Bringing ipywidgets Support to plotly.py

Jon Mease[‡*]

https://youtu.be/1ndo6C1KWjI

Abstract—Plotly.js is a declarative JavaScript data visualization library built on D3 and WebGL that supports a wide range of statistical, scientific, financial, geographic, and 3-dimensional visualizations. Support for creating Plotly.js visualizations from Python is provided by the plotly.py library. Version 3 of plotly.py integrates ipywidgets support, providing a host of benefits to plotly.py users working in the Jupyter notebook. This paper describes the architecture of this new version of plotly.py, and presents examples of several of these benefits.

Index Terms—ipywidgets, plotly, jupyter, visualization

Introduction

The Jupyter Notebook [KRKP+16] has emerged as the dominant interface for exploratory data analysis and visualization in the Python data science ecosystem. The ipywidgets library [GFC] provides a suite of interactive widgets for use in the Jupyter Notebook, and it serves as a foundation for library authors to build on to create their own custom widgets.

This paper describes our work to bring ipywidgets support to plotly.py version 3. Compared to version 2, plotly.py version 3 brings plotly.py users working in the Jupyter Notebook a host of benefits. Figures already displayed in the notebook may now be updated in-place using property assignment syntax. All properties throughout the entire figure hierarchy are now discoverable using tab completion and documented with informative docstrings. Property values are now fully validated by the Python library and helpful error messages are raised on validation failures. Figure transitions may now be animated. Numpy arrays are now transferred between the Python and JavaScript libraries using a binary serialization protocol for improved performance. Finally, Python callbacks may now be registered for execution upon zoom, pan, click, hover, and data selection events.

Plotly.js Overview

Plotly.js is a JavaScript data visualization library based on D3 and WebGL that supports a wide range of statistical, scientific, financial, geographic, and 3-dimensional visualizations [Inc15]. The library was initially developed by Plotly Inc. as a core component of their commercial visualization offerings. The library was open sourced under the MIT license in 2015 [Ploc], and may now be used fully offline without requiring any interaction with Plotly Inc's commercial infrastructure.

* Corresponding author: jon.mease@jhuapl.edu
‡ Johns Hopkins Applied Physics Laboratory

Copyright © 2018 Jon Mease. This is an open-access article distributed under the terms of the Creative Commons Attribution License, which permits unrestricted use, distribution, and reproduction in any medium, provided the original author and source are credited.

```
{"data": [
    {"type": "bar",
     "y": [2, 3, 1],
     "name": "A",},
    {"type": "scatter",
     "y": [3, 1, 2],
     "name": "B",
     "marker": {"size": 12}}
    ],
"layout": {"xaxis":
           {"range": [-1, 3],
            "tickvals": [0, 1, 2]}}}
```

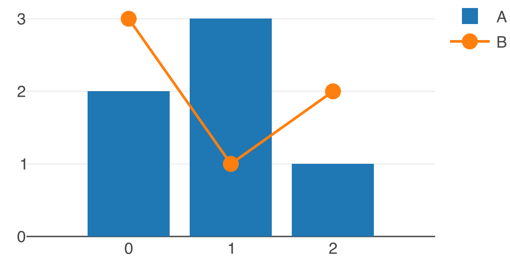


Fig. 1: JSON specification of a basic Plotly.js figure

Data model

Plotly figures are fully defined by a declarative JSON specification. Key components of this specification are shown in the example in Figure 1.

The top-level `'data'` property contains an array of the traces present in the figure. The object representing each trace contains a `'type'` property that identifies the trace type (e.g. `'scatter'`, `'bar'`, `'violin'`, `'mesh3d'`, etc.). The remaining properties are used to configure the trace. As of version 1.37.1, Plotly.js supports 32 distinct trace types covering many statistical, scientific, financial, geographic, and 3-dimensional use-cases.

The top-level `'layout'` property is an object with properties that specify characteristics of the figure that are independent of its traces. These include the figure's size, axis extents, legend styling, background color, and many others.

Of particular interest to this work, the Plotly.js library is capable of exporting a detailed schema corresponding to this JSON specification. The schema includes the names of all valid

```
{"layoutAttributes":
    ...
    "hovermode": {
        "valType": "enumerated",
        "values": [
            "x",
            "y",
            "closest",
            false
        ],
        "description": "Determines ... "
        ...
    }
    ...
}
```

Fig. 2: *Plotly.js schema example for the* `hovermode` *property*

properties and information about their permitted values. This schema is the basis for the Plotly rest API [Ploa] and, as discussed below, this schema enables us to use code-generation to generate a complete Python object hierarchy corresponding to the JSON structure. Figure 2 presents a excerpt of the plot schema describing the `'hovermode'` property of `layout`.

Next, we provide a brief overview of the relevant portions of the Plotly.js API that are used by the new widget library. For more information, including detailed method signatures, see [Plob].

Commands

The following Plotly.js commands are used to create and update figures.

`Plotly.newPlot`
: Create a new figure with initial traces and layout

`Plotly.restyle`
: Update one or more properties of one or more pre-existing traces

`Plotly.relayout`
: Update one or more properties of the figure's layout

`Plotly.update`
: Update both trace and layout properties simultaneously

`Plotly.addTraces`
: Add new traces to an existing figure

`Plotly.deleteTraces`
: Delete select traces from an existing figure

`Plotly.moveTraces`
: Move select traces to a new position in the figure's `data` array

`Plotly.animate`
: Animate property updates in supported trace types

Events

The following events are emitted by Plotly.js figures in response to various kinds of user interaction.

`plotly_restyle`
: Emitted when properties of one or more traces are updated. This may either be the result of a `Plotly.restyle` command or the result of user interaction. For example, clicking on a trace in the legend toggles the trace's visibility in the figure. This visibility state is stored in the top-level `visible` enumeration property on traces.

`plotly_relayout`
: Emitted when properties of the figure's layout are updated. This may either be the result of a `Plotly.relayout` command or the result of user interaction. For example, panning or zooming a figure's axis updates the `'range'` sub-property of the top-level `'xaxis'` and `'yaxis'` layout properties.

`plotly_selected`
: Emitted when a user completes a selection action using the box select or lasso select tools. The event's data contain the indices of the traces from which points were selected and the indices of the selected points themselves. Similar events are also emitted when a user clicks (`plotly_click`), hovers onto (`plotly_hover`), or hovers off of (`plotly_unhover`) points in a trace.

Variables

The current state of a figure is stored in the following four variables.

`data` and `layout`
: These variables store the trace and layout properties explicitly specified by the user.

`_fullData` and `_fullLayout`
: These variables store the full collection of trace and layout properties that are currently in use, whether specified by the user or selected by Plotly.js as defaults.

ipywidgets Overview

The ipywidgets library [GFC] provides a useful collection of interactive widgets (sliders, check boxes, radio buttons, etc.) for use in the Jupyter Notebook and in several other contexts [wida]. For the full list of built-in widgets see [widb].

The integration of graphical widgets into the notebook workflow allows users to configure ad-hoc control panels to interactively sweep over parameters using graphical widget controls, rather than by editing code or writing loops over fixed ranges of values.

The infrastructure behind the built-in ipywidgets is available to library authors and many custom ipywidgets libraries have been developed [Cus]. Three notable data visualization examples include bqplot [CSM+] for 2-dimensional Grammar of Graphics [Wil05] style visualizations, ipyvolume [Bre] for 3-dimensional and volumetric visualizations, and ipyleaflet [CG] for geographic visualization.

The high level architecture, shown in Figure 3, consists of four components: The Python model, the JavaScript model, the JavaScript views, and the Comms interface. These components are described below.

Python Model

The Python model is a Python class that inherits from the `ipywidgets.Widget` superclass and uses the traitlets library [tra] to declare typed attributes that should be synchronized with the JavaScript model.

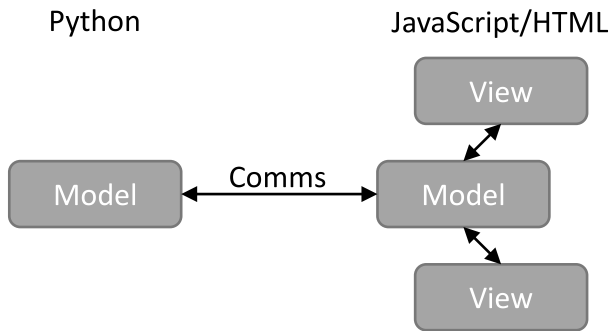


Fig. 3: *High level ipywidgets architecture*

JavaScript Model

The JavaScript model is a JavaScript class that extends the `@jupyter-widgets/base/WidgetModel` class and declares a collection of attributes that match the traitlet declarations in the corresponding Python model.

When used in the Jupyter Notebook, there is a one-to-one relationship between the Python and JavaScript models. The JavaScript model is constructed just after the Python model is constructed, which may be well before the widget is first displayed.

JavaScript View

The JavaScript view (hereafter referred to as "the view" since there is no ambiguity) is a JavaScript class that extends the `@jupyter-widgets/base/WidgetView` class. When used in the notebook, a separate view is constructed each time a model is displayed. Each view has a reference to one JavaScript model, and multiple views may share the same model.

Comms and Synchronization

The Jupyter Comms API provides an abstraction for performing two-way communication between the front-end and the Python kernel, hiding the complexity of the web server, ZeroMQ, and WebSocket implementation details.

The synchronization of the Python and JavaScript models is accomplished using the widget messaging protocol over the Jupyter Comms infrastructure.

A powerful feature of the widget messaging protocol is that it supports the efficient serialization of nested data structures containing binary buffers. This capability is used by ipyvolume [Bre] (and now plotly.py) to transfer Python numpy arrays into JavaScript TypedArrays without ASCII encoding.

New Plotly.py Figure API

In plotly.py version 3, a figure is represented by an instance of the `plotly.graph_objs.Figure` class. A `Figure` instance maintains an internal representation of the figure's JSON specification, and presents a convenient API for creating and updating this specification.

Code generation is used to create a rich hierarchy of Python classes that correspond to the object hierarchy specified in the plot schema described above. Figure 4 presents an example of property tab completion (a), a property docstring (b), and a validation error message (c) for the `'hovermode'` property of layout that is defined by the schema excerpt in Figure 2.

Select components of the new API are described below, and an example of their use is presented in Figure 5.

Construction

If the full specification of the desired figure is known in advance, the specification may be passed directly to the `Figure` constructor as a Python `dict`. This construction process will trigger the validation of all properties and nested properties according to the plot schema. Figure 5 (a) presents an example of constructing a `Figure` with a single `bar` trace.

Property Assignment

A `Figure`'s properties may be configured iteratively after construction using property assignment. Figure 5 (b) presents an example of setting the x-axis range to `[-1, 3]` using property assignment.

Add Traces

A new trace may be added to an existing `Figure` using the `add_{trace}` method that corresponds to the desired trace type. Figure 5 (c) presents an example of adding a new `scatter` trace to a `Figure` instance using the `add_scatter` method.

Batch Update

Multiple properties may be updated simultaneously using a `Figure.batch_update()` context manager. In this case, all property assignments specified inside the `batch_update` context will be executed simultaneously when the context exits. Figure 5 (d) presents an example of assigning four properties across two traces and the layout inside a `batch_update` context.

Reorder Traces

The ordering of traces in the `Figure`'s `data` list determines the order in which the traces are displayed in the legend, and the colors that are chosen for traces by default. The trace order can be updated by assigning to the `data` property a list that contains a permutation of the figure's current traces. Figure 5 (e) presents an example of swapping the order of the `bar` and `scatter` traces.

Delete Traces

Traces may be deleted by omitting them from the list of traces that is assigned to a `Figure`'s `data` property. Figure 5 (f) presents an example of deleting the `bar` trace by assigning a list that contains only the `scatter` trace.

Batch Animate

Multiple properties may be updated simultaneously using a `Figure.batch_animate()` context manager. When applied to a `Figure` instance this works just like the `batch_update` context manager. However, when applied to a `FigureWidget` instance (described below) the Plotly.js library will attempt to smoothly animate the transition to the new property values. Figure 5 (g) presents an example of animating a change in the `Figure`'s x-axis and y-axis range extents.

(a)
```
In [1]: import plotly.graph_objs as go
        fig = go.FigureWidget(
            data=[go.Bar(y=[2, 3, 1])])
```

(b)
```
In [ ]: fig.layout.hover
        fig.layout.hoverdistance
        fig.layout.hoverlabel
        fig.layout.hovermode
```

(c)
```
In [ ]: fig.layout.hovermode
        Type:        property
        String form: <property object at 0x106cf8ea8>
        Docstring:
        Determines the mode of hover interactions.

        The 'hovermode' property is an enumeration that may be specified as:
          - One of the following enumeration values:
                ['x', 'y', 'closest', False]

        Returns
        -------
        Any
```

(d)
```
In [2]: fig.layout.hovermode = 'nearest'
---------------------------------------------------------------------------
ValueError                                Traceback (most recent call last)
<ipython-input-2-5d15804abeb3> in <module>()
----> 1 fig.layout.hovermode = 'nearest'

...
ValueError:
    Invalid value of type 'builtins.str' received for the 'hovermode' property of layout
        Received value: 'nearest'

    The 'hovermode' property is an enumeration that may be specified as:
      - One of the following enumeration values:
            ['x', 'y', 'closest', False]
```

Fig. 4: *Tab completion, documentation, and validation of* `hovermode` *property*

New Plotly.py ipywidgets Implementation

The entry point for the new ipywidgets support is the `plotly.graph_objs.FigureWidget` class. `FigureWidget` is a subclass of `Figure` and, as such, inherits all of the `Figure` characteristics described in the previous section.

Implementing a custom ipywidgets library for Plotly.js presents some architectural challenges. Plotly.js does not expose a model-view separation, each figure stores its own data locally in the figure's root DOM element. This means that each ipywidgets JavaScript view will necessarily be an independent Plotly.js figure instance with its own data. As such, we must take responsibility for keeping the JavaScript model in sync with the state of the Plotly.js figures in each view.

An additional performance-based architectural restriction is that as few properties as possible should be transferred between the Python and JavaScript models. This restriction eliminates solutions that require serialization of the entire plot specification when only a subset of the properties are modified.

The following sections describe our solution to these challenges.

Python to JavaScript Synchronization

Python to JavaScript synchronization is achieved by translating Python `FigureWidget` mutation operations into Plotly.js API commands. These commands, and their associated data, are transferred to the JavaScript model and views using the widget messaging protocol, over the Jupyter Comms infrastructure, as described above. The views are updated by executing the specified Plotly.js command, and the JavaScript model is updated manually in a consistent fashion.

Construction

Construction operations are translated into `Plotly.newPlot` commands. Figure 6 (a) presents an example of the `newPlot` command that results from the construction operation in Figure 5 (a) if the `Figure` class is replaced by `FigureWidget`.

Property Assignment

Trace property assignments are translated into `Plotly.restyle` commands, and layout property assignments are translated into `Plotly.relayout` commands. Figure 6 (b) presents an example of the `relayout` command that results from the property assignment operation in Figure 5 (b).

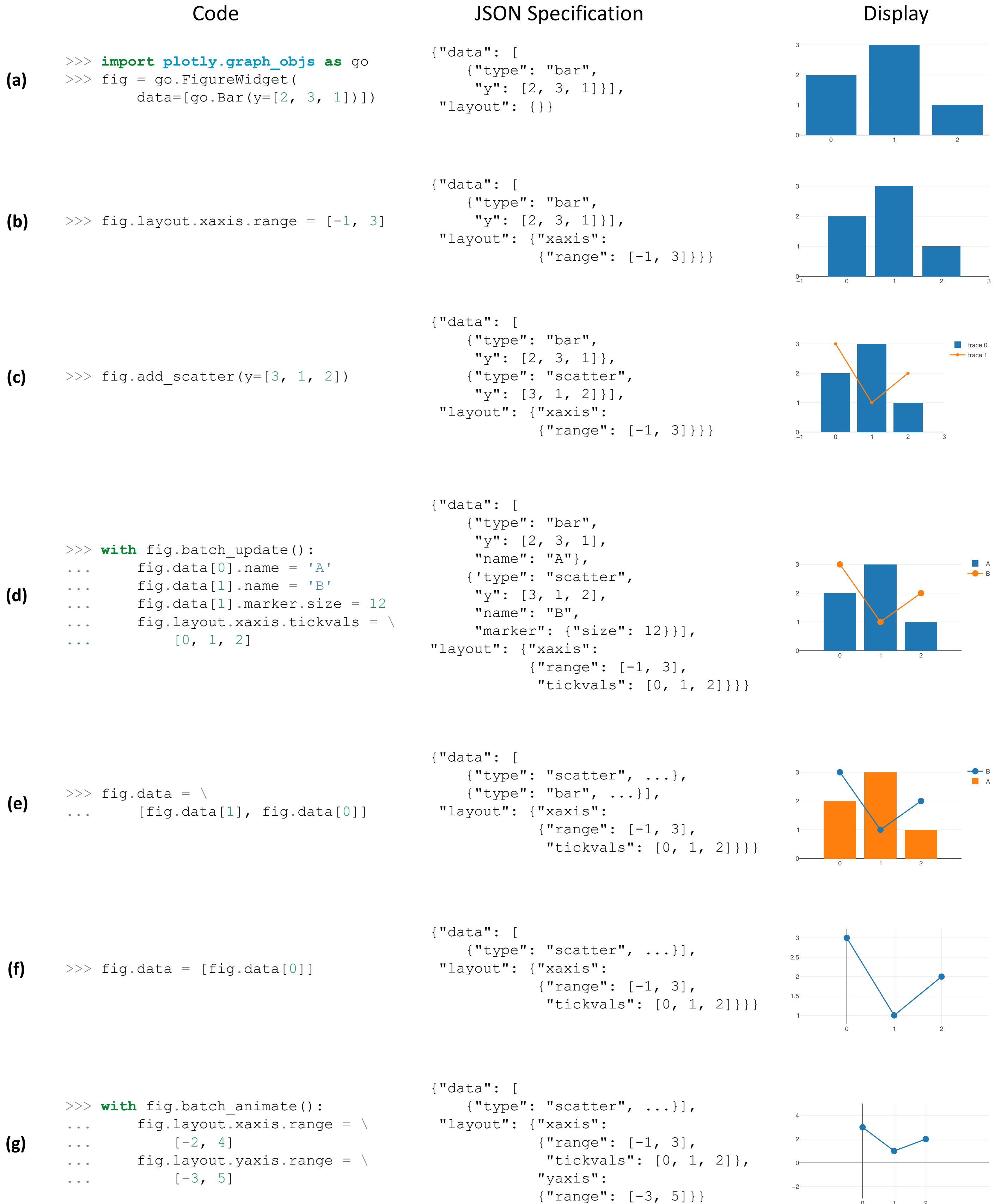


Fig. 5: *New `Figure` API Example*

	Plotly.js Command	Arguments
(a)	`Plotly.newPlot`	`{"data": [ {"type": "bar", "y": [2, 3, 1]}], "layout": {}}`
(b)	`Plotly.relayout`	`{"xaxis.range": [-1, 3]}`
(c)	`Plotly.addTraces`	`{"type": "scatter", "y": [3, 1, 2]}`
(d)	`Plotly.update`	`{"data": {"name": ["A", "B"], "marker.size": [undefined, 12]}, "layout": {"xaxis.tickvals": [0, 1, 2]}}`
(e)	`Plotly.moveTraces`	`{"traceInds": [0, 1], "newTraceIndes": [1, 0]}`
(f)	`Plotly.deleteTraces`	`{"traceInds": [1]}`
(g)	`Plotly.animate`	`{"layout": {"xaxis.range": [-1, 3], "yaxis.range": [-3, 5]}}`

Fig. 6: *Plotly.js commands corresponding to operations in Figure 5 if the* `Figure` *class is replaced by* `FigureWidget`

Add Traces

Add trace operations are translated into `Plotly.addTraces` commands. Figure 6 (c) presents an example of the `addTraces` command that results from the `add_scatter` operation in 5 (c).

Batch Update

Batch update operations are translated in to `Plotly.update` commands. Figure 6 (d) presents an example of the `update` command that results from the `batch_update` operation in 5 (d).

Reorder Traces

Trace reordering operations are translated into `Plotly.moveTraces` commands. Figure 6 (e) presents an example of the `moveTraces` command that results from the `data` assignment operation in 5 (e).

Delete Traces

Trace deletion operations are translated into `Plotly.deleteTraces` commands. Figure 6 (f) presents an example of the `deleteTraces` command that results from the `data` assignment operation in 5 (f).

Batch Animate

Batch animate operations are translated into `Plotly.animate` commands. Figure 6 (g) presents an example of the `animate` command that results from the `batch_animate` operation in 5 (g).

JavaScript to Python Synchronization

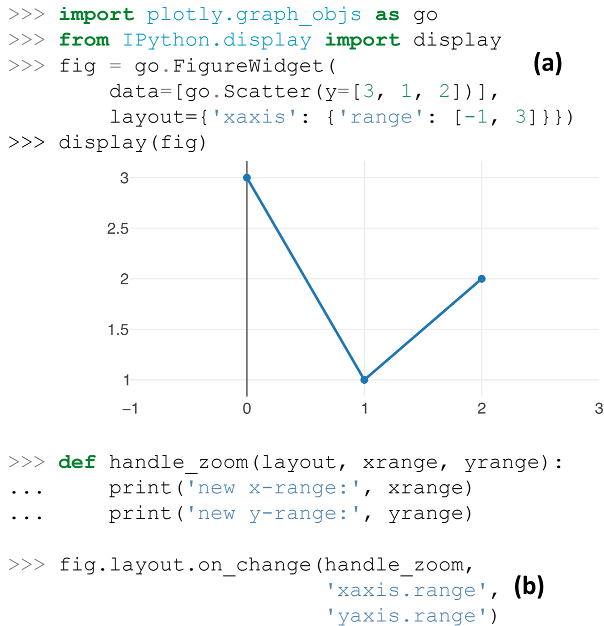
JavaScript to Python synchronization is required when a user interacts with a Plotly.js figure in a view in such a way that the figure's internal specification is modified. For example, the action of zooming or panning a figure causes a modification to the figure's x-axis and y-axis range properties.

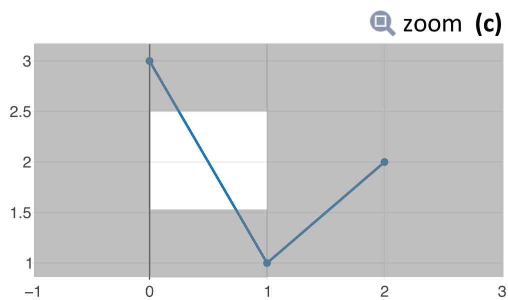
To maintain consistency, views listen for `plotly_restyle` and `plotly_relayout` events and forward these commands to the Python model. The Python model then applies the command to itself and forwards the command to the Java Script model and any additional views.

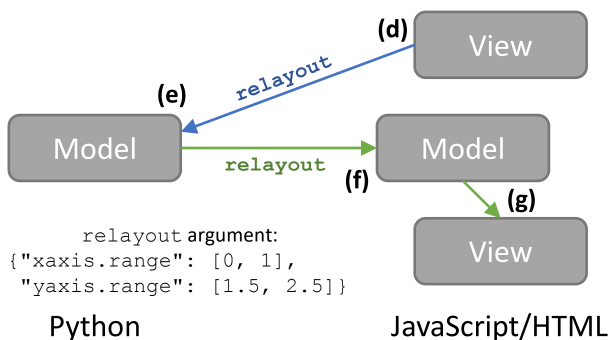
Property change callbacks

Python functions may be registered for execution when particular trace or layout properties are modified by using the `on_change` method. This method is available on all compound objects in the figure hierarchy.

Figure 7 presents an example of constructing and displaying a `FigureWidget` instance (a) and then registering the `handle_zoom` function for execution when the `range` sub-property of either the `xaxis` or the `yaxis` properties is changed (b).








```
new x-range: (0, 1)
new y-range: (1.5, 2.5)
```
(h)

Fig. 7: Zoom property change callback example

Next, the zoom tool is used to select a region that extends from 0 to 1 on the x-axis and from 1.5 to 2.5 on the y-axis (c). The Plotly.js figure that executes the zoom action emits a `plotly_relayout` event (d) which the view forwards to the Python model (e). The Python model applies the update to itself and then sends a `relayout` message to the JavaScript model (f) and any additional JavaScript views (g). Finally, the Python model executes any callback functions registered on the `range` sub-property of `xaxis` or `yaxis` (h).

Point interaction callbacks

As discussed above, a Plotly.js figure emits events when a user interacts with a trace by clicking (`plotly_click`), hovering onto (`plotly_hover`), hovering off of (`plotly_unhover`), or selecting (`plotly_selected`) points. Trace objects in plotly.py now support the registration of Python callbacks to be executed when these events occur.

Figure 8 presents an example of constructing and displaying a `FigureWidget` instance with a `scattergl` trace containing 100,000 normally distributed points (a). The `scattergl` trace is a WebGL optimized version of the SVG-based `scatter` trace used in previous examples.

Trace markers are configured to be colored based on a color scale and a numeric vector. The `cmin` and `cmax` properties specify that `color` values of 0 should be mapped to the bottom of the color scale (light gray for the default scale) and values of 1 should be mapped to the top of the color scale (dark red for the default scale). The color vector is initialized to all zeros so all points are initially light gray in color.

Next, the `brush` function is defined and then registered with the trace for execution when a selection event occurs using the trace's `on_selection` method (b). The first argument to the `brush` function is the trace that was selected (the `scattergl` trace in this case) and the second argument is a list of the indices of the points that were selected.

The box select tool is used to select a rectangular region of points (c). This triggers the execution of the `brush` function. The `brush` function updates the marker's `color` property to be an array where the elements corresponding to selected points have a value of 1 and all other elements have a value of 0. Due to the marker color configuration described above, this causes the selected points to be displayed in dark red.

It is significant to note that even though there are 100,000 points, the time to display the initial figure and the time to update point colors based on a new selection are each less than one second. This latency level is enabled by the efficient transfer of numpy arrays to the JavaScript front-end as binary buffers over the Jupyter Comms interface, and by the WebGL accelerated implementation of the `scattergl` trace.

Default Properties

Plotly.js provides a flexible range of configuration options to control the appearance of a figure's traces and layout, and it will attempt to compute reasonable defaults for properties not specified by the user.

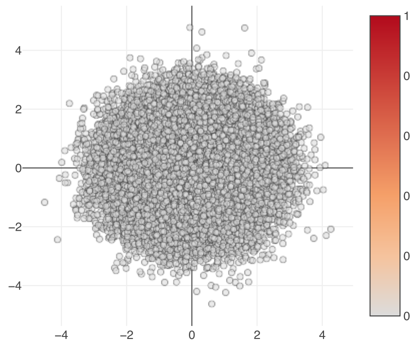
To improve the experience of interactively refining a figure's appearance, it is very helpful to provide the user with the default values of unspecified properties. For example, if a user would like to specify a `scatter` trace marker size that is slightly larger than the default, it is very helpful for the user to know that the default value is 6.

Default property information for traces may be determined by comparing the `data` and `_fullData` variables of the Plotly.js figure. Any property value specified in `_fullData` that is not specified in `data` is considered a default property value. Similarly, the `layout` and `_fullLayout` variables may be used to determine default values for layout properties.

Default properties are transferred from a view to the Python model upon any change to the Plotly.js figure. These default property values are then returned by the Python model during property access when no user specified value is available.

```
>>> import plotly.graph_objs as go
>>> import numpy as np
>>> from IPython.display import display
>>> N = 100000
>>> fig = go.FigureWidget(
...     data = [
...         go.Scattergl(
...             x = np.random.randn(N),
...             y = np.random.randn(N),
...             mode = 'markers',
...             marker={'color': np.zeros(N),
...                     'opacity': 0.6,
...                     'cmin': 0, 'cmax': 1,
...                     'line': {'width': 1},
...                     'showscale' : True})],
...     layout = {'width': 500,
...               'height': 500})
>>> display(fig)
```

(a)




```
>>> def brush(trace, points, *_):
...     inds = np.array(points.point_inds)
...     selected = np.zeros(N)
...     if inds.size:
...         selected[inds] = 1
...     trace.marker.color = selected

>>> fig.data[0].on_selection(brush)
```

(b)

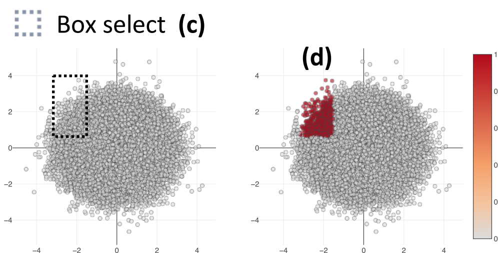


Fig. 8: Data selection and brushing example

Conclusion

The integration of plotly.py version 3 with the ipywidgets library brings a wide range of benefits to plotly.py users working in the Jupyter Notebook. Figure properties are now easily discoverable through the use of tab completion, and they are understandable thanks to the presence of detailed docstrings. This greatly reduces the need for users to interrupt their analysis workflow to consult external documentation resources.

All of these properties may be updated using property assignment syntax and the updates are immediately applied to all of the displayed views of the figure. This allows users to begin the visualization process with simple figures, and then iteratively refine them.

These iterative updates transfer as few properties from Python to JavaScript as possible, and numpy arrays are transferred as binary buffers without ASCII encoding. Combined with the Plotly.js library's performance optimized WebGL trace types, this allows users to create and interactively explore visualizations of data sets with hundreds of thousands of points.

Plotly figures may now be arranged in custom layouts with other ipywidgets, and Python functions may now be registered for execution in response to figure interactions including pan, zoom, click, hover, and selection. These features allow users to create rich dashboards right in the notebook.

In total, the integration of ipywidgets support in plotly.py version 3 dramatically enhances the interactive data visualization experience for plotly.py users working in the Jupyter Notebook, and we are excited to see what the SciPy community will build with these new tools.

Acknowledgements

The development of the ipywidgets integration was supported by the Johns Hopkins Applied Physics Laboratory. The integration of this work into plotly.py version 3 was additionally supported by Plotly Inc.

References

[Bre] Maarten Breddels. maartenbreddels/ipyvolume: 3d plotting for Python in the Jupyter notebook based on IPython widgets using WebGL. URL: https://github.com/maartenbreddels/ipyvolume.

[CG] Sylvain Corlay and Brian Granger. jupyter-widgets/ipyleaflet: A Jupyter - Leaflet.js bridge. URL: https://github.com/jupyter-widgets/ipyleaflet.

[CSM+] Sylvain Corlay, Srinivas Sunkara, Dhruv Madeka, Romain Menegaux, Chakri Cherukuri, and Jason Grout. bloomberg/bqplot: Plotting library for IPython/Jupyter Notebooks. URL: https://github.com/bloomberg/bqplot.

[Cus] Project Jupyter | Widgets. URL: http://jupyter.org/widgets.

[GFC] Jason Grout, Jonathan Frederic, and Sylvain Corlay. ipywidgets: Interactive widgets for the Jupyter Notebook. URL: https://github.com/jupyter-widgets/ipywidgets.

[Inc15] Plotly Technologies Inc. Collaborative data science, 2015. URL: https://plot.ly.

[KRKP+16] Thomas Kluyver, Benjamin Ragan-Kelley, Fernando Pérez, Brian Granger, Matthias Bussonnier, Jonathan Frederic, Kyle Kelley, Jessica Hamrick, Jason Grout, Sylvain Corlay, Paul Ivanov, Damián Avila, Safia Abdalla, and Carol Willing. Jupyter notebooks – a publishing format for reproducible computational workflows. In F. Loizides and B. Schmidt, editors, *Positioning and Power in Academic Publishing: Players, Agents and Agendas*, pages 87 – 90. IOS Press, 2016.

[Ploa] Plotly REST API, v2. URL: https://api.plot.ly/v2/plot-schema.

[Plob] Plotly.js Function Reference. URL: https://plot.ly/javascript/plotlyjs-function-reference/.

[Ploc] Plotly.js Open-Source Announcement. URL: https://plot.ly/javascript/open-source-announcement/.

[tra] Traitlets — traitlets 4.3.2 documentation. URL: https://traitlets.readthedocs.io/en/stable/.

[wida] Embedding Jupyter Widgets in Other Contexts than the Notebook — Jupyter Widgets 7.2.1 documentation. URL: https://ipywidgets.readthedocs.io/en/latest/embedding.html.

[widb] Widget List — Jupyter Widgets 7.2.1 documentation. URL: http://ipywidgets.readthedocs.io/en/latest/examples/Widget%20List.html.

[Wil05] Leland Wilkinson. *The Grammar of Graphics (Statistics and Computing)*. Springer-Verlag, Berlin, Heidelberg, 2005.