ipywidgets Documentation

Release 6.0.0.dev

https://jupyter.org

Oct 12, 2016

CHAPTER 1

Contents

1.1 Installation

Users can install the current version of **ipywidgets** with pip or conda.

1.1.1 With pip

```
pip install ipywidgets
jupyter nbextension enable --py widgetsnbextension
```

When using virtualenv and working in an activated virtual environment, the `--sys-prefix` option may be required to enable the extension and keep the environment isolated (i.e. `jupyter nbextension enable --py widgetsnbextension --sys-prefix`).

1.1.2 With conda

```
conda install -c conda-forge ipywidgets
```

Installing **ipywidgets** with conda will also enable the extension for you.

1.2 Using Interact

The `interact` function (`ipywidgets.interact`) automatically creates user interface (UI) controls for exploring code and data interactively. It is the easiest way to get started using IPython's widgets.

```
In [1]: from __future__ import print_function
        from ipywidgets import interact, interactive, fixed
        import ipywidgets as widgets
```

As of ipywidgets 5.0, only static images of the widgets in this notebook will show on http://nbviewer.ipython.org. To view the live widgets and interact with them, you will need to download this notebook and run it with a Jupyter Notebook server.

1.2.1 Basic `interact`

At the most basic level, `interact` autogenerates UI controls for function arguments, and then calls the function with those arguments when you manipulate the controls interactively. To use `interact`, you need to define a function that you want to explore. Here is a function that prints its only argument `x`.

```
In [2]: def f(x):
            return x
```

When you pass this function as the first argument to `interact` along with an integer keyword argument (`x=10`), a slider is generated and bound to the function parameter.

```
In [3]: interact(f, x=10);
```

examples/../_build/doctrees/nbsphinx/examples_UsingInteract_8_0.png

```
10
```

When you move the slider, the function is called, which prints the current value of `x`.

If you pass `True` or `False`, `interact` will generate a checkbox:

```
In [4]: interact(f, x=True);
```

examples/../_build/doctrees/nbsphinx/examples_UsingInteract_10_0.png

```
True
```

If you pass a string, `interact` will generate a text area.

```
In [5]: interact(f, x='Hi there!');
```

examples/../_build/doctrees/nbsphinx/examples_UsingInteract_12_0.png

```
'Hi there!'
```

`interact` can also be used as a decorator. This allows you to define a function and interact with it in a single shot. As this example shows, `interact` also works with functions that have multiple arguments.

```
In [6]: @interact(x=True, y=1.0)
        def g(x, y):
            return (x, y)
```

examples/../_build/doctrees/nbsphinx/examples_UsingInteract_14_0.png

```
(True, 1.0)
```

1.2.2 Fixing arguments using `fixed`

There are times when you may want to explore a function using `interact`, but fix one or more of its arguments to specific values. This can be accomplished by wrapping values with the `fixed` function.

```
In [7]: def h(p, q):
            return (p, q)
```

When we call `interact`, we pass `fixed(20)` for q to hold it fixed at a value of `20`.

```
In [8]: interact(h, p=5, q=fixed(20));
```

examples/../_build/doctrees/nbsphinx/examples_UsingInteract_19_0.png

```
(5, 20)
```

Notice that a slider is only produced for `p` as the value of `q` is fixed.

1.2.3 Widget abbreviations

When you pass an integer-valued keyword argument of `10` (`x=10`) to `interact`, it generates an integer-valued slider control with a range of $[-10, +3 \times 10]$. In this case, `10` is an *abbreviation* for an actual slider widget:

```
IntSlider(min=-10,max=30,step=1,value=10)
```

In fact, we can get the same result if we pass this `IntSlider` as the keyword argument for `x`:

```
In [9]: interact(f, x=widgets.IntSlider(min=-10,max=30,step=1,value=10));
```

examples/../_build/doctrees/nbsphinx/examples_UsingInteract_23_0.png

```
10
```

This examples clarifies how `interact` proceses its keyword arguments:

1. If the keyword argument is a `Widget` instance with a `value` attribute, that widget is used. Any widget with a `value` attribute can be used, even custom ones.
2. Otherwise, the value is treated as a *widget abbreviation* that is converted to a widget before it is used.

The following table gives an overview of different widget abbreviations:

Keyword argument

Widget

`True` or `False`

Checkbox

`'Hi there'`

Text

`value` or `(min,max)` or `(min,max,step)` if integers are passed

IntSlider

`value` or `(min,max)` or `(min,max,step)` if floats are passed

FloatSlider

`('orange','apple')` or `{'one':1,'two':2}`

Dropdown

You have seen how the checkbox and textarea widgets work above. Here, more details about the different abbreviations for sliders and dropdowns are given.

If a 2-tuple of integers is passed `(min,max)`, an integer-valued slider is produced with those minimum and maximum values (inclusively). In this case, the default step size of `1` is used.

```
In [10]: interact(f, x=(0,4));
```

examples/../_build/doctrees/nbsphinx/examples_UsingInteract_26_0.png

```
2
```

If a 3-tuple of integers is passed `(min,max,step)`, the step size can also be set.

```
In [11]: interact(f, x=(0,8,2));
```

examples/../_build/doctrees/nbsphinx/examples_UsingInteract_28_0.png

```
4
```

A float-valued slider is produced if the elements of the tuples are floats. Here the minimum is `0.0`, the maximum is `10.0` and step size is `0.1` (the default).

```
In [12]: interact(f, x=(0.0,10.0));
```

examples/../_build/doctrees/nbsphinx/examples_UsingInteract_30_0.png

```
5.0
```

The step size can be changed by passing a third element in the tuple.

```
In [13]: interact(f, x=(0.0,10.0,0.01));
```

examples/../_build/doctrees/nbsphinx/examples_UsingInteract_32_0.png

```
4.99
```

For both integer and float-valued sliders, you can pick the initial value of the widget by passing a default keyword argument to the underlying Python function. Here we set the initial value of a float slider to `5.5`.

```
In [14]: @interact(x=(0.0,20.0,0.5))
         def h(x=5.5):
             return x
```

examples/../_build/doctrees/nbsphinx/examples_UsingInteract_34_0.png

```
5.5
```

Dropdown menus are constructed by passing a tuple of strings. In this case, the strings are both used as the names in the dropdown menu UI and passed to the underlying Python function.

```
In [15]: interact(f, x=('apples','oranges'));
```

examples/../_build/doctrees/nbsphinx/examples_UsingInteract_36_0.png

```
'apples'
```

If you want a dropdown menu that passes non-string values to the Python function, you can pass a dictionary. The keys in the dictionary are used for the names in the dropdown menu UI and the values are the arguments that are passed to the underlying Python function.

```
In [16]: interact(f, x={'one': 10, 'two': 20});
```

examples/../_build/doctrees/nbsphinx/examples_UsingInteract_38_0.png

```
20
```

1.2.4 Using function annotations with `interact`

If you are using Python 3, you can also specify widget abbreviations using function annotations.

Define a function with a checkbox widget abbreviation for the argument `x`.

```
In [22]: def f(x:True): # python 3 only
             return x
```

Then, because the widget abbreviation has already been defined, you can call `interact` with a single argument.

```
In [23]: interact(f);
```

examples/../_build/doctrees/nbsphinx/examples_UsingInteract_43_0.png

```
True
```

If you are running Python 2, function annotations can be defined using the `@annotate` function.

```
In [24]: from IPython.utils.py3compat import annotate
```

```
In [25]: @annotate(x=True)
         def f(x):
             return x
```

```
In [26]: interact(f);
```

examples/../_build/doctrees/nbsphinx/examples_UsingInteract_47_0.png

```
True
```

1.2.5 `interactive`

In addition to `interact`, IPython provides another function, `interactive`, that is useful when you want to reuse the widgets that are produced or access the data that is bound to the UI controls.

Here is a function that returns the sum of its two arguments.

```
In [27]: def f(a, b):
             return a+b
```

Unlike `interact`, `interactive` returns a `Widget` instance rather than immediately displaying the widget.

```
In [28]: w = interactive(f, a=10, b=20)
```

The widget is a `Box`, which is a container for other widgets.

```
In [29]: type(w)

Out[29]: ipywidgets.widgets.widget_box.Box
```

The children of the `Box` are two integer-valued sliders produced by the widget abbreviations above.

```
In [30]: w.children

Out[30]: (<ipywidgets.widgets.widget_int.IntSlider at 0x7fc0f88ea828>,
          <ipywidgets.widgets.widget_int.IntSlider at 0x7fc0f918cef0>)
```

To actually display the widgets, you can use IPython's `display` function.

```
In [31]: from IPython.display import display
         display(w)
```

examples/../_build/doctrees/nbsphinx/examples_UsingInteract_59_0.png

```
30
```

At this point, the UI controls work just like they would if `interact` had been used. You can manipulate them interactively and the function will be called. However, the widget instance returned by `interactive` also give you access to the current keyword arguments and return value of the underlying Python function.

Here are the current keyword arguments. If you rerun this cell after manipulating the sliders, the values will have changed.

```
In [32]: w.kwargs

Out[32]: {'a': 10, 'b': 20}
```

Here is the current return value of the function.

```
In [33]: w.result

Out[33]: 30
```

1.2.6 Disabling continuous updates

When interacting with long running functions, realtime feedback is a burden instead of being helpful. See the following example:

```
In [34]: def slow_function(i):
             print(int(i),list(x for x in range(int(i)) if
                         str(x)==str(x)[::-1] and
                         str(x**2)==str(x**2)[::-1]))
             return

In [35]: %%time
         slow_function(1e6)
```

```
1000000 [0, 1, 2, 3, 11, 22, 101, 111, 121, 202, 212, 1001, 1111, 2002, 10001, 10101, 1020
CPU times: user 1.08 s, sys: 0 ns, total: 1.08 s
Wall time: 1.08 s
```

Notice that the output is updated even while dragging the mouse on the slider. This is not useful for long running functions due to lagging:

```
In [36]: from ipywidgets import FloatSlider
         interact(slow_function,i=FloatSlider(min=1e5, max=1e7, step=1e5))
```

examples/../_build/doctrees/nbsphinx/examples_UsingInteract_69_0.png

```
100000 [0, 1, 2, 3, 11, 22, 101, 111, 121, 202, 212, 1001, 1111, 2002, 10001, 10101, 10201,
```

```
Out[36]: <function __main__.slow_function>
```

There are two ways to mitigate this. You can either only execute on demand, or restrict execution to mouse release events.

`__manual`

The `__manual` kwarg of interact allows you to restrict execution so it is only done on demand. A button is added to the interact controls that allows you to trigger an execute event.

```
In [37]: interact(slow_function,i=FloatSlider(min=1e5, max=1e7, step=1e5),__manual=True)
```

examples/../_build/doctrees/nbsphinx/examples_UsingInteract_73_0.png

```
Out[37]: <function __main__.slow_function>
```

`continuous_update`

If you are using slider widgets, you can set the `continuous_update` kwarg to `False`. `continuous_update` is a kwarg of slider widgets that restricts executions to mouse release events.

```
In [38]: interact(slow_function,i=FloatSlider(min=1e5, max=1e7, step=1e5,continuous_update=
```

examples/../_build/doctrees/nbsphinx/examples_UsingInteract_76_0.png

```
100000 [0, 1, 2, 3, 11, 22, 101, 111, 121, 202, 212, 1001, 1111, 2002, 10001, 10101, 10201,
```

```
Out[38]: <function __main__.slow_function>
```

1.2.7 Arguments that are dependent of each other

Arguments that are dependent of each other can be expressed manually using `observe`. See the following example, where one variable is used to describe the bounds of another. For more information, please see the widget events example notebook.

```
In [39]: x_widget = FloatSlider(min=0.0, max=10.0, step=0.05)
         y_widget = FloatSlider(min=0.5, max=10.0, step=0.05, value=5.0)

         def update_x_range(*args):
             x_widget.max = 2.0 * y_widget.value
         y_widget.observe(update_x_range, 'value')

         def printer(x, y):
             print(x, y)
         interact(printer,x=x_widget, y=y_widget)
```

examples/../_build/doctrees/nbsphinx/examples_UsingInteract_79_0.png

```
0.0 5.0
```

```
Out[39]: <function __main__.printer>
```

Index - Next

1.3 Simple Widget Introduction

1.3.1 What are widgets?

Widgets are eventful python objects that have a representation in the browser, often as a control like a slider, textbox, etc.

1.3.2 What can they be used for?

You can use widgets to build **interactive GUIs** for your notebooks.
You can also use widgets to **synchronize stateful and stateless

information** between Python and JavaScript.

1.3.3 Using widgets

To use the widget framework, you need to import `ipywidgets`.

```
In [1]: from ipywidgets import *
```

repr

Widgets have their own display `repr` which allows them to be displayed using IPython's display framework. Constructing and returning an `IntSlider` automatically displays the widget (as seen below). Widgets are displayed inside the widget area, which sits between the code cell and output. You can hide all of the widgets in the widget area by clicking the grey *x* in the margin.

```
In [2]: IntSlider()
```

examples/../_build/doctrees/nbsphinx/examples_WidgetBasics_11_0.png

display()

You can also explicitly display the widget using `display(...)`.

```
In [3]: from IPython.display import display
        w = IntSlider()
        display(w)
```

Multiple display() calls

If you display the same widget twice, the displayed instances in the front-end will remain in sync with each other. Try dragging the slider below and watch the slider above.

```
In [4]: display(w)
```

1.3.4 Why does displaying the same widget twice work?

Widgets are represented in the back-end by a single object. Each time a widget is displayed, a new representation of that same object is created in the front-end. These representations are called views.

Closing widgets

You can close a widget by calling its `close()` method.

```
In [5]: display(w)
```

```
In [6]: w.close()
```

1.3.5 Widget properties

All of the IPython widgets share a similar naming scheme. To read the value of a widget, you can query its `value` property.

```
In [7]: w = IntSlider()
        display(w)
```

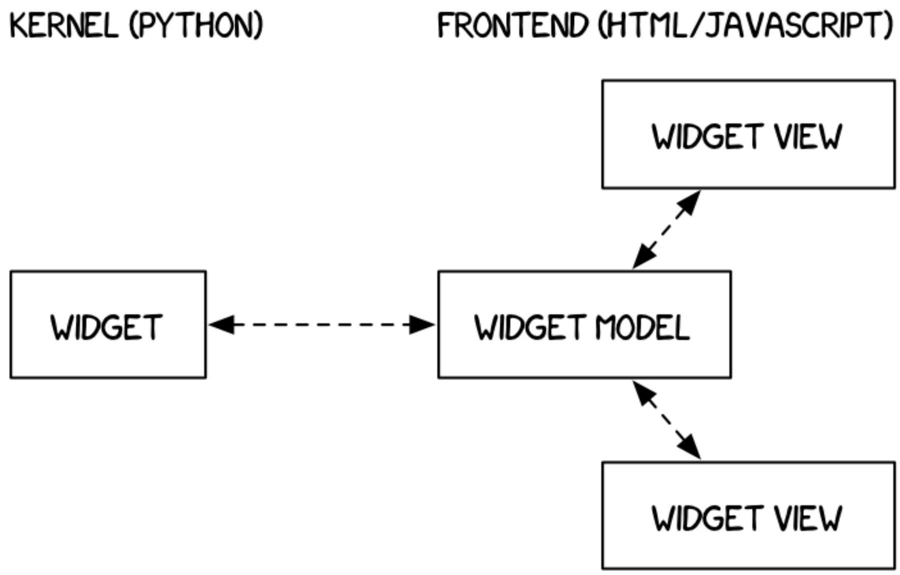


Fig. 1.1: Kernel & front-end diagram

examples/../_build/doctrees/nbsphinx/examples_WidgetBasics_26_0.png

```
In [8]: w.value
```

```
Out[8]: 0
```

Similarly, to set a widget's value, you can set its `value` property.

```
In [9]: w.value = 100
```

Keys

In addition to `value`, most widgets share `keys`, `description`, `disabled`, and `visible`. To see the entire list of synchronized, stateful properties of any specific widget, you can query the `keys` property.

```
In [10]: w.keys
```

```
Out[10]: ['max',
          'font_family',
          'layout',
          'border_radius',
          'disabled',
          'font_style',
          'background_color',
          '_view_module',
          'slider_color',
          'readout',
          'font_size',
```

```
 '_dom_classes',
 '_view_name',
 '_model_name',
 'margin',
 'color',
 'border_color',
 'continuous_update',
 '_model_module',
 'value',
 'description',
 'padding',
 'orientation',
 'border_width',
 'width',
 'step',
 'height',
 'border_style',
 'min',
 'visible',
 '_range',
 'msg_throttle',
 'font_weight']
```

Shorthand for setting the initial values of widget properties

While creating a widget, you can set some or all of the initial values of that widget by defining them as keyword arguments in the widget's constructor (as seen below).

```
In [11]: Text(value='Hello World!', disabled=True)
```

examples/../_build/doctrees/nbsphinx/examples_WidgetBasics_35_0.png

1.3.6 Linking two similar widgets

If you need to display the same value two different ways, you'll have to use two different widgets. Instead of attempting to manually synchronize the values of the two widgets, you can use the `traitlet link` function to link two properties together. Below, the values of two widgets are linked together.

```
In [12]: from traitlets import link
         a = FloatText()
         b = FloatSlider()
         display(a,b)

         mylink = link((a, 'value'), (b, 'value'))
```

examples/../_build/doctrees/nbsphinx/examples_WidgetBasics_38_0.png

Unlinking widgets

Unlinking the widgets is simple. All you have to do is call `.unlink` on the link object. Try changing one of the widgets above after unlinking to see that they can be independently changed.

```
In [13]: mylink.unlink()
```

Index - Next

Index - Back - Next

1.4 Widget List

1.4.1 Complete list

For a complete list of the GUI widgets available to you, you can list the registered widget types. `Widget` and `DOMWidget`, not listed below, are base classes.

```
In [1]: import ipywidgets as widgets
        widgets.Widget.widget_types.values()
```

```
Out[1]: [ipywidgets.widgets.widget_int.IntRangeSlider,
         ipywidgets.widgets.widget_selection.RadioButtons,
         ipywidgets.widgets.widget_int.Play,
         ipywidgets.widgets.widget_controller.Axis,
         ipywidgets.widgets.widget_float.FloatSlider,
         ipywidgets.widgets.widget_int.IntSlider,
         ipywidgets.widgets.widget_image.Image,
         ipywidgets.widgets.widget_selection.SelectMultiple,
         ipywidgets.widgets.widget_string.HTML,
         ipywidgets.widgets.widget_box.Box,
         ipywidgets.widgets.widget_selectioncontainer.Accordion,
         ipywidgets.widgets.widget_button.Button,
         ipywidgets.widgets.widget_bool.Valid,
         ipywidgets.widgets.widget_selection.Dropdown,
         ipywidgets.widgets.widget_selection.SelectionSlider,
         ipywidgets.widgets.widget_box.Proxy,
         ipywidgets.widgets.widget_string.Text,
         ipywidgets.widgets.widget_selection.ToggleButtons,
         ipywidgets.widgets.widget_float.FloatRangeSlider,
         ipywidgets.widgets.widget_color.ColorPicker,
         ipywidgets.widgets.widget_bool.Checkbox,
         ipywidgets.widgets.widget_string.Label,
         ipywidgets.widgets.widget_controller.Controller,
         ipywidgets.widgets.widget_box.PlaceProxy,
         ipywidgets.widgets.widget_float.FloatText,
         ipywidgets.widgets.widget_float.BoundedFloatText,
```

```
 ipywidgets.widgets.widget_link.DirectionalLink,
 ipywidgets.widgets.widget_bool.ToggleButton,
 ipywidgets.widgets.widget_selection.Select,
 ipywidgets.widgets.widget_int.IntProgress,
 ipywidgets.widgets.widget_string.Textarea,
 ipywidgets.widgets.widget_selectioncontainer.Tab,
 ipywidgets.widgets.widget_int.IntText,
 ipywidgets.widgets.widget_float.FloatProgress,
 ipywidgets.widgets.widget_int.BoundedIntText,
 ipywidgets.widgets.widget_link.Link,
 ipywidgets.widgets.widget_controller.Button]
```

1.4.2 Numeric widgets

There are 8 widgets distributed with IPython that are designed to display numeric values. Widgets exist for displaying integers and floats, both bounded and unbounded. The integer widgets share a similar naming scheme to their floating point counterparts. By replacing `Float` with `Int` in the widget name, you can find the Integer equivalent.

FloatSlider

```
In [2]: widgets.FloatSlider(
            value=7.5,
            min=5.0,
            max=10.0,
            step=0.1,
            description='Test:',
        )
```

examples/../_build/doctrees/nbsphinx/examples_WidgetList_8_0.png

Data type cannot be displayed: application/vnd.jupyter.widget

Sliders can also be **displayed vertically**.

```
In [3]: widgets.FloatSlider(
            value=7.5,
            min=5.0,
            max=10.0,
            step=0.1,
            description='Test',
            orientation='vertical',
            readout_format='.3f'
        )
```

examples/../_build/doctrees/nbsphinx/examples_WidgetList_10_0.png

Data type cannot be displayed: application/vnd.jupyter.widget

FloatProgress

```
In [4]: widgets.FloatProgress(
            value=7.5,
            min=5.0,
            max=10.0,
            step=0.1,
            description='Loading:',
        )
```

examples/../_build/doctrees/nbsphinx/examples_WidgetList_12_0.png

Data type cannot be displayed: application/vnd.jupyter.widget

BoundedFloatText

```
In [5]: widgets.BoundedFloatText(
            value=7.5,
            min=5.0,
            max=10.0,
            description='Text:',
        )
```

examples/../_build/doctrees/nbsphinx/examples_WidgetList_14_0.png

Data type cannot be displayed: application/vnd.jupyter.widget

FloatText

```
In [6]: widgets.FloatText(
            value=7.5,
            description='Any:',
        )
```

examples/../_build/doctrees/nbsphinx/examples_WidgetList_16_0.png

Data type cannot be displayed: application/vnd.jupyter.widget

1.4.3 Boolean widgets

There are three widgets that are designed to display a boolean value.

ToggleButton

```
In [7]: widgets.ToggleButton(
            description='Click me',
            value=False,
        )
```

examples/../_build/doctrees/nbsphinx/examples_WidgetList_20_0.png

Data type cannot be displayed: application/vnd.jupyter.widget

Checkbox

```
In [8]: widgets.Checkbox(
            description='Check me',
            value=True,
        )
```

examples/../_build/doctrees/nbsphinx/examples_WidgetList_22_0.png

Data type cannot be displayed: application/vnd.jupyter.widget

Valid

The valid widget provides a read-only indicator.

```
In [9]: widgets.Valid(
            value=True,
        )
```

examples/../_build/doctrees/nbsphinx/examples_WidgetList_24_0.png

Data type cannot be displayed: application/vnd.jupyter.widget

1.4.4 Selection widgets

There are four widgets that can be used to display single selection lists, and one that can be used to display multiple selection lists. All inherit from the same base class. You can specify the **enumeration of selectable options by passing a list**. You can **also specify the enumeration as a dictionary**, in which case the **keys will be used as the item displayed** in the list and the corresponding **value will be returned** when an item is selected.

Dropdown

```
In [10]: from IPython.display import display
         w = widgets.Dropdown(
             options=['1', '2', '3'],
             value='2',
             description='Number:',
         )
         display(w)
```

examples/../_build/doctrees/nbsphinx/examples_WidgetList_28_0.png

Data type cannot be displayed: application/vnd.jupyter.widget

```
In [11]: w.value
```

```
Out[11]: '2'
```

The following is also valid:

```
In [12]: w = widgets.Dropdown(
             options={'One': 1, 'Two': 2, 'Three': 3},
             value=2,
             description='Number:',
         )
         display(w)
```

examples/../_build/doctrees/nbsphinx/examples_WidgetList_31_0.png

Data type cannot be displayed: application/vnd.jupyter.widget

```
In [13]: w.value
```

```
Out[13]: 2
```

Furthermore, if a dropdown contains too man items, a scrollbar is automatically added.

```
In [14]: from IPython.display import display
         w = widgets.Dropdown(
             options=['1', '2', '3', '4', '5', '6', '7', '8', '9'],
             value='2',
             description='Number:',
         )
         display(w)
```

examples/../_build/doctrees/nbsphinx/examples_WidgetList_34_0.png

Data type cannot be displayed: application/vnd.jupyter.widget

```
In [15]: w.value
```

```
Out[15]: '2'
```

The following is also valid:

```
In [16]: w = widgets.Dropdown(
             options={'One': 1, 'Two': 2, 'Three': 3,
                      'Four': 4, 'Five': '5', 'Six': 6,
                      'Seven': 7, 'Eight': 8, 'Nine': 9},
             value=2,
             description='Number:',
         )
         display(w)
```

examples/../_build/doctrees/nbsphinx/examples_WidgetList_37_0.png

Data type cannot be displayed: application/vnd.jupyter.widget

```
In [17]: w.value
```

```
Out[17]: 2
```

RadioButtons

```
In [18]: widgets.RadioButtons(
             description='Pizza topping:',
             options=['pepperoni', 'pineapple', 'anchovies'],
         )
```

examples/../_build/doctrees/nbsphinx/examples_WidgetList_40_0.png

Data type cannot be displayed: application/vnd.jupyter.widget

Select

```
In [19]: widgets.Select(
             description='OS:',
             options=['Linux', 'Windows', 'OSX'],
         )
```

examples/../_build/doctrees/nbsphinx/examples_WidgetList_42_0.png

Data type cannot be displayed: application/vnd.jupyter.widget

SelectionSlider

```
In [20]: widgets.SelectionSlider(
             description='I like my eggs ...',
```

```
            options=['scrambled', 'sunny side up', 'poached', 'over easy'],
        )
```

examples/../_build/doctrees/nbsphinx/examples_WidgetList_44_0.png

Data type cannot be displayed: application/vnd.jupyter.widget

ToggleButtons

```
In [21]: widgets.ToggleButtons(
             description='Speed:',
             options=['Slow', 'Regular', 'Fast'],
         )
```

examples/../_build/doctrees/nbsphinx/examples_WidgetList_46_0.png

Data type cannot be displayed: application/vnd.jupyter.widget

SelectMultiple

Multiple values can be selected with shift and/or ctrl (or command) pressed and mouse clicks or arrow keys.

```
In [22]: w = widgets.SelectMultiple(
             description="Fruits",
             options=['Apples', 'Oranges', 'Pears']
         )
         display(w)
```

examples/../_build/doctrees/nbsphinx/examples_WidgetList_48_0.png

Data type cannot be displayed: application/vnd.jupyter.widget

```
In [23]: w.value
```

```
Out[23]: ()
```

1.4.5 String widgets

There are 4 widgets that can be used to display a string value. Of those, the `Text` and `Textarea` widgets accept input. The `Label` and `HTML` widgets display the string as either Label or HTML respectively, but do not accept input.

Text

```
In [24]: widgets.Text(
             description='String:',
             value='Hello World',
         )
```

examples/../_build/doctrees/nbsphinx/examples_WidgetList_53_0.png

Data type cannot be displayed: application/vnd.jupyter.widget

Textarea

```
In [25]: widgets.Textarea(
             description='String:',
             value='Hello World',
         )
```

examples/../_build/doctrees/nbsphinx/examples_WidgetList_55_0.png

Data type cannot be displayed: application/vnd.jupyter.widget

Label

```
In [26]: widgets.Label(
             value="$$\\frac{n!}{k!(n-k)!} = \\binom{n}{k}$$",
         )
```

examples/../_build/doctrees/nbsphinx/examples_WidgetList_57_0.png

Data type cannot be displayed: application/vnd.jupyter.widget

HTML

```
In [27]: widgets.HTML(
             value="Hello <b>World</b>"
         )
```

examples/../_build/doctrees/nbsphinx/examples_WidgetList_59_0.png

Data type cannot be displayed: application/vnd.jupyter.widget

1.4.6 Button

```
In [28]: widgets.Button(description='Click me')
```

examples/../_build/doctrees/nbsphinx/examples_WidgetList_61_0.png

Data type cannot be displayed: application/vnd.jupyter.widget

1.4.7 Animation widget

The `Play` widget is useful to perform animations by iterating on a sequence of integers with a certain speed.

```
In [29]: play = widgets.Play()
         slider = widgets.IntSlider()
         widgets.jslink((play, 'value'), (slider, 'value'))
         widgets.HBox([play, slider])
```

examples/../_build/doctrees/nbsphinx/examples_WidgetList_64_0.png

Data type cannot be displayed: application/vnd.jupyter.widget

Index - Back - Next

Index - Back - Next

1.5 Widget Events

1.5.1 Special events

```
In [1]: from __future__ import print_function
```

The `Button` is not used to represent a data type. Instead the button widget is used to handle mouse clicks. The `on_click` method of the `Button` can be used to register function to be called when the button is clicked. The doc string of the `on_click` can be seen below.

```
In [2]: import ipywidgets as widgets
        print(widgets.Button.on_click.__doc__)
```

```
Register a callback to execute when the button is clicked.

        The callback will be called with one argument, the clicked button
        widget instance.

        Parameters
        ----------
        remove: bool (optional)
            Set to true to remove the callback from the list of callbacks.
```

Example

Since button clicks are stateless, they are transmitted from the front-end to the back-end using custom messages. By using the `on_click` method, a button that prints a message when it has been clicked is shown below.

```
In [3]: from IPython.display import display
        button = widgets.Button(description="Click Me!")
        display(button)

        def on_button_clicked(b):
            print("Button clicked.")

        button.on_click(on_button_clicked)
```

examples/../_build/doctrees/nbsphinx/examples_WidgetEvents_8_0.png

on_submit

The `Text` widget also has a special `on_submit` event. The `on_submit` event fires when the user hits return.

```
In [4]: text = widgets.Text()
        display(text)

        def handle_submit(sender):
            print(text.value)

        text.on_submit(handle_submit)
```

examples/../_build/doctrees/nbsphinx/examples_WidgetEvents_11_0.png

1.5.2 Traitlet events

Widget properties are IPython traitlets and traitlets are eventful. To handle changes, the `observe` method of the widget can be used to register a callback. The doc string for `observe` can be seen below.

```
In [5]: print(widgets.Widget.observe.__doc__)
```

```
Setup a handler to be called when a trait changes.

        This is used to setup dynamic notifications of trait changes.

        Parameters
        ----------
        handler : callable
            A callable that is called when a trait changes. Its
            signature should be ``handler(change)``, where ``change```is a
            dictionary. The change dictionary at least holds a 'type' key.
            * ``type``: the type of notification.
            Other keys may be passed depending on the value of 'type'. In the
            case where type is 'change', we also have the following keys:
            * ``owner`` : the HasTraits instance
            * ``old`` : the old value of the modified trait attribute
            * ``new`` : the new value of the modified trait attribute
            * ``name`` : the name of the modified trait attribute.
        names : list, str, All
            If names is All, the handler will apply to all traits.  If a list
            of str, handler will apply to all names in the list.  If a
            str, the handler will apply just to that name.
        type : str, All (default: 'change')
            The type of notification to filter by. If equal to All, then all
            notifications are passed to the observe handler.
```

Signatures

Mentioned in the doc string, the callback registered must have the signature `handler(change)` where `change` is a dictionary holding the information about the change.

Using this method, an example of how to output an `IntSlider`'s value as it is changed can be seen below.

```
In [6]: int_range = widgets.IntSlider()
        display(int_range)

        def on_value_change(change):
            print(change['new'])

        int_range.observe(on_value_change, names='value')
```

examples/../_build/doctrees/nbsphinx/examples_WidgetEvents_17_0.png

1.5.3 Linking Widgets

Often, you may want to simply link widget attributes together. Synchronization of attributes can be done in a simpler way than by using bare traitlets events.

Linking traitlets attributes from the server side

The first method is to use the `link` and `dlink` functions from the `traitlets` module.

```
In [7]: import traitlets
```

```
In [8]: caption = widgets.Label(value = 'The values of slider1 and slider2 are synchronized
        sliders1, slider2 = widgets.IntSlider(description='Slider 1'),\
                                     widgets.IntSlider(description='Slider 2')
        l = traitlets.link((sliders1, 'value'), (slider2, 'value'))
        display(caption, sliders1, slider2)
```

examples/../_build/doctrees/nbsphinx/examples_WidgetEvents_22_0.png

```
In [9]: caption = widgets.Label(value = 'Changes in source values are reflected in target1
        source, target1 = widgets.IntSlider(description='Source'),\
                                   widgets.IntSlider(description='Target 1')
        dl = traitlets.dlink((source, 'value'), (target1, 'value'))
        display(caption, source, target1)
```

examples/../_build/doctrees/nbsphinx/examples_WidgetEvents_23_0.png

Function `traitlets.link` and `traitlets.dlink` return a `Link` or `DLink` object. The link can be broken by calling the `unlink` method.

```
In [10]: l.unlink()
         dl.unlink()
```

Linking widgets attributes from the client side

When synchronizing traitlets attributes, you may experience a lag because of the latency due to the roundtrip to the server side. You can also directly link widget attributes in the browser using the link widgets, in either a unidirectional or a bidirectional fashion.

```
In [11]: caption = widgets.Label(value = 'The values of range1 and range2 are synchronized
         range1, range2 = widgets.IntSlider(description='Range 1'),\
                          widgets.IntSlider(description='Range 2')
         l = widgets.jslink((range1, 'value'), (range2, 'value'))
         display(caption, range1, range2)
```

examples/../_build/doctrees/nbsphinx/examples_WidgetEvents_28_0.png

```
In [12]: caption = widgets.Label(value = 'Changes in source_range values are reflected in t
         source_range, target_range1 = widgets.IntSlider(description='Source range'),\
                                       widgets.IntSlider(description='Target
         dl = widgets.jsdlink((source_range, 'value'), (target_range1, 'value'))
         display(caption, source_range, target_range1)
```

examples/../_build/doctrees/nbsphinx/examples_WidgetEvents_29_0.png

Function `widgets.jslink` returns a `Link` widget. The link can be broken by calling the `unlink` method.

```
In [13]: l.unlink()
         dl.unlink()
```

Index - Back - Next

1.6 Layout and Styling of Jupyter widgets

This notebook presents how to layout and style Jupyter interactive widgets to build rich and *reactive* widget-based applications.

1.6.1 The `layout` attribute.

Every Jupyter interactive widget has a `layout` attribute exposing a number of css properties that impact how widgets are laid out.

Exposed css properties

The following properties map to the values of the css properties of the same name (underscores being replaced with dashes), applied to the top DOM elements of the corresponding widget.

** Sizes ** - `height` - `width` - `max_height` - `max_width` - `min_height` - `min_width`

** Display **

- `visibility`
- `display`
- `overflow`
- `overflow_x`
- `overflow_y`

** Box model ** - `border` - `margin` - `padding`

** Positioning ** - `top` - `left` - `bottom` - `right`

** Flexbox ** - `order` - `flex_flow` - `align_items` - `flex` - `align_self` - `align_content` - `justify_content`

Shorthand css properties

You may have noticed that certain css properties such as `margin-[top/right/bottom/left]` seem to be missing. The same holds for `padding-[top/right/bottom/left]` etc.

In fact, you can atomically specify `[top/right/bottom/left]` margins via the `margin` attribute alone by passing the string

```
margin: 100px 150px 100px 80px;
```

for a respectively `top`, `right`, `bottom` and `left` margins of `100`, `150`, `100` and `80` pixels.

Similarly, the `flex` attribute can hold values for `flex-grow`, `flex-shrink` and `flex-basis`. The `border` attribute is a shorthand property for `border-width`, `border-style (required)`, and `border-color`.

Simple examples

The following example shows how to resize a `Button` so that its views have a height of `80px` and a width of `50%` of the available space:

```
In [1]: from ipywidgets import *
```

```
In [2]: s = Button(description='(50% width, 80px height) button')
        s.layout.width = '50%'
        s.layout.height = '80px'
        s
```

```
Widget Javascript not detected.  It may not be installed properly. Did you enable the widge
```

The `layout` property can be shared amongst multiple widgets and assigned directly.

```
In [3]: i = Button(description='Another button with the same layout', layout=s.layout)
        i
```

```
Widget Javascript not detected.  It may not be installed properly. Did you enable the widge
```

Description

You may have noticed that the widget's length is shorter in presence of a description. This because the description is added *inside* of the widget's total length. You **cannot** change the width of the internal description field. If you need more flexibility to layout widgets and captions, you should use a combination with the `Label` widgets arranged in a layout.

```
In [4]: HBox([Label('A too long description'), IntSlider()])
```

```
Widget Javascript not detected.  It may not be installed properly. Did you enable the widge
```

More Styling (colors, inner-widget details)

The `layout` attribute only exposes layout-related css properties for the top-level DOM element of widgets. Individual widgets may expose more styling-related properties, or none. For example, the `Button` widget has a `button_style` attribute that may take 5 different values:

- `'primary'`
- `'success'`
- `'info'`
- `'warning'`
- `'danger'`

besides the default empty string ‘’.

```
In [5]: Button(description='Danger Button', button_style='danger')
```

```
Widget Javascript not detected.  It may not be installed properly. Did you enable the widge
```

Natural sizes, and arangements using HBox and VBox

Most of the core-widgets have - a natural width that is a multiple of `148` pixels - a natural height of `32` pixels or a multiple of that number. - a default margin of `2` pixels

which will be the ones used when it is not specified in the `layout` attribute.

This allows simple layouts based on the `HBox` and `VBox` helper functions to align naturally:

```
In [6]: words = ("Lorem ipsum dolor sit amet, consectetur adipiscing elit, sed " +
                 "do eiusmod tempor incididunt ut labore et dolore magna aliqua.").split('
        items = [Button(description=w) for w in words]
```

```
HBox([VBox([items[0], items[1]]), VBox([items[2], items[3]])])
```

```
Widget Javascript not detected.  It may not be installed properly. Did you enable the widge
```

Latex

Widgets such as sliders and text inputs have a description attribute that can render Latex Equations. The `Label` widget also renders Latex equations.

```
In [7]: IntSlider(description='$\int_0^t f$')
```

```
Widget Javascript not detected.  It may not be installed properly. Did you enable the widge
```

```
In [8]: Label(value='$e=mc^2$')
```

```
Widget Javascript not detected.  It may not be installed properly. Did you enable the widge
```

Number formatting

Sliders have a readout field which can be formatted using Python's *Format Specification Mini-Language*: https://docs.python.org/3/library/string.html#format-specification-mini-language. If the space available for the readout is too narrow for the string representation of the slider value, a different styling is applied to show that not all digits are visible.

1.6.2 The Flexbox layout

In fact, the `HBox` and `VBox` helpers used above are functions returning instances of the `Box` widget with specific options.

The `Box` widgets enables the entire CSS Flexbox spec, enabling rich reactive layouts in the Jupyter notebook. It aims at providing an efficient way to lay out, align and distribute space among items in a container.

Again, the whole Flexbox spec is exposed via the `layout` attribute of the container widget (`Box`) and the contained items. One may share the same `layout` attribute among all the contained items.

Aknowledgement

The following tutorial on the Flexbox layout follows the lines of the article *A Complete Guide to Flexbox* by Chris Coyier available here: https://css-tricks.com/snippets/css/a-guide-to-flexbox/

Basics and terminology

Since flexbox is a whole module and not a single property, it involves a lot of things including its whole set of properties. Some of them are meant to be set on the container (parent element, known as "flex container") whereas the others are meant to be set on the children (said "flex items").

If regular layout is based on both block and inline flow directions, the flex layout is based on "flex-flow directions". Please have a look at this figure from the specification, explaining the main idea behind the flex layout.

Basically, items will be laid out following either the `main axis` (from `main-start` to `main-end`) or the `cross axis` (from `cross-start` to `cross-end`).

- `main axis` - The main axis of a flex container is the primary axis along which flex items are laid out. Beware, it is not necessarily horizontal; it depends on the flex-direction property (see below).

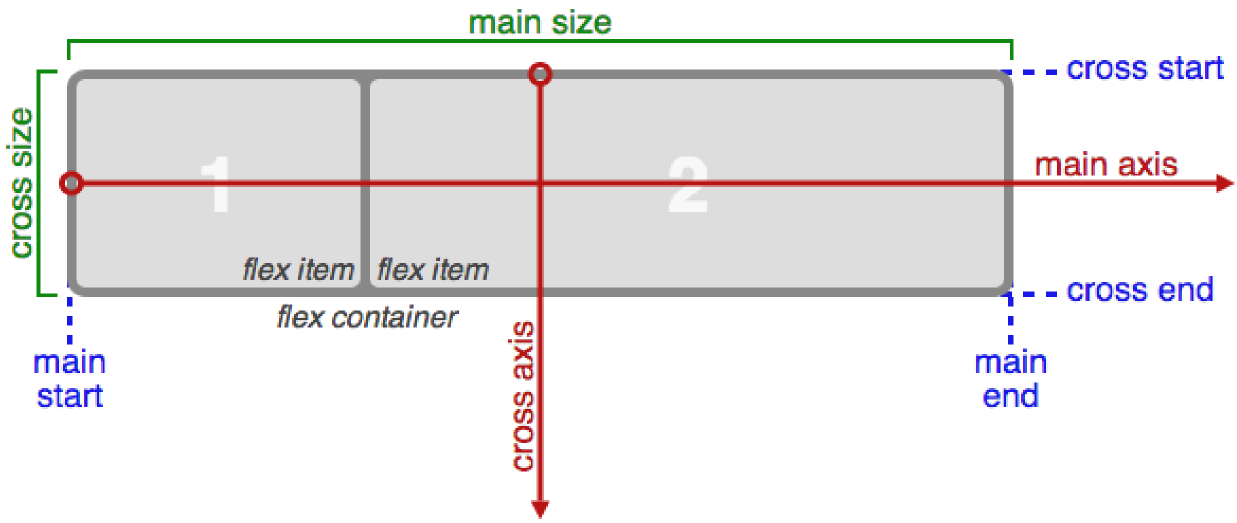


Fig. 1.2: Flexbox

- `main-start | main-end` - The flex items are placed within the container starting from main-start and going to main-end.
- `main size` - A flex item's width or height, whichever is in the main dimension, is the item's main size. The flex item's main size property is either the 'width' or 'height' property, whichever is in the main dimension. cross axis - The axis perpendicular to the main axis is called the cross axis. Its direction depends on the main axis direction.
- `cross-start | cross-end` - Flex lines are filled with items and placed into the container starting on the cross-start side of the flex container and going toward the cross-end side.
- `cross size` - The width or height of a flex item, whichever is in the cross dimension, is the item's cross size. The cross size property is whichever of 'width' or 'height' that is in the cross dimension.

Properties of the parent

Fig. 1.3: Container

- `display` (must be equal to 'flex' or 'inline-flex')

This defines a flex container (inline or block). - `flex-flow` **(shorthand for two properties)**

This is a shorthand `flex-direction` and `flex-wrap` properties, which together define the flex container's main and cross axes. Default is `row nowrap`.

```
- `flex-direction` (row | row-reverse | column | column-reverse)

  This establishes the main-axis, thus defining the direction flex items are placed␣
↪in the flex container. Flexbox is (aside from optional wrapping) a single-direction␣
↪layout concept. Think of flex items as primarily laying out either in horizontal␣
↪rows or vertical columns.
  ![Direction](./images/flex-direction1.svg)

- `flex-wrap` (nowrap | wrap | wrap-reverse)
```

```
By default, flex items will all try to fit onto one line. You can change that and␣
↪allow the items to wrap as needed with this property. Direction also plays a role␣
↪here, determining the direction new lines are stacked in.
![Wrap](./images/flex-wrap.svg)
```

- `justify-content` (flex-start | flex-end | center | space-between | space-around)

This defines the alignment along the main axis. It helps distribute extra free space left over when either all the flex items on a line are inflexible, or are flexible but have reached their maximum size. It also exerts some control over the alignment of items when they overflow the line.

- `align-items` (flex-start | flex-end | center | baseline | stretch)

This defines the default behaviour for how flex items are laid out along the cross axis on the current line. Think of it as the justify-content version for the cross-axis (perpendicular to the main-axis).

- `align-content` (flex-start | flex-end | center | baseline | stretch)

This aligns a flex container's lines within when there is extra space in the cross-axis, similar to how justify-content aligns individual items within the main-axis.

Note: this property has no effect when there is only one line of flex items.

Properties of the items

Fig. 1.4: Item

The flexbox-related css properties of the items have no impact if the parent element is not a flexbox container (i.e. has a `display` attribute equal to `flex` or `inline-flex`).

- `order`

By default, flex items are laid out in the source order. However, the order property controls the order in which they appear in the flex container.

- `flex` **(shorthand for three properties)** This is the shorthand for flex-grow, flex-shrink and flex-basis combined. The second and third parameters (flex-shrink and flex-basis) are optional. Default is `0 1 auto`.

  – `flex-grow`

  This defines the ability for a flex item to grow if necessary. It accepts a unitless value that serves as a proportion. It dictates what amount of the available space inside the flex container the item should take up.

  If all items have flex-grow set to 1, the remaining space in the container will be distributed equally to all children. If one of the children a value of 2, the remaining space would take up twice as much space as the others (or it will try to, at least).

  – `flex-shrink`

  This defines the ability for a flex item to shrink if necessary.

  – `flex-basis`

  This defines the default size of an element before the remaining space is distributed. It can be a length (e.g. `20%`, `5rem`, etc.) or a keyword. The `auto` keyword means *"look at my width or height property"*.

- `align-self`

This allows the default alignment (or the one specified by align-items) to be overridden for individual flex items.

Fig. 1.5: Align

The VBox and HBox helpers

The `VBox` and `HBox` helper provide simple defaults to arrange child widgets in Vertical and Horizontal boxes.

```
def VBox(*pargs, **kwargs):
    """Displays multiple widgets vertically using the flexible box model."""
    box = Box(*pargs, **kwargs)
    box.layout.display = 'flex'
    box.layout.flex_flow = 'column'
    box.layout.align_items = 'stretch'
    return box

def HBox(*pargs, **kwargs):
    """Displays multiple widgets horizontally using the flexible box model."""
    box = Box(*pargs, **kwargs)
    box.layout.display = 'flex'
    box.layout.align_items = 'stretch'
    return box
```

Examples

Four buttons in a ‘‘VBox‘‘. Items stretch to the maximum width, in a vertical box taking ‘‘50%‘‘ of the available space.

```
In [9]: items_layout = Layout(flex='1 1 auto',
                              width='auto')     # override the default width of the button

        box_layout = Layout(display='flex',
                            flex_flow='column',
                            align_items='stretch',
                            border='solid',
                            width='50%')

        items = [Button(description=w, layout=items_layout, button_style='danger') for w in
        box = Box(children=items, layout=box_layout)
        box
```

```
Widget Javascript not detected.  It may not be installed properly. Did you enable the widge
```

Three buttons in an HBox. Items flex proportionaly to their weight.

```
In [10]: items_layout = Layout(flex='1 1 auto', width='auto')     # override the default w

         items = [
             Button(description='weight=1'),
             Button(description='weight=2', layout=Layout(flex='2 1 auto', width='auto')),
             Button(description='weight=1'),
          ]

         box_layout = Layout(display='flex',
                             flex_flow='row',
                             align_items='stretch',
```

```
                    border='solid',
                    width='50%')
box = Box(children=items, layout=box_layout)
box
```

```
Widget Javascript not detected.  It may not be installed properly. Did you enable the widge
```

A more advanced example: a reactive form.

The form is a `VBox` of width '50%'. Each row in the VBox is an HBox, that justifies the content with space between..

```
In [11]: label_layout = Layout()

         form_item_layout = Layout(
             display='flex',
             flex_flow='row',
             justify_content='space-between'
         )

         form_items = [
             Box([Label(value='Age of the captain'), IntSlider(min=40, max=60)], layout=fo
             Box([Label(value='Egg style'),
                  Dropdown(options=['Scrambled', 'Sunny side up', 'Over easy'])], layout=f
             Box([Label(value='Ship size'),
                  FloatText()], layout=form_item_layout),
             Box([Label(value='Information'),
                  Textarea()], layout=form_item_layout)
         ]

         form = Box(form_items, layout=Layout(
             display='flex',
             flex_flow='column',
             border='solid 2px',
             align_items='stretch',
             width='50%'
         ))
         form
```

```
Widget Javascript not detected.  It may not be installed properly. Did you enable the widge
```

A more advanced example: a carousel.

```
In [12]: item_layout = Layout(height='100px')
         items = [Button(layout=item_layout, button_style='warning') for i in range(40)]
         box_layout = Layout(overflow_x='scroll',
                             border='3px solid black',
                             width='500px',
                             height='',
                             flex_direction='row',
                             display='flex')
         HBox(children=items, layout=box_layout)
```

```
Widget Javascript not detected.  It may not be installed properly. Did you enable the widge
```

Index - Back

```
In [1]: from __future__ import print_function
```

1.7 Building a Custom Widget - Hello World

The widget framework is built on top of the Comm framework (short for communication). The Comm framework is a framework that allows the kernel to send/receive JSON messages to/from the front end (as seen below).

Fig. 1.6: Widget layer

To create a custom widget, you need to define the widget both in the browser and in the python kernel.

1.7.1 Building a Custom Widget

To get started, you'll create a simple hello world widget. Later you'll build on this foundation to make more complex widgets.

1.7.2 Python Kernel

DOMWidget and Widget

To define a widget, you must inherit from the Widget or DOMWidget base class. If you intend for your widget to be displayed in the Jupyter notebook, you'll want to inherit from the DOMWidget. The DOMWidget class itself inherits from the Widget class. The Widget class is useful for cases in which the Widget is not meant to be displayed directly in the notebook, but instead as a child of another rendering environment. For example, if you wanted to create a three.js widget (a popular WebGL library), you would implement the rendering window as a DOMWidget and any 3D objects or lights meant to be rendered in that window as Widgets.

*view*name

Inheriting from the DOMWidget does not tell the widget framework what front end widget to associate with your back end widget.

Instead, you must tell it yourself by defining specially named trait attributes, `_view_name` and `_view_module` (as seen below) and optionally `_model_name` and `_model_module`.

```
In [2]: import ipywidgets as widgets
        from traitlets import Unicode, validate


        class HelloWidget(widgets.DOMWidget):
            _view_name = Unicode('HelloView').tag(sync=True)
            _view_module = Unicode('hello').tag(sync=True)
```

sync=True traitlets

Traitlets is an IPython library for defining type-safe properties on configurable objects. For this tutorial you do not need to worry about the *configurable* piece of the traitlets machinery. The `sync=True` keyword argument tells the widget framework to handle synchronizing that value to the browser. Without `sync=True`, the browser would have no knowledge of `_view_name` or `_view_module`.

Other traitlet types

Unicode, used for *view*name, is not the only Traitlet type, there are many more some of which are listed below:

- Any
- Bool
- Bytes
- CBool
- CBytes
- CComplex
- CFloat
- CInt
- CLong
- CRegExp
- CUnicode
- CaselessStrEnum
- Complex
- Dict
- DottedObjectName
- Enum
- Float
- FunctionType
- Instance
- InstanceType
- Int
- List
- Long

- Set
- TCPAddress
- Tuple
- Type
- Unicode
- Union

Not all of these traitlets can be synchronized across the network, only the JSON-able traits and Widget instances will be synchronized.

1.7.3 Front end (JavaScript)

Models and views

The IPython widget framework front end relies heavily on Backbone.js. Backbone.js is an MVC (model view controller) framework. Widgets defined in the back end are automatically synchronized with generic Backbone.js models in the front end. The traitlets are added to the front end instance automatically on first state push. The `_view_name` trait that you defined earlier is used by the widget framework to create the corresponding Backbone.js view and link that view to the model.

Import jupyter-js-widgets

You first need to import the `jupyter-js-widgets` module. To import modules, use the `define` method of require.js (as seen below).

```
In [3]: %%javascript
        define('hello', ["jupyter-js-widgets"], function(widgets) {

        });
```

```
<IPython.core.display.Javascript object>
```

Define the view

Next define your widget view class. Inherit from the `DOMWidgetView` by using the `.extend` method.

```
In [4]: %%javascript
        require.undef('hello');

        define('hello', ["jupyter-js-widgets"], function(widgets) {

            // Define the HelloView
            var HelloView = widgets.DOMWidgetView.extend({

            });

            return {
                HelloView: HelloView
            }
        });
```

```
<IPython.core.display.Javascript object>
```

Render method

Lastly, override the base `render` method of the view to define custom rendering logic. A handle to the widget's default DOM element can be acquired via `this.el`. The `el` property is the DOM element associated with the view.

```
In [5]: %%javascript
        require.undef('hello');

        define('hello', ["jupyter-js-widgets"], function(widgets) {

            var HelloView = widgets.DOMWidgetView.extend({

                // Render the view.
                render: function() {
                    this.el.textContent = 'Hello World!';
                },
            });

            return {
                HelloView: HelloView
            };
        });
```

```
<IPython.core.display.Javascript object>
```

1.7.4 Test

You should be able to display your widget just like any other widget now.

```
In [6]: HelloWidget()
```

examples/../_build/doctrees/nbsphinx/examples_WidgetCustom_30_0.png

Data type cannot be displayed: application/vnd.jupyter.widget

1.7.5 Making the widget stateful

There is not much that you can do with the above example that you can't do with the IPython display framework. To change this, you will make the widget stateful. Instead of displaying a static "hello world" message, it will display a string set by the back end. First you need to add a traitlet in the back end. Use the name of `value` to stay consistent with the rest of the widget framework and to allow your widget to be used with interact.

```
In [7]: class HelloWidget(widgets.DOMWidget):
            _view_name = Unicode('HelloView').tag(sync=True)
            _view_module = Unicode('hello').tag(sync=True)
            value = Unicode('Hello World!').tag(sync=True)
```

Accessing the model from the view

To access the model associate with a view instance, use the `model` property of the view. `get` and `set` methods are used to interact with the Backbone model. `get` is trivial, however you have to be careful when using `set`. After calling the model `set` you need call the view's `touch` method. This associates the `set` operation with a particular view so output will be routed to the correct cell. The model also has an `on` method which allows you to listen to events triggered by the model (like value changes).

Rendering model contents

By replacing the string literal with a call to `model.get`, the view will now display the value of the back end upon display. However, it will not update itself to a new value when the value changes.

```
In [8]: %%javascript
        require.undef('hello');

        define('hello', ["jupyter-js-widgets"], function(widgets) {

            var HelloView = widgets.DOMWidgetView.extend({

                render: function() {
                    this.el.textContent = this.model.get('value');
                },
            });

            return {
                HelloView : HelloView
            };
        });
```

```
<IPython.core.display.Javascript object>
```

Dynamic updates

To get the view to update itself dynamically, register a function to update the view's value when the model's `value` property changes. This can be done using the `model.on` method. The `on` method takes three parameters, an event name, callback handle, and callback context. The Backbone event named `change` will fire whenever the model changes. By appending `:value` to it, you tell Backbone to only listen to the change event of the `value` property (as seen below).

```
In [9]: %%javascript
        require.undef('hello');

        define('hello', ["jupyter-js-widgets"], function(widgets) {

            var HelloView = widgets.DOMWidgetView.extend({

                render: function() {
                    this.value_changed();
                    this.model.on('change:value', this.value_changed, this);
                },

                value_changed: function() {
                    this.el.textContent = this.model.get('value');
```

```
                },
            });

            return {
                HelloView : HelloView
            };
        });
```

```
<IPython.core.display.Javascript object>
```

1.7.6 Test

```
In [10]: w = HelloWidget()
         w
```

examples/../_build/doctrees/nbsphinx/examples_WidgetCustom_43_0.png

Data type cannot be displayed: application/vnd.jupyter.widget

```
In [11]: w.value = 'test'
```

1.7.7 Conclusion

The example above dumps the value directly into the DOM. There is no way for you to interact with this dumped data in the front end. To create an example that accepts input, you will have to do something more than blindly dumping the contents of value into the DOM.

In the next section of the tutorial, you will build a date picker to display and accept input in the front end.

1.8 Building a Custom Widget - Date Picker

In the last section we created a simple widget that displayed *Hello World!* To make an actual date widget, we need to add a property that will be synced between the Python model and the JavaScript model. The new attribute must be a traitlet, so the widget machinery can handle it. The traitlet must be constructed with a `sync=True` keyword argument, to tell the widget machinery knows to synchronize it with the front-end. Adding this to the code from the last section:

```
In [12]: class DateWidget(widgets.DOMWidget):
             _view_name = Unicode('DatePickerView').tag(sync=True)
             _view_module = Unicode('datepicker').tag(sync=True)
             value = Unicode().tag(sync=True)
```

1.8.1 JavaScript

In the JavaScript, there is no need to define counterparts to the traitlets. When the JavaScript model is created for the first time, it copies all of the traitlet `sync=True` attributes from the Python model. We need to replace *Hello World!*

with an actual HTML date picker widget.

```
In [13]: %%javascript
         requirejs.undef('datepicker');

         define('datepicker', ["jupyter-js-widgets"], function(widgets) {

             var DatePickerView = widgets.DOMWidgetView.extend({
                 render: function() {
                     this.date = document.createElement('input');
                     this.date.setAttribute('type', 'date');
                     this.el.appendChild(this.date);
                 },
             });

             return {
                 DatePickerView: DatePickerView
             };
         });
```

```
<IPython.core.display.Javascript object>
```

In order to get the HTML date picker to update itself with the value set in the back-end, we need to implement an `update()` method.

```
In [14]: %%javascript
         requirejs.undef('datepicker');

         define('datepicker', ["jupyter-js-widgets"], function(widgets) {

             var DatePickerView = widgets.DOMWidgetView.extend({
                 render: function() {
                     this.date = documentElement('input');
                     this.date.setAttribute('type', 'date');
                     this.el.appendChild(this.date);
                     this.update();
                 },

                 update: function() {
                     // Set the value of the date control and then call base.
                     // ISO format "YYYY-MM-DDTHH:mm:ss.sssZ" is required
                     this.date.value = this.model.get('value');
                     return DatePickerView.__super__.update.apply(this);
                 },
             });

             return {
                 DatePickerView: DatePickerView
             };
         });
```

```
<IPython.core.display.Javascript object>
```

To get the changed value from the frontend to publish itself to the backend, we need to listen to the change event triggered by the HTM date control and set the value in the model. After the date change event fires and the new value is set in the model, it is very important that we call `this.touch()` to let the widget machinery know which view changed the model. This is important because the widget machinery needs to know which cell to route the message callbacks to.

Final JavaScript code below:

In [15]:
```
%%javascript
requirejs.undef('datepicker');

define('datepicker', ["jupyter-js-widgets"], function(widgets) {

    var DatePickerView = widgets.DOMWidgetView.extend({
        render: function() {
            this.date = document.createElement('input');
            this.date.setAttribute('type', 'date');
            this.el.appendChild(this.date);
            this.update();
        },

        update: function() {
            // Set the value of the date control and then call base.
            // ISO format "YYYY-MM-DDTHH:mm:ss.sssZ" is required
            this.date.value = this.model.get('value');
            return DatePickerView.__super__.update.apply(this);
        },

        // Tell Backbone to listen to the change event of input controls (which th
        events: {
            "change": "handle_date_change"
        },

        // Callback for when the date is changed.
        handle_date_change: function(event) {
            this.model.set('value', this.date.value);
            this.touch();
        },
    });

    return {
        DatePickerView: DatePickerView
    };
});
```

```
<IPython.core.display.Javascript object>
```

1.8.2 Test

To test, create the widget the same way that the other widgets are created.

In [16]:
```
my_widget = DateWidget()
my_widget
```

examples/../_build/doctrees/nbsphinx/examples_WidgetCustom_55_0.png

Data type cannot be displayed: application/vnd.jupyter.widget

Display the widget again to make sure that both views remain in sync.

```
In [17]: my_widget
```

examples/../_build/doctrees/nbsphinx/examples_WidgetCustom_57_0.png

Data type cannot be displayed: application/vnd.jupyter.widget

Read the date from Python

```
In [18]: my_widget.value
```

```
Out[18]: ''
```

Set the date from Python

```
In [19]: my_widget.value = "1998-12-01" # December 1st, 1998
```

1.9 Building a Custom Widget - A More Advanced Date Picker

The 3rd party `dateutil` library is required to continue. https://pypi.python.org/pypi/python-dateutil

1.9.1 Import the dateutil library to parse date strings.

```
In [20]: from dateutil import parser
```

In the last section we created a fully working date picker widget. Now we will add custom validation and support for labels. So far, only the ISO date format "YYYY-MM-DD" is supported. Now, we will add support for all of the date formats recognized by the Python dateutil library.

1.9.2 Python

The standard property name used for widget labels is `description`. In the code block below, `description` has been added to the Python widget.

```
In [21]: class DateWidget(widgets.DOMWidget):
             _view_name = Unicode('DatePickerView').tag(sync=True)
```

```
            _view_module = Unicode('datepicker').tag(sync=True)
            value = Unicode().tag(sync=True)
            description = Unicode().tag(sync=True)
```

Adding a custom validator for `value`

```
In [22]: class DateWidget(widgets.DOMWidget):
             _view_name = Unicode('DatePickerView').tag(sync=True)
             _view_module = Unicode('datepicker').tag(sync=True)
             value = Unicode().tag(sync=True)
             description = Unicode().tag(sync=True)

             @validate('value')
             def _validate_value(self, proposal):
                 value = proposal['value']
                 try:
                     parsed_date = parser.parse(value)
                     parsed_date_string = parsed_date.strftime("%Y-%m-%d")
                 except:
                     parsed_date_string = ''
                 return parsed_date_string
```

1.9.3 JavaScript

Using the Javascript code from the last section, we add a label to the date time object. The label is a div with the `widget-hlabel` class applied to it. `widget-hlabel` is a class provided by the widget framework that applies special styling to a div to make it look like the rest of the horizontal labels used with the built-in widgets. Similar to the `widget-hlabel` class is the `widget-hbox-single` class. The `widget-hbox-single` class applies special styling to widget containers that store a single line horizontal widget.

We hide the label if the description value is blank.

```
In [23]: %%javascript
         requirejs.undef('datepicker');

         define('datepicker', ["jupyter-js-widgets"], function(widgets) {

             // Define the DatePickerView
             var DatePickerView = widgets.DOMWidgetView.extend({
                 render: function() {
                     // apply the standard widget classes so the css styles are consistent
                     this.el.className = 'widget-hbox jupyter-widgets widget-width';

                     this.label = document.createElement('div')
                     this.label.className = 'widget-label';
                     this.label.style.display = 'none';

                     this.date = document.createElement('input');
                     this.date.className = "form-control";
                     this.date.setAttribute('type', 'date');

                     this.el.appendChild(this.label);
                     this.el.appendChild(this.date);

                     this.update();
```

```
                },

                update: function() {
                    // Set the value of the date control and then call base.
                    // ISO format "YYYY-MM-DDTHH:mm:ss.sssZ" is required
                    this.date.value = this.model.get('value');

                    // Hide or show the label depending on the existance of a description
                    var description = this.model.get('description');
                    if (description == undefined || description == '') {
                        this.label.style.display = 'none';
                    } else {
                        this.label.innerText = description;
                        this.label.style.display = '';
                    }
                    return DatePickerView.__super__.update.apply(this);
                },

                // Tell Backbone to listen to the change event of input controls (which th
                events: {
                    "change": "handle_date_change"
                },

                // Callback for when the date is changed.
                handle_date_change: function(event) {
                    this.model.set('value', this.date.value);
                    this.touch();
                },
            });

            return {
                DatePickerView: DatePickerView
            };
        });
```

```
<IPython.core.display.Javascript object>
```

1.9.4 Test

To test the drawing of the label we create the widget like normal but supply the additional description property a value.

```
In [24]: # Add some additional widgets for aesthetic purpose
         my_widget = DateWidget()
         my_widget.description="DOB:"

         widgets.VBox([widgets.Text(description="First:"),
                       widgets.Text(description="Last:"),
                       my_widget])
```

examples/../_build/doctrees/nbsphinx/examples_WidgetCustom_76_0.png

Data type cannot be displayed: application/vnd.jupyter.widget

1.9.5 More advanced uses: Packaging and distributing Jupyter widgets

A template project is available in the form of a cookie cutter: https://github.com/jupyter/widget-cookiecutter

This project is meant to help custom widget authors get started with the packaging and the distribution of Jupyter interactive widgets.

It produces a project for a Jupyter interactive widget library following the current best practices for using interactive widgets. An implementation for a placeholder "Hello World" widget is provided.

Index - Back

```
In [1]: from ipywidgets import *
```

1. `VBox(HBox)`

```
In [2]: VBox([HBox([VBox([Dropdown(description='Choice', options=['foo', 'bar']),
                          ColorPicker(description='Color'),
                          HBox([Button(), Button()])]),
                    Textarea(value="Lorem ipsum dolor sit amet, consectetur adipiscing elit
        "sed do eiusmod tempor incididunt ut labore et dolore magna aliqua. "
        "Ut enim ad minim veniam, quis nostrud exercitation ullamco laboris "
        "nisi ut aliquip ex ea commodo consequat. Duis aute irure dolor in "
        "reprehenderit in voluptate velit esse cillum dolore eu fugiat nulla "
        "pariatur. Excepteur sint occaecat cupidatat non proident, sunt in culpa "
        "qui officia deserunt mollit anim id est laborum.")]),
              HBox([Text(), Checkbox(description='Check box')]),
              IntSlider(),
              Controller()], background_color='#EEE')
```

examples/../_build/doctrees/nbsphinx/examples_WidgetAlignment_2_0.png

2. `HBox(VBox)`

```
In [3]: HBox([VBox([Button(description='Press'), Dropdown(options=['a', 'b']), Button(desc
              VBox([Button(), Checkbox(), IntText()]),
              VBox([Button(), IntSlider(), Button()])], background_color='#EEE')
```

examples/../_build/doctrees/nbsphinx/examples_WidgetAlignment_4_0.png

3. `VBox(HBox)` width sliders, range sliders and progress bars

```
In [4]: VBox([HBox([Button(), FloatRangeSlider(), Text(), Button()]),
              HBox([Button(), FloatText(), Button(description='Button'),
                    FloatProgress(value=40), Checkbox(description='Check')]),
              HBox([ToggleButton(), IntSlider(description='Foobar'),
                    Dropdown(options=['foo', 'bar']), Valid()]),
             ])
```

examples/../_build/doctrees/nbsphinx/examples_WidgetAlignment_6_0.png

4. Dropdown resize

```
In [5]: dd = Dropdown(description='Foobar')
        dd
```

examples/../_build/doctrees/nbsphinx/examples_WidgetAlignment_8_0.png

```
In [6]: dd.height = '50px'
        dd.width = '148px'
```

```
In [7]: cp = ColorPicker(description='foobar')
```

5. Colorpicker alignment, concise and long version

```
In [8]: VBox([HBox([Dropdown(width='148px'), Button(description='Button')]), cp, HBox([Butt
```

examples/../_build/doctrees/nbsphinx/examples_WidgetAlignment_12_0.png

```
In [9]: cp.concise = True
```

```
In [10]: cp.concise = False
```

```
In [11]: cp2 = ColorPicker()
```

```
In [12]: VBox([HBox([Button(), Button()]), cp2])
```

examples/../_build/doctrees/nbsphinx/examples_WidgetAlignment_16_0.png

```
In [13]: cp2.concise = True
```

```
In [14]: cp2.concise = False
```

6. Vertical slider and progress bar alignment and resize

```
In [15]: HBox([IntSlider(description='Slider', orientation='vertical', height='200px'),
               FloatProgress(description='Progress', value=50, orientation='vertical', heig
```

examples/../_build/doctrees/nbsphinx/examples_WidgetAlignment_20_0.png

```
In [16]: HBox([IntSlider(description='Slider', orientation='vertical'),
               FloatProgress(description='Progress', value=50, orientation='vertical')])
```

examples/../_build/doctrees/nbsphinx/examples_WidgetAlignment_21_0.png

7. Tabs

```
In [ ]: t = Tab(children=[FloatText(), IntSlider()], _titles={0: 'Text', 1: 'Slider'})
        t
```

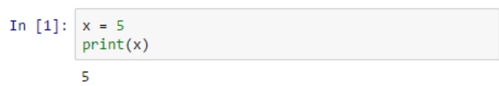
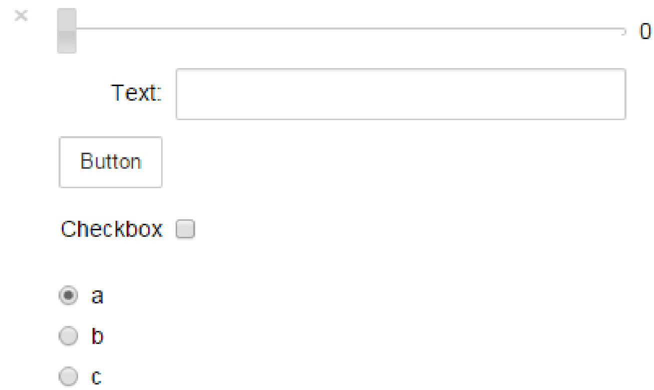
```
In [ ]: t.selected_index = 1
```

```
In [ ]:
```

1.10 Low Level Widget Tutorial

1.10.1 How do they fit into the picture?

One of the goals of the Jupyter Notebook is to minimize the “distance” the user is from their data. This means allowing the user to quickly view and manipulate the data.

<table>
<tr>
<td>
</td>
<td>
</td>
</tr>
<tr>
<td>Before the widgets, this was just the segmentation of code and results from executing those segments.</td>
<td>Widgets further decrease the distance between the user and their data by allowing UI interactions to directly manipulate data in the kernel.</td>
</tr>
</table>

1.10.2 How?

Jupyter interactive widgets are interactive elements, think sliders, textboxes, buttons, that have representations both in the kernel (place where code is executed) and the front-end (the Notebook web interface). To do this, a clean, well abstracted communication layer must exist.

1.10.3 Comms

This is where Jupyter notebook "comms" come into play. The comm API is a symmetric, asynchronous, fire and forget style messaging API. It allows the programmer to send JSON-able blobs between the front-end and the back-end. The comm API hides the complexity of the webserver, ZMQ, and websockets.

1.10.4 Synchronized state

Using comms, the widget base layer is designed to keep state in sync. In the kernel, a Widget instance exists. This Widget instance has a corresponding WidgetModel instance in the front-end. The Widget and WidgetModel store the same state. The widget framework ensures both models are kept in sync with eachother. If the WidgetModel is changed in the front-end, the Widget receives the same change in the kernel. Vise versa, if the Widget in the kernel is changed, the WidgetModel in the front-end receives the same change. There is no single source of truth, both models have the same precedence. Although a notebook has the notion of cells, neither Widget or WidgetModel are bound to any single cell.

1.10.5 Models and Views

In order for the user to interact with widgets on a cell by cell basis, the WidgetModels are represented by WidgetViews. Any single WidgetView is bound to a single cell. Multiple WidgetViews can be linked to a single WidgetModel. This is how you can redisplay the same Widget multiple times and it still works. To accomplish this, the widget framework uses Backbone.js. In a traditional MVC framework, the WidgetModel is the (M)odel, and the WidgetView is both the (V)iew and (C)ontroller. Meaning that, the views both display the state of the model and manipulate it. Think about a slider control, it both displays the value and allows the user to change the value by dragging the slide handle.

```
In [1]: from ipywidgets import *
        from IPython.display import display
        w = IntSlider()
        display(w, w)
```

examples/../_build/doctrees/nbsphinx/examples_WidgetLowLevel_14_0.png

Data type cannot be displayed: application/vnd.jupyter.widget

Data type cannot be displayed: application/vnd.jupyter.widget

```
In [2]: display(w)
```

examples/../_build/doctrees/nbsphinx/examples_WidgetLowLevel_15_0.png

Data type cannot be displayed: application/vnd.jupyter.widget

1.10.6 Code execution

The user code required to display a simple FloatSlider widget is:

```
from ipywidgets import FloatSlider
from IPython.display import display
slider = FloatSlider()
display(slider)
```

In order to understand how a widget is displayed, one must understand how code is executed in the Notebook. Execution begins in the code cell. A user event triggers the code cell to send an evaluate code message to the kernel, containing all of the code in the code cell. This message is given a GUID, which the front-end associates to the code cell, and remembers it (**important**).

Once that message is received by the kernel, the kernel immediately sends the front-end an “I’m busy” status message. The kernel then proceeds to execute the code.

1.10.7 Model construction

When a Widget is constructed in the kernel, the first thing that happens is that a comm is constructed and associated with the widget. When the comm is constructed, it is given a GUID (globally unique identifier). A comm-open message is sent to the front-end, with metadata stating that the comm is a widget comm and what the corresponding WidgetModel class is.

The WidgetModel class is specified my module and name. Require.js is then used to asynchronously load the WidgetModel class. The message triggers a comm to be created in the front-end with same GUID as the back-end. Then, the new comm gets passed into the WidgetManager in the front-end, which creates an instance of the WidgetModel class, linked to the comm. Both the Widget and WidgetModel repurpose the comm GUID as their own.

Asynchronously, the kernel sends an initial state push, containing all of the initial state of the Widget, to the front-end, immediately after the comm-open message. This state message may or may not be received by the time the WidgetModel is constructed. Regardless, the message is cached and gets processed once the WidgetModel has been constructed. The initial state push is what causes the WidgetModel in the front-end to become in sync with the Widget in the kernel.

1.10.8 Displaying a view

After the Widget has been constructed, it can be displayed. Calling `display(widgetinstance)` causes a specially named repr method in the widget to run. This method sends a message to the front-end that tells the front-end to construct and display a widget view. The message is in response to the original code execution message, and the original message's GUID is stored in the new message's header. When the front-end receives the message, it uses the original messsage's GUID to determine what cell the new view should belong to. Then, the view is created, using the WidgetView class specified in the WidgetModel's state. The same require.js method is used to load the view class. Once the class is loaded, an instance of it is constructed, displayed in the right cell, and registers listeners for changes of the model.

1.10.9 Widget skeleton

```
In [ ]: %%javascript
        this.model.get('count');


        this.model.set('count', 999);
        this.touch();

        /////////////////////////////////

        this.colorpicker = document.createElement('input');
        this.colorpicker.setAttribute('type', 'color');
        this.el.appendChild(this.colorpicker);
```

Since widgets exist in both the front-end and kernel, they consist of both Python (if the kernel is IPython) and Javascript code. A boilerplate widget can be seen below:

Python:

```python
from ipywidgets import DOMWidget
from traitlets import Unicode, Int

class MyWidget(DOMWidget):
    _view_module = Unicode('nbextensions/mywidget/mywidget').tag(sync=True)
    _view_name = Unicode('MyWidgetView').tag(sync=True)
    count = Int().tag(sync=True)
```

JavaScript:

```javascript
define(['jupyter-js-widgets'], function(widgets) {
    var MyWidgetView = widgets.DOMWidgetView.extend({
        render: function() {
            MyWidgetView.__super__.render.apply(this, arguments);
            this._count_changed();
            this.listenTo(this.model, 'change:count', this._count_changed, this);
        },

        _count_changed: function() {
            var old_value = this.model.previous('count');
            var new_value = this.model.get('count');
            this.el.textContent = String(old_value) + ' -> ' + String(new_value);
        }
    });

    return {
        MyWidgetView: MyWidgetView
    }
});
```

Describing the Python:

The base widget classes are `DOMWidget` and `Widget`.

`_view_module` and `_view_name` are how the front-end knows what view class to construct for the model.

`sync=True` is what makes the traitlets behave like state.

A similarly named `_model_module` and `_model_name` can be used to specify the corresponding WidgetModel.

`count` is an example of a custom piece of state.

Describing the JavaScript:

The `define` call asynchronously loads the specified dependencies, and then passes them in as arguments into the callback. Here, the only dependency is the base widget module are loaded.

Custom views inherit from either `DOMWidgetView` or `WidgetView`.

Likewise, custom models inherit from `WidgetModel`.

The `render` method is what is called to render the view's contents. If the view is a `DOMWidgetView`, the `.el` attribute contains the DOM element that will be displayed on the page.

`.listenTo` allows the view to listen to properties of the model for changes.

`_count_changed` is an example of a method that could be used to handle model changes.

`this.model` is how the corresponding model can be accessed.

`this.model.previous` will get the previous value of the trait.

`this.model.get` will get the current value of the trait.

`this.model.set` followed by `this.save_changes();` changes the model. The view method `save_changes` is needed to associate the changes with the current view, thus associating any response messages with the view's cell.

The dictionary returned is the public members of the module.

1.10.10 Installation

Because the API of any given widget **must exist in the kernel**, the kernel is the natural place for widgets to be installed. However, **kernels, as of now, don't host static assets**. Instead, static assets are hosted by the webserver, which is the entity that sits between the kernel and the front-end. This is a problem, because it means widgets have components that need to be **installed both in the webserver and the kernel**. The kernel components are easy to install, because you can rely on the language's built in tools. The static assets for the webserver complicate things, because an extra step is required to let the webserver know where the assets are.

1.10.11 Static assets

Static assets can be made available to the Jupyter notebook webserver a few ways: 1. Custom.js: By placing your static assets inside the custom directory, alongside custom.js, you can load them within custom.js. The problem with deployment utilizing this method is that the users will have to manually edit their custom.js file. 2. Server extension: You can write a server extension which adds a new handler to the server that hosts your static content. However, the server extension needs to be registered using a config file, which means the user needs to edit it by hand. Also, code needs to be written in custom.js to load the static content. 3. Notebook extension: By placing your static assets inside the nbextensions directory, they are made available by the `nbextensions/` handler. Nbextensions also have a mechanism for running your code on page load. This can be set using the install-nbextension command.

Notebook extensions are the best solution, for now, because they can be used without manual user intervention.

1.10.12 Distribution

Integrating the static assets with the kernel code is tricky. Ideally, the user would be able to execute a single command to install the package, kernel and static assets included.

A template project is available in the form of a cookie cutter: https://github.com/jupyter/widget-cookiecutter

This project is meant to help custom widget authors get started with the packaging and the distribution of Jupyter interactive widgets.

It produces a project for a Jupyter interactive widget library following the current best practices for using interactive widgets. An implementation for a placeholder "Hello World" widget is provided.

1.11 Contributing

We appreciate contributions from the community.

We follow the IPython Contributing Guide and Jupyter Contributing Guides.

1.12 ipywidgets changelog

A summary of changes in ipywidgets. For more detailed information, see GitHub.

1.12.1 4.1.x

4.1.1

4.1.0

1.12.2 4.0.x

4.0.3

Bump version with miscellaneous bug fixes.

4.0.2

Add README.rst documentation.

4.0.1

Remove ipynb checkpoints.

4.0.0

First release of **ipywidgets** as a standalone package.

1.13 Developer Install

1.13.1 Prerequisites

To install ipywidgets from git, you will need:

- npm **version 3.x or later**
  - Check your version by running `npm -v` from a terminal.
  - *Note: If you install using sudo, you need to make sure that npm is also available in the PATH used with sudo.*
- the latest Jupyter notebook development release
  - Everything in the ipywidgets repository is developed using Jupyter notebook's master branch.
  - If you want to have a copy of ipywidgets that works against a stable version of the notebook, checkout the appropriate tag.
  - See the Compatibility table.

1.13.2 Steps

1. Clone the repo:

```
git clone https://github.com/ipython/ipywidgets
```

2. Navigate into the cloned repo and install:

```
cd ipywidgets
bash dev-install.sh --sys-prefix
```

1.13.3 Rebuild after making changes

After you've made changes to `jupyter-js-widgets` if you want to test those changes, run the following commands, empty your browser's cache, and refresh the page:

```
    cd widgetsnbextension
    npm run update
    cd ..
```

1.13.4 Tips and troubleshooting

- If you have any problems with the above install procedure, make sure that permissions on npm and pip related install directories are correct.
- Sometimes, it helps to clear cached files too by running `git clean -dfx` from the root of the cloned repository.
- When you make changes to the Javascript and you're not seeing the changes, it could be your browser is caching aggressively. Try clearing the browser's cache. Try also using "incognito" or "private" browsing tabs to avoid caching.
- If troubleshooting an upgrade and its build, you may need to do the following process:
  - Deep clean of the cloned repository:

    ```
    git clean -dfx .
    ```

  - Remove anything with `widgetsnbextension` in the name of files within the `conda` directory
  - Try reinstalling ipywidgets

1.13.5 Releasing new versions

See dev_release.md for a details on how to release new versions of ipywidgets to PyPI and jupyter-js-widgets on npm.

1.13.6 Testing

See dev_testing.md for a details on how to run Python and Javascript tests.

1.13.7 Building documentation

See dev_docs.md for a details on how to build the docs.

1.14 Testing

To run the Python tests:

```
nosetests --with-coverage --cover-package=ipywidgets ipywidgets
```

To run the Javascript tests:

```
cd jupyter-js-widgets; npm run test
```

This will run the test suite using `karma` with 'debug' level logging.

1.15 Building the Documentation

To build the documentation you'll need Sphinx, pandoc and a few other packages.

To install (and activate) a conda environment named `notebook_docs` containing all the necessary packages (except pandoc), use:

```
conda env create -f docs/environment.yml
source activate ipywidget_docs  # Linux and OS X
activate ipywidget_docs         # Windows
```

If you want to install the necessary packages with `pip` instead, use (omitting –user if working in a virtual environment):

```
pip install -r docs/requirements.txt --user
```

Once you have installed the required packages, you can build the docs with:

```
cd docs
make clean
make html
```

After that, the generated HTML files will be available at `build/html/index.html`. You may view the docs in your browser.

You can automatically check if all hyperlinks are still valid:

```
make linkcheck
```

Windows users can find `make.bat` in the `docs` folder.

You should also have a look at the Project Jupyter Documentation Guide.

1.16 Developer Release Procedure

To release a new version of the widgets on PyPI and npm, first checkout master and cd into the repo root. Make sure the version in `widget.py` matches the JS frontend version. Also check that the frontend version specified in `manager-base.js` (`version`) is correct.

1.16.1 Publish jupyter-js-widgets

```
# nuke the  `dist` and `node_modules`
git clean -fdx
edit manager-base.py to point "version" to next release of ipywidgets
npm version [patch/minor/major]
npm install
npm publish
```

1.16.2 Prepare widgetsnbextension npm module

- npm module

```
edit package.json to point to new jupyter-js-widgets version
npm version [patch/minor/major]
```

1.16.3 Here we commit our changes to the two package.json files

- python module

```
edit widgetsnbextension/_version.py (Remove dev from the version.  If it's the first␣
↪beta, use b1, etc...)
python setup.py sdist upload && python setup.py bdist_wheel upload

edit ipywidgets/_version.py (remove dev from the version and update the frontend␣
↪version requirement to match the one of jupyter-js-widgets)

Change install_requires to point to new widgetsnbextension version
python setup.py sdist upload && python setup.py bdist_wheel upload
commit and tag (ipywidgets) release
```

1.16.4 Back to dev

```
edit ipywidgets/_version.py (increase version and add dev tag)
edit widgetsnbextension/widgetsnbextension/_version.py (increase version and add dev␣
↪tag)
git add ipywidgets/_version.py
git add widgetsnbextension/widgetsnbextension/_version.py
git commit -m "Back to dev"
git push [upstream master]
git push [upstream] --tags
```

On GitHub

1. Go to https://github.com/ipython/ipywidgets/milestones and click “Close” for the released version.
2. Make sure patch, minor, and/or major milestones exist as appropriate.