
ipywidgets Documentation

Release 7.0.0a3

<https://jupyter.org>

May 28, 2017

1	Full Table of Contents	1
1.1	User Guide	1
1.2	Developer Docs	52

Full Table of Contents

User Guide

Installation

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

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`).

With conda

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

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

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, interact_manual
        import ipywidgets as widgets
```

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);  
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);  
True
```

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

```
In [5]: interact(f, x='Hi there!');  
'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)  
  
(True, 1.0)
```

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));  
(5, 20)
```

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

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*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));  
10
```

This examples clarifies how `interact` processes 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

Note that a dropdown is used if a list or a dict is given (signifying discrete choices), and a slider is used if a tuple is given (signifying a range).

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));
```

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));
```

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));
```

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));
```

5.0

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
```

5.5

Dropdown menus are constructed by passing a list 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']);  
'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});  
10
```

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

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

```
In [18]: w = interactive(f, a=10, b=20)
```

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

```
In [19]: type(w)
```

```
Out[19]: ipywidgets.widgets.interaction.interactive
```

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

```
In [20]: w.children
```

```
Out[20]: (<ipywidgets.widgets.widget_int.IntSlider at 0x7fd0ec5c92e8>,  
         <ipywidgets.widgets.widget_int.IntSlider at 0x7fd0ec63d5c0>,  
         <ipywidgets.widgets.widget_output.Output at 0x7fd0ec63df60>)
```

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

```
In [21]: from IPython.display import display  
         display(w)
```

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 gives you access to the current keyword arguments and return value of the underlying Python function. Note that unlike `interact`, the return value of the function will not be displayed automatically, but you can display a value inside the function with `IPython.display.display`.

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

```
In [22]: w.kwargs
```

```
Out[22]: {'a': 10, 'b': 20}
```

Here is the current return value of the function.


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

```
Out[23]: 30
```

Disabling continuous updates

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

```
In [24]: 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 [25]: %%time
          slow_function(1e6)
```

```
1000000 [0, 1, 2, 3, 11, 22, 101, 111, 121, 202, 212, 1001, 1111, 2002, 10001, 10101, 10201]
CPU times: user 788 ms, sys: 2.82 ms, total: 791 ms
Wall time: 810 ms
```

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

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

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

`interact_manual`

The `interact_manual` function provides a variant of interaction that 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 [27]: interact_manual(slow_function, i=FloatSlider(min=1e5, max=1e7, step=1e5));
```

A Jupyter Widget

`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 [28]: interact(slow_function, i=FloatSlider(min=1e5, max=1e7, step=1e5, continuous_update=False));
```

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

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 [29]: 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);

0.0 5.0
```

Simple Widget Introduction

What are widgets?

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

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.

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()
```

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)
```

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.

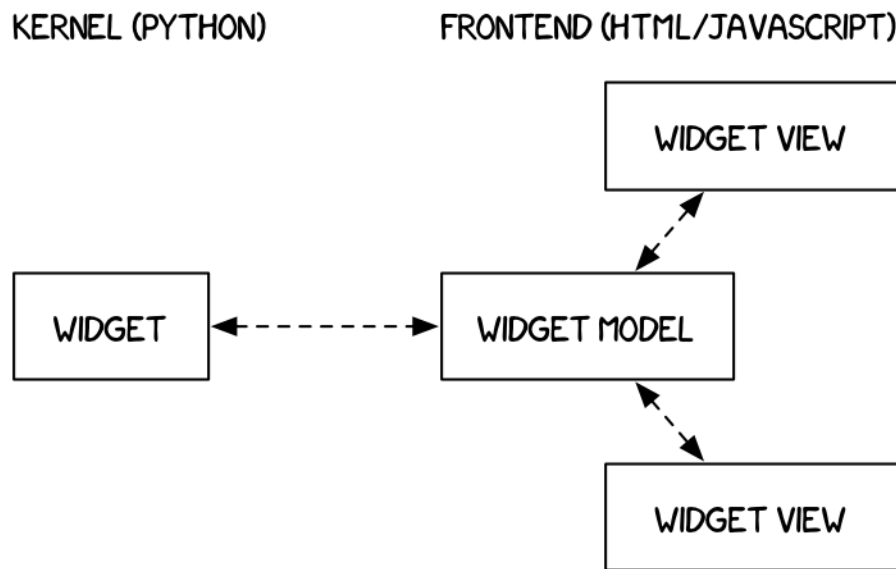


Fig. 1.1: Kernel & front-end diagram

Closing widgets

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

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

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

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)
```

```
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]: ['readout_format',
          'description',
          'readout',
          '_model_module_version',
          'min',
          '_model_module',
          '_range',
          '_view_module',
          'layout',
          '_view_name',
          'continuous_update',
          'msg_throttle',
          'orientation',
          'step',
          'max',
          '_dom_classes',
          '_model_name',
          'style',
          '_view_module_version',
          'value',
          'disabled']
```

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)
```

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]: a = FloatText()
         b = FloatSlider()
         display(a,b)

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

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()
```

Widget List

```
In [1]: import ipywidgets as widgets
```

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.

IntSlider

```
In [2]: widgets.IntSlider(  
        value=7,  
        min=0,  
        max=10,  
        step=1,  
        description='Test:',  
        disabled=False,  
        continuous_update=False,  
        orientation='horizontal',  
        readout=True,  
        readout_format='i',  
        slider_color='white'  
    )
```

A Jupyter Widget

FloatSlider

```
In [3]: widgets.FloatSlider(  
        value=7.5,  
        min=0,  
        max=10.0,  
        step=0.1,  
        description='Test:',  
        disabled=False,  
        continuous_update=False,  
        orientation='horizontal',  
        readout=True,  
        readout_format='.1f',  
        slider_color='white'  
    )
```

A Jupyter Widget

Sliders can also be **displayed vertically**.

```
In [4]: widgets.FloatSlider(  
        value=7.5,  
        min=0,  
        max=10.0,  
        step=0.1,  
        description='Test:',  
        disabled=False,  
        continuous_update=False,  
        orientation='vertical',  
        readout=True,  
        readout_format='.1f',  
        slider_color='white'  
    )
```

A Jupyter Widget

IntRangeSlider

```
In [5]: widgets.IntRangeSlider(  
        value=[5, 7],  
        min=0,  
        max=10,  
        step=1,  
        description='Test:',  
        disabled=False,  
        continuous_update=False,  
        orientation='horizontal',  
        readout=True,  
        readout_format='i',  
        slider_color='white',  
        color='black'  
    )
```

A Jupyter Widget

FloatRangeSlider

```
In [6]: widgets.FloatRangeSlider(  
        value=[5, 7.5],  
        min=0,  
        max=10.0,  
        step=0.1,  
        description='Test:',  
        disabled=False,  
        continuous_update=False,  
        orientation='horizontal',  
        readout=True,  
        readout_format='i',  
        slider_color='white',  
        color='black'  
    )
```

A Jupyter Widget

IntProgress

```
In [7]: widgets.IntProgress(  
        value=7,  
        min=0,  
        max=10,  
        step=1,  
        description='Loading:',  
        bar_style='', # 'success', 'info', 'warning', 'danger' or ''  
        orientation='horizontal'  
    )
```

A Jupyter Widget

FloatProgress

```
In [8]: widgets.FloatProgress(  
        value=7.5,  
        min=0,  
        max=10.0,  
        step=0.1,  
        description='Loading:',  
        bar_style='info',  
        orientation='horizontal'  
    )
```

A Jupyter Widget

BoundedIntText

```
In [9]: widgets.BoundedIntText(  
        value=7,  
        min=0,  
        max=10,  
        step=1,  
        description='Text:',  
        disabled=False  
    )
```

A Jupyter Widget

BoundedFloatText

```
In [10]: widgets.BoundedFloatText(  
        value=7.5,  
        min=0,  
        max=10.0,  
        step=0.1,  
        description='Text:',  
    )
```

```
        disabled=False,  
        color='black'  
    )
```

A Jupyter Widget

IntText

```
In [11]: widgets.IntText(  
        value=7,  
        description='Any:',  
        disabled=False  
    )
```

A Jupyter Widget

FloatText

```
In [12]: widgets.FloatText(  
        value=7.5,  
        description='Any:',  
        disabled=False,  
        color='black'  
    )
```

A Jupyter Widget

Boolean widgets

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

ToggleButton

```
In [13]: widgets.ToggleButton(  
        value=False,  
        description='Click me',  
        disabled=False,  
        button_style='', # 'success', 'info', 'warning', 'danger' or ''  
        tooltip='Description',  
        icon='check'  
    )
```

A Jupyter Widget

Checkbox

```
In [14]: widgets.Checkbox(  
        value=False,  
        description='Check me',  
        disabled=False  
    )
```


A Jupyter Widget

Valid

The valid widget provides a read-only indicator.

```
In [15]: widgets.Valid(
        value=False,
        description='Valid!',
        disabled=False
    )
```

A Jupyter Widget

Selection widgets

There are several widgets that can be used to display single selection lists, and two that can be used to select multiple values. All inherit from the same base class. You can specify the **enumeration of selectable options by passing a list** (options are either (label, value) pairs, or simply values for which the label is derived automatically). 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 used** when an item is selected (in this case, since dictionaries are unordered, the displayed order of items in the widget is unspecified).

Dropdown

```
In [16]: widgets.Dropdown(
        options=['1', '2', '3'],
        value='2',
        description='Number:',
        disabled=False,
        button_style='' # 'success', 'info', 'warning', 'danger' or ''
    )
```

A Jupyter Widget

The following is also valid:

```
In [17]: widgets.Dropdown(
        options={'One': 1, 'Two': 2, 'Three': 3},
        value=2,
        description='Number:',
    )
```

A Jupyter Widget

RadioButtons

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

A Jupyter Widget

Select

```
In [19]: widgets.Select(
        options=['Linux', 'Windows', 'OSX'],
        value='OSX',
        # rows=10,
        description='OS:',
        disabled=False
    )
```

A Jupyter Widget

SelectionSlider

```
In [20]: widgets.SelectionSlider(
        options=['scrambled', 'sunny side up', 'poached', 'over easy'],
        value='sunny side up',
        description='I like my eggs ...',
        disabled=False,
        continuous_update=False,
        orientation='horizontal',
        readout=True,
        # readout_format='i',
        # slider_color='black'
    )
```

A Jupyter Widget

SelectionRangeSlider

The value, index, and label keys are 2-tuples of the min and max values selected. The options must be nonempty.

```
In [21]: import datetime
        dates = [datetime.date(2015,i,1) for i in range(1,13)]
        options = [(i.strftime('%b'), i) for i in dates]
        widgets.SelectionRangeSlider(
            options=options,
            index=(0,11),
            description='Months (2015)'
        )
```

A Jupyter Widget

ToggleButtons

```
In [22]: widgets.ToggleButtons(
        options=['Slow', 'Regular', 'Fast'],
        description='Speed:',
        disabled=False,
        button_style='', # 'success', 'info', 'warning', 'danger' or ''
```

```
        tooltip='Description',  
        #     icon='check'  
    )
```

A Jupyter Widget

SelectMultiple

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

```
In [23]: widgets.SelectMultiple(  
        options=['Apples', 'Oranges', 'Pears'],  
        value=['Oranges'],  
        #rows=10,  
        description='Fruits',  
        disabled=False  
    )
```

A Jupyter Widget

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(  
        value='Hello World',  
        placeholder='Type something',  
        description='String:',  
        disabled=False  
    )
```

A Jupyter Widget

Textarea

```
In [25]: widgets.Textarea(  
        value='Hello World',  
        placeholder='Type something',  
        description='String:',  
        disabled=False  
    )
```

A Jupyter Widget

Label

The `Label` widget is useful if you need to build a custom description next to a control.

```
In [26]: description = widgets.Label(value="My Control")
        # control could be any custom control
        control = widgets.IntSlider()
        widgets.HBox([description, control])
```

A Jupyter Widget

HTML

```
In [27]: widgets.HTML(
        value="Hello <b>World</b>",
        placeholder='Some HTML',
        description='Some HTML',
        disabled=False
    )
```

A Jupyter Widget

HTML Math

```
In [28]: widgets.HTMLMath(
        value=r"Some math and <i>HTML</i>: \((x^2)\) and $$\frac{x+1}{x-1}$$",
        placeholder='Some HTML',
        description='Some HTML',
        disabled=False
    )
```

A Jupyter Widget

Image

```
In [29]: file = open("images/WidgetArch.png", "rb")
        image = file.read()
        widgets.Image(
            value=image,
            format='png',
            width=300,
            height=400,
        )
```

A Jupyter Widget

Button

```
In [30]: widgets.Button(
        description='Click me',
        disabled=False,
        button_style='', # 'success', 'info', 'warning', 'danger' or ''
        tooltip='Click me',
        icon='check'
    )
```

A Jupyter Widget

Play (Animation) widget

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

```
In [31]: play = widgets.Play(  
        #     interval=10,  
        value=50,  
        min=0,  
        max=100,  
        step=1,  
        description="Press play",  
        disabled=False  
    )  
    slider = widgets.IntSlider()  
    widgets.jslink((play, 'value'), (slider, 'value'))  
    widgets.HBox([play, slider])
```

A Jupyter Widget

Date picker

The date picker widget works in Chrome and IE Edge, but does not currently work in Firefox or Safari because they do not support the HTML date input field.

```
In [32]: widgets.DatePicker(  
        description='Pick a Date'  
    )
```

A Jupyter Widget

Color picker

```
In [33]: widgets.ColorPicker(  
        concise=False,  
        description='Pick a color',  
        value='blue'  
    )
```

A Jupyter Widget

Controller

```
In [34]: widgets.Controller(  
        index=0,  
    )
```

A Jupyter Widget

Layout widgets

Box

HBox

```
In [35]: items = [widgets.Label(str(i)) for i in range(4)]
         widgets.HBox(items)
```

A Jupyter Widget

VBox

```
In [36]: items = [widgets.Label(str(i)) for i in range(4)]
         widgets.HBox([widgets.VBox([items[0], items[1]]), widgets.VBox([items[2], items[3]])])
```

A Jupyter Widget

Accordion

```
In [37]: accordion = widgets.Accordion(children=[widgets.IntSlider(), widgets.Text()])
         accordion.set_title(0, 'Slider')
         accordion.set_title(1, 'Text')
         accordion
```

A Jupyter Widget

Tabs

```
In [38]: list = ['P0', 'P1', 'P2', 'P3', 'P4']
         children = [widgets.Text(description=name) for name in list]
         tab = widgets.Tab(children=children)
         tab
```

A Jupyter Widget

Widget Events

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)
```

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)
```

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')
```

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 in the kernel

The first method is to use the `link` and `dlink` functions from the `traitlets` module. This only works if we are interacting with a live kernel.

```
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)

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)
```

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()
```


Registering callbacks to trait changes in the kernel

Since attributes of widgets on the Python side are traitlets, you can register handlers to the change events whenever the model gets updates from the front-end.

The handler passed to the decorator will be called with one change argument. The change object holds at least a `type` key and a `name` key, corresponding respectively to the type of notification and the name of the attribute that triggered the 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.

```
In [11]: caption = widgets.Label(value='The values of range1 and range2 are synchronized')
         slider = widgets.IntSlider(min=-5, max=5, value=1, description='Slider')

         def handle_slider_change(change):
             caption.value = 'The slider value is ' + (
                 'negative' if change.new < 0 else 'nonnegative'
             )

         slider.observe(handle_slider_change, names='value')

         display(caption, slider)
```

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.

Javascript links persist when embedding widgets in html web pages without a kernel.

```
In [12]: 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)

In [13]: caption = widgets.Label(value='Changes in source_range values are reflected in target_range')
         source_range, target_range1 = widgets.IntSlider(description='Source range'), \
             widgets.IntSlider(description='Target range 1')
         dl = widgets.jsdlink((source_range, 'value'), (target_range1, 'value'))
         display(caption, source_range, target_range1)
```

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

```
In [14]: # l.unlink()
         # dl.unlink()
```

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.

The `layout` attribute.

Jupyter interactive widgets have 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 Button, Layout

        b = Button(description='(50% width, 80px height) button',
                  layout=Layout(width='50%', height='80px'))
        b
```

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

```
In [2]: Button(description='Another button with the same layout', layout=b.layout)
```

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 [3]: from ipywidgets import HBox, Label, IntSlider

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

Natural sizes, and arrangements 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 [4]: from ipywidgets import Button, HBox, VBox

        words = ['correct', 'horse', 'battery', 'staple']
        items = [Button(description=w) for w in words]

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

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 [5]: from ipywidgets import IntSlider, Label
In [6]: IntSlider(description=r'\(\int_0^t f\)\')
In [7]: Label(value=r'\(e=mc^2\)\')
```

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.

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` widget 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.

Acknowledgement

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

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.

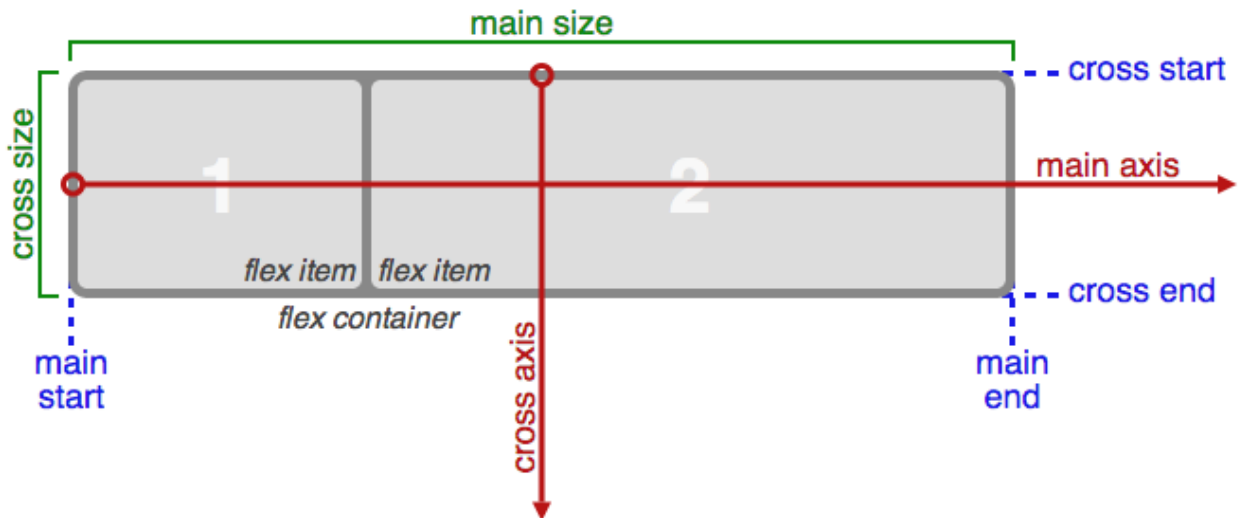


Fig. 1.2: Flexbox

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).
- `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.

Fig. 1.4: Item

Properties of the items

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 [8]: from ipywidgets import Layout, Button, Box

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%')

words = ['correct', 'horse', 'battery', 'staple']
items = [Button(description=w, layout=items_layout, button_style='danger') for w in words]
box = Box(children=items, layout=box_layout)
box
```

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

```
In [9]: from ipywidgets import Layout, Button, Box

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
```

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 [10]: from ipywidgets import Layout, Button, Box, FloatText, Textarea, Dropdown, Label,

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
```

A more advanced example: a carousel.

```
In [11]: from ipywidgets import Layout, Button, Box

item_layout = Layout(height='100px', min_width='40px')
items = [Button(layout=item_layout, description=str(i), button_style='warning') f
box_layout = Layout(overflow_x='scroll',
                    border='3px solid black',
                    width='500px',
                    height='',
                    flex_direction='row',
                    display='flex')

carousel = Box(children=items, layout=box_layout)
VBox([Label('Scroll horizontally:'), carousel])
```

Predefined styles

If you wish the styling of widgets to make use of colors and styles defined by the environment (to be consistent with e.g. a notebook theme), many widgets enable choosing in a list of pre-defined styles.

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 [12]: from ipywidgets import Button

Button(description='Danger Button', button_style='danger')
```


The `style` attribute

While the `layout` attribute only exposes layout-related CSS properties for the top-level DOM element of widgets, the

`style` attribute is used to expose non-layout related styling attributes of widgets.

However, the properties of the `style` attribute are specific to each widget type.

```
In [13]: b1 = Button(description='Custom color')
         b1.style.button_color = 'lightgreen'
         b1
```

Just like the `layout` attribute, widget styles can be assigned to other widgets.

```
In [14]: b2 = Button()
         b2.style = b1.style
         b2
```

Widget styling attributes are specific to each widget type.

```
In [15]: s1 = IntSlider(description='Blue handle')
         s1.style.handle_color = 'lightblue'
         s1
```

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

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.

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.

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 HelloWorldWidget(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.

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>
```

Test

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

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

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 HelloWorld(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 associated 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>
```

Test

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

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

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.

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.

```
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')
```

2. HBox (VBox)

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

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

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

4. Dropdown resize

```
In [5]: dd = Dropdown(description='Foobar', options=['foo', 'bar'])
        dd
```

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

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

5. Colorpicker alignment, concise and long version

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

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

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

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

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

```
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', height='200px')])

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

7. Tabs

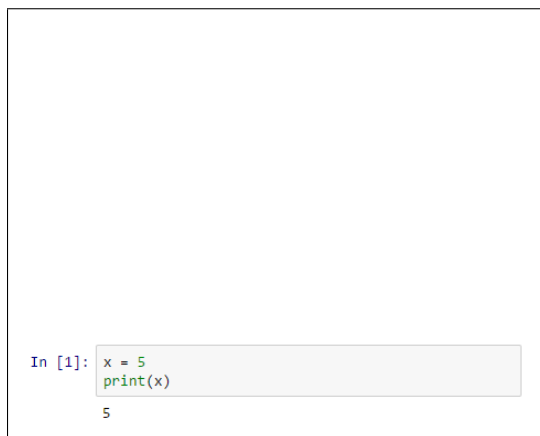
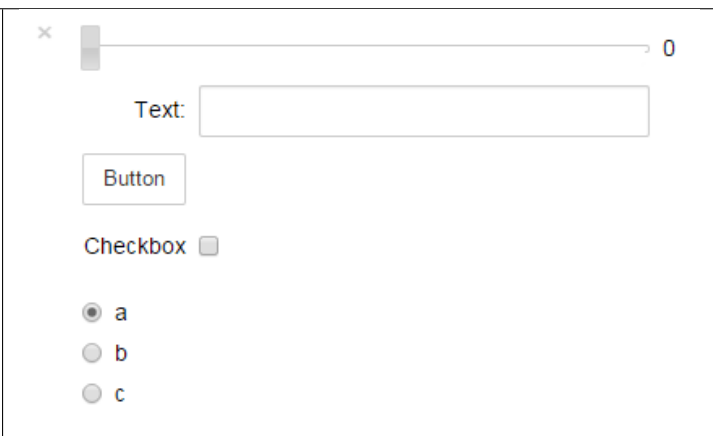
```
In [17]: t = Tab(children=[FloatText(), IntSlider()], _titles={0: 'Text', 1: 'Slider'})
         t

In [18]: t.selected_index = 1
```

Low Level Widget Tutorial

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.

 <pre>In [1]: x = 5 print(x) 5</pre>	
<p>Before the widgets, this was just the segmentation of code and results from executing those segments.</p>	<p>Widgets further decrease the distance between the user and their data by allowing UI interactions to directly manipulate data in the kernel.</p>

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.

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.

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 each other. If the WidgetModel is changed in the front-end, the Widget receives the same change in the kernel. Vice 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.

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)
```

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

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.

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 by 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.

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 message'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.

Widget skeleton

```
In [3]: %%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);
```

<IPython.core.display.Javascript object>

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:

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

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

JavaScript:

```

define('mywidget', ['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 that is loaded is the base widget module.

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.model.save_changes()`; changes the model.

Use the view method `touch` instead of `model.save_changes` 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.

Serialization of widget attributes

Widget trait attributes tagged with `sync=True` are synchronized with the JavaScript model instance on the JavaScript side. For this reason, they need to be serialized into `json`.

By default, basic Python types such as `int`, `float`, `list` and `dict` are simply be mapped to `Number`, `Array` and `Object`. For more complex types, serializers and de-serializers mustbe specified on both the Python side and the JavaScript side.

Custom serialization and de-serialization on the Python side

In many cases, a custom serialization must be specified for trait attributes. For example

- if the trait attribute is not json serializable
- if the trait attribute contains data that is not needed by the JavaScript side.

Custom serialization can be specified for a given trait attribute through the `to_json` and `from_json` metadata. These must be functions that take two arguments

- the value to be [de]serialized
- the instance of the underlying widget model.

In most cases, the second argument is not used in the implementation of the serializer.

Example

For example, in the case of the `value` attribute of the `DatePicker` widget, the declaration is

```
value = Datetime(None, allow_none=True).tag(sync=True, to_json=datetime_to_json, from_
↪ json=datetime_from_json)
```

where `datetime_to_json(value, widget)` and `datetime_from_json(value, widget)` return or handle json data-structures that are amenable to the front-end.

The case of parent child relationships between widget models

When a widget model holds other widget models, you must use the serializers and deserializers provided in ipywidgets packed into the `widget_serialization` dictionary.

For example, the `HBox` widget declares its `children` attribute in the following fashion:

```
from .widget import widget_serialization

[...]

children = Tuple().tag(sync=True, **widget_serialization)
```

The actual result of the serialization of a widget model is a string holding the widget id prefixed with `"IPY_MODEL_"`.

Custom serialization and de-serialization on the JavaScript side

In order to mirror the custom serializer and deserializer of the Python side, symmetric methods must be provided on the JavaScript side.

On the JavaScript side, serializers are specified through the `serializers` class-level attribute of the widget model.

They are generally specified in the following fashion, extending the dictionary of serializers and deserializers of the base class. In the following example, which comes from the `DatePicker`, the deserializer for the `value` attribute is specified.

```
static serializers = _.extend({
  value: {
    serialize: serialize_datetime,
    deserialize: deserialize_datetime
  }
}, BaseModel.serializers)
```

Custom serializers are functions taking two arguments: the value of the object to [de]serialize, and the widget manager. In most cases, the widget manager is actually not used.

Widget Messaging

The protocol for

- instantiating jupyter widgets
- synchronizing widget state between the front-end and the back-end companion objects
- sending custom messages between these objects

Is entirely based upon the `Comm` section of the Jupyter kernel protocol.

For more details on comms *per se*, we refer to the [relevant section of the specification for the Jupyter kernel protocol](#).

Implementation of a backend for the Jupyter widgets protocol.

Jupyter widget libraries built upon ipywidgets tend to have a large part of their code-base in JavaScript, since this is where the logic for drawing and rendering widgets resides. The Python side mostly consists in a declaration of the widget model attributes.

A byproduct of the *thin backend* of widget libraries is that once the widget protocol is implemented for another kernel, all the widgets and custom widget libraries can be reused in that language.

Therefore, in this documentation, we concentrate on the viewpoint of a kernel author implementing a jupyter widget backend.

The `jupyter.widget comm` target

Jupyter interactive widgets define two `comm` targets

- `jupyter.widget`
- `jupyter.widget.version`

The first one is the target handling all the widget state synchronization as well as the custom messages. The other target is meant for a version check between the front-end and the backend, and can be ignored from now.

Instanciating widgets from the front-end and the backend

Reception of a “`comm_open`” message

Upon reception of the `comm_open` message for target `jupyter.widget`

```
{
  'comm_id' : 'u-u-i-d',
  'target_name' : 'jupyter.widget',
  'data' : {
    'widget_class': 'some.string'
  }
}
```

The type of widget to be instantiated is determined with the `widget_class` string.

In the python implementation, this string is actually the key in a registry of widget types. In the case where the key is not found, it is parsed as a `module + class` string.

In the Python implementation of the backend, widget types are registered in the dictionary with the `register` decorator. For example the integral progress bar is registered with `register('Jupyter.IntProgress')`.

Emmission of the “comm_open“ message upon instantiation of a widget

Symmetrically, when instantiating a widget in the backend, a `comm_open` message is sent to the front-end.

```
{
  'comm_id' : 'u-u-i-d',
  'target_name' : 'jupyter.widget',
  'data': {
    '[serialized widget state]'
  }
}
```

The type of widget to be instantiated in the front-end is determined by the `_model_name`, `_model_module` and `_model_module_version` keys in the state, which respectively stand for the name of the class that must be instantiated in the frontend, the javascript module where this class is defined, and a semver range for that module.

Sending updates of the state for a widget model

```
{
  'comm_id' : 'u-u-i-d',
  'data' : {
    'method': 'state',
    'state': '[serialized widget state or portion of the serialized widget sate]',
    'buffers': '[optional list of keys for attributes sent in the form of binary_
↪buffers]'
  }
}
```

Comm messages for state synchronization optionally contain a list binary buffers. If this list is not empty, a corresponding list of strings must be provided in the `data` message providing the names for these buffers.

The front-end will unpack these buffer and insert them in the state for the specified keys.

Sending custom messages

In the Python implementation, the base widget class provides a means to send raw comm messages directly. `Widget.send(content, buffers=None)` will produce a message of the form

```
{
  'comm_id' : 'u-u-i-d',
  'data' : {
    'method': 'custom',
    'content': 'the specified content',
    'buffers': 'the provided buffers'
  }
}
```

```
}
}
```

Receiving data synchronization messages

Up on updates of the JavaScript model state, the front-end emits widget state patches messages

```
{
  'comm_id' : 'u-u-i-d',
  'data' : {
    'method': 'backbone',
    'sync_data': 'the patch to the data',
    'buffers': 'optional buffer names list'
  }
}
```

The `sync_data` contains the serialized state of the changed model attributes in the form of a dictionary.

Optionally, the message may specify a list of buffer names. When provided, the corresponding binary buffers in the zmq message should be appended in the `sync_data` dictionary with the keys specified in the `buffers` list.

State requests

In the case of a front-end connecting to a running kernel where widgets have already been instantiated, it may send a request state message, of the form

```
{
  'comm_id' : 'u-u-i-d',
  'data' : {
    'method': 'request_state'
  }
}
```

The expected response to that message is a regular `update` message as specified above containing the entirety of the widget model state.

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.

Static assets

In the case of the classic Jupyter notebook, static assets are made available to the Jupyter notebook in the form of a Jupyter extensions. JavaScript bundles are copied in a directory accessible through the `nbextensions/` handler. Nbextensions also have a mechanism for running your code on page load. This can be set using the `install-nbextension` command.

Distribution

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.

Asynchronous widget updates

You may want to pause your Python code to wait for an update from a widget. Typically this would be hard to do since running Python code blocks any widget messages from the frontend until the Python code is done.

We’ll do this in two approaches: using the event loop integration, and using plain generator functions.

Event loop integration

If we take advantage of the event loop integration IPython offers, we can have a nice solution using the `async/await` syntax in Python 3.

First we register a new event loop integration. This registration will not be needed once <https://github.com/ipython/ipykernel/pull/246> is merged and released.

```
In [1]: import asyncio
        from ipykernel.eventloops import register_integration

        @register_integration('asyncio')
        def loop_asyncio(kernel):
            '''Start a kernel with asyncio event loop support.'''
            loop = asyncio.get_event_loop()

            def kernel_handler():
                loop.call_soon(kernel.do_one_iteration)
                loop.call_later(kernel._poll_interval, kernel_handler)

            loop.call_soon(kernel_handler)
            try:
                if not loop.is_running():
                    loop.run_forever()
            finally:
                loop.run_until_complete(loop.shutdown_asyncgens())
                loop.close()
```

Next we invoke our asyncio event loop.

```
In [2]: %gui asyncio
```

We define a new function that returns a future for when a widget attribute changes.

```
In [3]: import asyncio
        def wait_for_change(widget, value):
            future = asyncio.Future()
            def getvalue(change):
                # make the new value available
                future.set_result(change.new)
                widget.unobserve(getvalue, value)
            widget.observe(getvalue, value)
            return future
```


And we finally get to our function where we will wait for widget changes. We'll do 10 units of work, and pause after each one until we observe a change in the widget. Notice that the widget's value is available to us, since it is what the `wait_for_change` future has as a result.

Run this function, and change the slider 10 times.

```
In [4]: from ipywidgets import IntSlider
        slider = IntSlider()

        async def f():
            for i in range(10):
                print('did work %s'%i)
                x = await wait_for_change(slider, 'value')
                print('async function continued with value %s'%x)
        asyncio.ensure_future(f())

        slider
```

A Jupyter Widget

Generator approach

If you can't take advantage of the `async/await` syntax, or you don't want to modify the event loop, you can also do this with generator functions.

First, we define a decorator which hooks a generator function up to widget change events.

```
In [5]: from functools import wraps
        def yield_for_change(widget, attribute):
            """Pause a generator to wait for a widget change event.

            This is a decorator for a generator function which pauses the generator on yield
            until the given widget attribute changes. The new value of the attribute is
            sent to the generator and is the value of the yield.
            """
            def f(iterator):
                @wraps(iterator)
                def inner():
                    i = iterator()
                    def next_i(change):
                        try:
                            i.send(change.new)
                        except StopIteration as e:
                            widget.unobserve(next_i, attribute)
                    widget.observe(next_i, attribute)
                    # start the generator
                    next(i)
                return inner
            return f
```

```
did work 0
```

Then we set up our generator.

```
In [6]: from ipywidgets import IntSlider, VBox, HTML
        slider2=IntSlider()
```

```
@yield_for_change(slidebar2, 'value')
def f():
    for i in range(10):
        print('did work %s'%i)
        x = yield
        print('generator function continued with value %s'%x)

f()

slidebar2

did work 0
A Jupyter Widget
```

Modifications

The above two approaches both waited on widget change events, but can be modified to wait for other things, such as button event messages (as in a “Continue” button), etc.

Embedding Jupyter Widgets in Other Contexts than the Notebook

Jupyter interactive widgets can be serialized and embedded into

- static web pages
- sphinx documentation
- html-converted notebooks on nbviewer

Embedding Widgets in HTML Web Pages

The notebook interface provides a context menu for generating an HTML snippet that can be embedded into any static web page:

The context menu provides three actions

- Save Notebook with Widgets
- Download Widget State
- Embed Widgets

Embeddable HTML Snippet

The last option, `Embed widgets`, provides a dialog containing an HTML snippet which can be used to embed Jupyter interactive widgets into any web page.

This HTML snippet is composed of multiple `<script>` tags:

- The first script tag loads a custom widget manager from the `unpkg` cdn.
- The second script tag contains the state of all the widget models currently in use. It has the mime type `application/vnd.jupyter.widget-state+json`.

The JSON schema for the content of that script tag is found in the `jupyter-widgets-schema` npm package.

- The following script tags correspond to the views which you want to display in the web page. They have the mime type `application/vnd.jupyter.widget-view+json`.

The *Embed Widgets* action currently creates such a tag for each view displayed in the notebook at this time.

The JSON schema for the content of that script tag is found in the `jupyter-widgets-schema` npm package.

If you want to lay out these script tags in a custom fashion or only keep some of them, you can change their location in the DOM when including the snippet into a web page.

Widget State JSON

The second option, `Download Widget State`, triggers the downloading of a JSON file containing the serialized state of all the widget models currently in use, corresponding to the same JSON schema.

Embedding Widgets in the Sphinx HTML Documentation

As of ipywidgets 6.0, Jupyter interactive widgets can be rendered and interacted with in sphinx html documentation. Two means of achieving this are provided:

Using the Jupyter Sphinx Extension

The `jupyter_sphinx` extension enables jupyter-specific features in sphinx. It can be install with `pip` and `conda`.

In the `conf.py` sphinx configuration file, add `jupyter_sphinx.embed_widgets` to list of enabled extensions.

Two directives are provided: `ipywidgets-setup` and `ipywidgets-display`.

`ipywidgets-setup` code is used to run potential boilerplate and configuration code prior to running the display code. For example:

```
.. ipywidgets-setup::

    from ipywidgets import VBox, jsdlink, IntSlider, Button

.. ipywidgets-display::

    s1, s2 = IntSlider(max=200, value=100), IntSlider(value=40)
    b = Button(icon='legal')
    jsdlink((s1, 'value'), (s2, 'max'))
    VBox([s1, s2, b])
```

In the case of the `ipywidgets-display` code, the *last statement* of the code-block should contain the widget object you wish to be rendered.

Quoting the `jupyter_sphinx` readme,

Widgets rendered on the same page use the same widget manager. As a consequence, they can be linked with each other via JavaScript link widgets. However, no kernel is connect and therefore, interaction with the backend will not happen.

Using the nbsphinx Project

The `nbsphinx` sphinx extension provides a source parser for `*.ipynb` files. Custom Sphinx directives are used to show Jupyter Notebook code cells (and of course their results) in both HTML and LaTeX output.

In the case of the HTML output, Jupyter Interactive Widgets are also supported. However, it is a requirement that the notebook was correctly saved with the special “Save Notebook with Widgets” action in the widgets menu.

Rendering Interactive Widgets on nbviewer

If your notebook was saved with the special “Save Notebook with Widgets” action in the Widgets menu, interactive widgets displayed in your notebook should also be rendered on nbviewer.

See e.g. the [Widget List](#) example from the documentation.

The Case of Custom Widget Libraries

Custom widgets can also be rendered on nbviewer, static HTML and RTD documentation. An illustration of this is the <http://jupyter.org/widgets> gallery.

The widget embedder attempts to fetch the model and view implementation of the custom widget from the npmjs CDN, <https://unpkg.com>. The URL that is requested for, e.g. the `bqplot` module name, with the semver range `^2.0.0` is

```
https://unpkg.com/bqplot@^2.0.0/dist/index.js
```

which holds the webpack bundle for the `bqplot` library.

The [widget-cookiecutter](#) template project contains a template project for a custom widget library following the best practices for authoring widgets, which ensure that your custom widget library can render on nbviewer.

Using `jupyter-js-widgets` in web contexts

The core `jupyter-js-widgets` library, the JavaScript package of `ipywidgets`, is agnostic to the context in which it is used (Notebook, JupyterLab, static web page). For each context, we specialize the base widget manager implemented in `jupyter-js-widgets` to provide the logic for

- where widgets should be displayed,
- how to retrieve information about their state.

Specifically:

- `widgetsnbextension` provides the implementation of a specialized widget manager for the Classic Notebook, and the packaging logic as a notebook extension.
- `jupyterlab_widgets` provides the implementation of a specialized widget manager for the context of JupyterLab, and the packaging logic as a lab extension.
- The embed manager implemented in `jupyter-js-widgets` is a specialization of the base widget manager used for the static embedding of widgets used by the `Sphinx` extension, `nbviewer`, and the “Embed Widgets” command discussed above.

We provide additional examples of specializations of the base widget manager implementing other usages of `jupyter-js-widgets` in web contexts.

1. The `web1` example is a simplistic example showcasing the use of `jupyter-js-widgets` in a web context.
2. The `web2` example is a simple example making use of the `application/vnd.jupyter.widget-state+json` mime type.
3. The `web3` example showcases how communication with a Jupyter kernel can happen in a web context outside of the notebook or jupyterlab contexts.
4. The `web4` example is a tests for the embed widget manager.

- The `web5` example makes use of the `tmpnb` service to spawn a Jupyter server, request a kernel from this server and implement the same feature as the `web3` example.

Contributing

We appreciate contributions from the community.

We follow the [IPython Contributing Guide](#) and [Jupyter Contributing Guides](#).

ipywidgets changelog

A summary of changes in ipywidgets. For more detailed information, see [GitHub](#).

7.0

Major user-visible changes in ipywidgets 7.0 include:

- The `Label` widget is now right-aligned and has no width restriction: (#1269)
- Using function annotations to specify interact controls for a function is now deprecated and will be removed in a future version of ipywidgets. (#1292)
- Widgets are now displayed in the output area in the classic notebook and are treated as any other output. This allows the widgets to work more naturally with other cell output. To delete a widget, clear the output from the cell. Output from functions triggered by a widget view is appended to the output area that contains the widget view. This means that printed text will be appended to the output, and calling `clear_output()` will delete the entire output, including the widget view. (#1274, #1353)
- The `Select` widget now is a listbox instead of a dropdown, reverting back to the pre-6.0 behavior. (#1238)
- The `Select` and `SelectMultiple` widgets now have a `rows` attribute for the number of rows to display, consistent with the `Textarea` widget. The `layout.height` attribute overrides this to control the height of the widget. (#1250)
- Selection widgets (`Select`, `Dropdown`, etc.) now coerce their `options` argument to a tuple of `(label, value)` pairs. This makes it clearer the order in which the options will be presented. New `.value`, `.label`, and `.index` traits make it easier to observe or access the selected option. (#1262)
- The `layout` and `style` traits can be set with a dictionary for convenience, which will automatically converted to a `Layout` or `Style` object. (#1253)

```
IntSlider(layout={'width': '100%'}, style={'handle_color': 'lightgreen'})
```

- Removed the version validation check since it was causing too many false warnings about the widget javascript not being installed or the wrong version number. It is now up to the user to ensure that the ipywidgets and `widgetsnbextension` packages are compatible. (#1219)
- `Play` range is now inclusive (max value is max, instead of max-1), to be consistent with `Sliders`
- A refactoring of the `text`, `slider`, `slider range`, and `progress` widgets in resulted in the `BoundedIntText` and `BoundedFloatText` losing their `step` attribute (which was previously ignored), and a number of these widgets changing their `_model_name` and/or `_view_name` attributes (#1290)
- The `Checkbox` description is now on the right of the checkbox and is clickable. The `Checkbox` widget has a new `indent` attribute (defaults to `True`) to line up nicely with controls that have descriptions. To make the checkbox align to the left, set `indent` to `False`. (#1346)
- The `Play` widget now has an optional repeat toggle button (visible by default). (#1190)

- A new Password widget, which behaves exactly like the Text widget, but hides the typed text: `Password()`. (#1310)
- A new SelectionRangeSlider widget for selecting ranges from ordered lists of objects. For example, this enables having a slider to select a date range. (#1356)
- The documentation theme is changed to the new standard Jupyter theme. (#1363)

Major changes developers should be aware of include:

- The python `@register` decorator for widget classes no longer takes a string argument, but registers a widget class using the `_model_*` and `_view_*` traits in the class. Using the decorator as `@register('name')` is deprecated and should be changed to just `@register`. #1228, #1276
- Widgets will now need correct `_model_module` and `_view_module` Unicode traits defined.
- Selection widgets now sync the index of the selected item, rather than the label. (#1262)
- The widget protocol was significantly overhauled. The new widget messaging protocol (version 2) is specified in the [version 2 protocol documentation](#).
- Custom serializers in Javascript are now synchronous, and should return a snapshot of the widget state. The default serializer makes a copy of JSONable objects. (#1270)
- Custom serializers in either Python or Javascript can now return a structure which contains binary buffers. If a binary buffer is in the serialized data structure, the message will be synced in binary, which is much more efficient. (#1194)
- A custom serializer is given the widget instance as its second argument, and a custom deserializer is given the widget manager as its second argument.
- The widget manager `clear_state` method no longer has a `commlessOnly` argument. All models in the widget manager will be closed and cleared when `clear_state` is called. (#1354)

6.0

Major user-visible changes in ipywidgets 6.0 include:

- Rendering of Jupyter interactive widgets in various web contexts
sphinx documentation: <http://ipywidgets.readthedocs.io/en/latest/examples/Widget%20List.html> nbviewer:
<http://nbviewer.jupyter.org/github/jupyter-widgets/ipywidgets/blob/master/docs/source/examples/Widget%20List.ipynb>
Static web pages: <http://jupyter.org/widgets>
- Addition of a DatePicker widget in the core widget collection.
- Changes to the automatic control generation syntax in `@interact`, inspired by the Sage interact syntax.
- Removal of APIs which had been deprecated in 5.0, including top-level styling in attributes of `DOMWidgets` and some corner cases in the behavior of `@interact`.
- A new API for custom styling of widgets is provided, through a top-level `style` attribute. For example, the color of a slider handler can be set by `slider.style.handle_color`.
- Removal of the Proxy and PlaceProxy widgets.
- Removal of the deprecated `FlexBox` widget. Use the `Box`, `HBox`, or `VBox` widgets instead. Various flex properties can be set using the `layout` attribute.
- Removal of the deprecated `Latex` widget. Use the new `HTMLMath` widget with Latex math inside `$` or `$$` delimiters instead.
- Removal of the deprecated layout properties on a widget such as `.width`, `.height`, etc. Use the `layout` attribute with a `Layout` widget to manage various layout properties instead.

- The `Label` widget now has styling to make it consistent with labels on various widgets. To have freeform text with mathematics, use the new `HTMLMath` widget.
- Removal of the `button_style` attribute of the `Dropdown` widget
- Addition of an `OutputWidget` for capturing output and rich display objects. `@interact` has changed to use an `OutputWidget` for function output instead of overwriting the output area of a cell.
- The `jupyter-js-widgets` Javascript implementation now relies on the `PhosphorJS` framework for the management of rich layout and a better integration of `JupyterLab`.
- Numerous bug fixes.

Note for custom widget authors:

ipywidgets 6.0 breaks backward compatibility with respect to the handling of default values of the JavaScript side. Now, the default values for core widget models are specified with a `default()` method returning a dictionary instead of a `default` dictionary attribute. If you want your library to be backwards compatible with ipywidgets 5.x, you could use `._result` like this:

```
...
defaults: function() {
    return _.extend(_.result(this, 'widgets.DOMWidgetModel.prototype.defaults'), {
        ....
    })
},
...
```

This should not have an impact when using your custom widgets in the classic notebook, but will be really important when deploying your interactive widgets in web contexts.

5.x

4.1.x

4.1.1

4.1.0

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.

Developer Docs

Developer Install

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](#).

Steps

1. Clone the repo:

```
git clone https://github.com/jupyter-widgets/ipywidgets
```

2. Navigate into the cloned repo and install:

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

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 ..
```

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

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.

Testing

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

Building documentation

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

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.

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](#).

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 of `jupyter-js-widgets` matches the semver range `__frontend_version__` specified in `ipywidgets/_version.py`.

Publish jupyter-js-widgets

```
# nuke the `dist` and `node_modules`
git clean -fdx
npm version [patch/minor/major]
npm install
npm publish --tag next
```

widgetsnextension

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

Edit `widgetsnextension/_version.py` (Remove dev from the version. If it's the first beta, use b1, etc...)

```
python setup.py sdist
python setup.py bdist_wheel --universal
twine upload dist/*
```

JupyterLab

Edit the `package.json` to have `jupyter-js-widgets` point to the correct version.

```
npm version patch/minor/major
npm install
npm run build
npm publish
python setup.py sdist
```

```
python setup.py bdist_wheel --universal
twine upload dist/*
```

ipywidgets

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

Change `setup.py install_requires` parameter to point to new `widgetsnbextension` version

```
python setup.py sdist
python setup.py bdist_wheel --universal
twine upload dist/*
```

Push changes back

commit and tag (ipywidgets) release

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/jupyter-widgets/ipywidgets/milestones> and click “Close” for the released version.
2. Make sure patch, minor, and/or major milestones exist as appropriate.