exists, to prioritize minified versions of files.

If the file has been loaded before in this GAP session, it will not be reloaded, but will be returned from a cache in memory, for efficiency.

If no such file exists, returns `fail` and caches nothing.

2.1.3 CreateVisualization

▷ `CreateVisualization(`*`data[, code]`*`)` (function)

Returns: an object that, if rendered in a Jupyter notebook, will run a script to create the desired visualization

The *`data`* must be a record that will be converted to JSON using GAP's json package.

The second argument is optional, a string containing JavaScript *`code`* to run once the visualization has been created. When that code is run, the variables `element` and `visualization` will be in its environment, the former holding the output element in the notebook containing the visualization, and the latter holding the visualization element itself.

The *data* should have the following attributes.

- tool (required) - the name of the visualization tool to use. Currently supported tools:
  - anychart, whose JSON data format is given here: https://docs.anychart.com/Working_with_Data/Data_From_JSON
  - canvas, that is, a regular HTML canvas element, on which you can draw using arbitrary JavaScript included in the *code* parameter
  - canvasjs, whose JSON data format is given here: https://canvasjs.com/docs/charts/chart-types/
  - chartjs, whose JSON data format is given here: http://www.chartjs.org/docs/latest/getting-started/usage.html
  - cytoscape, whose JSON data format is given here: http://js.cytoscape.org/#notation/elements-json
  - d3, which is loaded into an SVG element in the notebook's output cell, and the caller can call any D3 methods on that element thereafter, using arbitrary JavaScript included in the *code* parameter
  - html, which fills the output element with arbitrary HTML, which the caller should provide as a string in the html field of *data*, as documented below
  - plotly, whose JSON data format is given here: https://plot.ly/javascript/plotlyjs-function-reference/#plotlynewplot
- data (required) - subobject containing all options specific to the content of the visualization, often passed intact to the external JavaScript visualization library. You should prepare this data in the format required by the library specified in the tool field, following the documentation for that library cited above.
- width (optional) - width to set on the output element being created
- height (optional) - similar, but height

Example

```
CreateVisualization( rec(
    tool := "html",
    data := rec( html := "I am <i>SO</i> excited about this." )
), "console.log( 'Visualization created.' );" );
```

2.2 Internal methods

Using the convention common to GAP packages, we prefix all methods not intended for public use with a sequence of characters that indicate our particular package. In this case, we use the JUPVIZ prefix. This is a sort of "poor man's namespacing."

None of these methods should need to be called by a client of this package. We provide this documentation here for completeness, not out of necessity.

2.2.1 JUPVIZAbsoluteJavaScriptFilename

▷ `JUPVIZAbsoluteJavaScriptFilename(`*`filename`*`)` (function)

Returns: a JavaScript filename to an absolute path in the package dir

Given a relative *`filename`*, convert it into an absolute filename by prepending the path to the `lib/js/` folder within the Jupyter-Viz package's installation folder. This is used by functions that need to find JavaScript files stored there.

A `.js` extension is appended if none is included in the given *`filename`*.

2.2.2 JUPVIZLoadedJavaScriptCache

▷ `JUPVIZLoadedJavaScriptCache` (global variable)

A cache of the contents of any JavaScript files that have been loaded from this package's folder. The existence of this cache means needing to go to the filesystem for these files only once per GAP session. This cache is used by `LoadJavaScriptFile` (2.1.2).

2.2.3 JUPVIZFillInJavaScriptTemplate

▷ `JUPVIZFillInJavaScriptTemplate(`*`filename, dictionary`*`)` (function)

Returns: a string containing the contents of the given template file, filled in using the given dictionary

A template file is one containing identifiers that begin with a dollar sign ($). For example, `$one` and `$two` are both identifiers. One "fills in" the template by replacing such identifiers with whatever text the caller associates with them.

This function loads the file specified by *`filename`* by passing that argument directly to `LoadJavaScriptFile` (2.1.2). If no such file exists, returns `fail`. Otherwise, it proceed as follows.

For each key-value pair in the given *`dictionary`*, prefix a `$` onto the key, suffix a newline character onto the value, and then replace all occurrences of the new key with the new value. The resulting string is the result.

The newline character is included so that if any of the values in the *`dictionary`* contains single-line JavaScript comment characters (`//`) then they will not inadvertently affect later code in the template.

2.2.4 JUPVIZRunJavaScriptFromTemplate

▷ `JUPVIZRunJavaScriptFromTemplate(`*`filename, dictionary`*`)` (function)

Returns: the composition of `RunJavaScript` (2.1.1) with `JUPVIZFillInJavaScriptTemplate` (2.2.3)

This function is quite simple, and is just a convenience function

2.2.5 JUPVIZRunJavaScriptUsingRunGAP

▷ `JUPVIZRunJavaScriptUsingRunGAP(`*`jsCode`*`)` (function)

Returns: an object that, if rendered in a Jupyter notebook, will run *`jsCode`* as JavaScript after `runGAP` has been defined

There is a JavaScript function called `runGAP`, defined in the `using-runGAP.js` file distributed with this package. That function makes it easy to make callbacks from JavaScript in a Jupyter notebook to the GAP kernel underneath that notebook. This GAP function runs the given *jsCode* in the notebook, but only after ensuring that `runGAP` is defined globally in that notebook, so that *jsCode* can call `runGAP` as needed.

Here is an example use, from JavaScript, of the `runGAP` function.

Example
```
var calculation = "2^50";
runGAP( calculation + ";", function ( result, error ) {
    if ( result )
        alert( calculation + "=" + result );
    else
        alert( "There was an error: " + error );
} );
```

2.2.6 JUPVIZRunJavaScriptUsingLibraries

▷ JUPVIZRunJavaScriptUsingLibraries(*libraries, jsCode*) (function)

Returns: an object that, if rendered in a Jupyter notebook, will run *jsCode* as JavaScript after all *libraries* have been loaded

There are a set of JavaScript libraries stored in the `lib/js/` subfolder of this package's installation folder. The Jupyter notebook does not, by default, know about any of those libraries. This GAP function runs the given *jsCode* in the notebook, but only after ensuring that all JavaScript files on the list *libraries* have been loaded, so that *jsCode* can make use of the functions and variables that they define.

If the first parameter is given as a string instead of a list of strings, it is treated as a list of just one string.

Example
```
JUPVIZRunJavaScriptUsingLibraries( [ "mylib.js" ],
    "alert( 'My Lib defines foo to be: ' + window.foo );" );
# Equivalently:
JUPVIZRunJavaScriptUsingLibraries( "mylib.js",
    "alert( 'My Lib defines foo to be: ' + window.foo );" );
```

2.3 Representation wrapper

This code is documented for completeness's sake only. It is not needed for clients of this package. Package maintainers may be interested in it in the future.

The JupyterKernel package defines a method `JupyterRender` that determines how GAP data will be shown to the user in the Jupyter notebook interface. When there is no method implemented for a specific data type, the fallback method uses the built-in GAP method `ViewString`.

This presents a problem, because we are often transmitting string data (the contents of JavaScript files) from the GAP kernel to the notebook, and `ViewString` is not careful about how it escapes characters such as quotation marks, which can seriously mangle code. Thus we must define our own type and `JupyterRender` method for that type, to prevent the use of `ViewString`.

The declarations documented below do just that. In the event that `ViewString` were upgraded to more useful behavior, this workaround could probably be removed. Note that it is used explicitly in the `using-library.js` file in this package.

2.3.1 JUPVIZIsFileContents (for IsObject)

▷ `JUPVIZIsFileContents(arg)` (filter)

Returns: `true` or `false`

The type we create is called `FileContents`, because that is our purpose for it (to preserve, unaltered, the contents of a text file).

2.3.2 JUPVIZIsFileContentsRep (for IsComponentObjectRep and JUPVIZIsFileContents)

▷ `JUPVIZIsFileContentsRep(arg)` (filter)

Returns: `true` or `false`

The representation for the `FileContents` type

2.3.3 JUPVIZFileContents (for IsString)

▷ `JUPVIZFileContents(arg)` (operation)

A constructor for `FileContents` objects

Elsewhere, the Jupyter-Viz package also installs a `JupyterRender` method for `FileContents` objects that just returns their text content untouched.

Index