ipywidgets Documentation

Release 7.6.2

https://jupyter.org

Dec 27, 2020

CONTENTS

ONE

FULL TABLE OF CONTENTS

1.1 User Guide

1.1.1 Installation

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

In most cases, installing the Python `ipywidgets` package will also automatically configure classic Jupyter Notebook and JupyterLab 3.0 to display ipywidgets. With pip, do:

```
pip install ipywidgets
```

or with conda, do:

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

Installing in classic Jupyter Notebook

Most of the time, installing `ipywidgets` automatically configures Jupyter Notebook to use widgets. The `ipywidgets` package does this by depending on the `widgetsnbextension` package, which configures the classic Jupyter Notebook to display and use widgets. If you have an old version of Jupyter Notebook installed, you may need to manually enable the ipywidgets notebook extension with:

```
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`).

If your Jupyter Notebook and the IPython kernel are installed in different environments (for example, separate environments are providing different Python kernels), then the installation requires two steps:

1. Install the `widgetsnbextension` package in the environment containing the Jupyter Notebook server.

2. Install `ipywidgets` in each kernel's environment that will use ipywidgets.

For example, if using conda environments, with Jupyter Notebook installed on the `base` environment and the kernel installed in an environment called `py36`, the commands are:

```
conda install -n base -c conda-forge widgetsnbextension
conda install -n py36 -c conda-forge ipywidgets
```

Installing in JupyterLab 3.0

Most of the time, installing `ipywidgets` automatically configures JupyterLab 3.0 to use widgets. The `ipywidgets` package does this by depending on the `jupyterlab_widgets` package, version 1.0, which configures JupyterLab 3 to display and use widgets.

If your JupyterLab and the IPython kernel are installed in different environments (for example, separate environments are providing different Python kernels), then the installation requires two steps:

1. Install the `jupyterlab_widgets` package (version 1.0 or later) in the environment containing JupyterLab.

2. Install `ipywidgets` in each kernel's environment that will use ipywidgets.

For example, if using conda environments, with JupyterLab installed on the `base` environment and the kernel installed in an environment called `py36`, the commands are:

```
conda install -n base -c conda-forge jupyterlab_widgets
conda install -n py36 -c conda-forge ipywidgets
```

Installing into JupyterLab 1 or 2

To install the JupyterLab extension into JupyterLab 1 or 2, you also need to run the command below in a terminal which requires that you have nodejs installed.

For example, if using conda environments, you can install nodejs with:

```
conda install -c conda-forge nodejs
```

Then you can install the labextension:

```
jupyter labextension install @jupyter-widgets/jupyterlab-manager
```

This command defaults to installing the latest version of the **ipywidgets** JupyterLab extension. Depending on the version of JupyterLab you have installed, you may need to install an older version.

If you install this extension while JupyterLab is running, you will need to refresh the page or restart JupyterLab before the changes take effect.

Note: A clean reinstall of the JupyterLab extension can be done by first running the `jupyter lab clean` command which will remove the staging and static directories from the lab directory. The location of the lab directory can be queried by executing the command `jupyter lab path` in your terminal.

Frequently Asked Questions

The issues in the Reference milestone on GitHub include many questions, discussions, and answers about ipywidgets.

Question: When I display a widget or interact, I just see some text, such as `IntSlider(value=0)` or `interactive(children=(IntSlider(value=0, description='x', max=1), Output()), _dom_classes=('widget-interact',))`. What is wrong?

Answer: A text representation of the widget is printed if the widget control is not available. It may mean the widget JavaScript is still loading. If the message persists in the Jupyter Notebook or JupyterLab, it likely means that the widgets JavaScript library is either not installed or not enabled. See the installation instructions above for setup instructions.

If you see this message in another frontend (for example, a static rendering on GitHub or NBViewer), it may mean that your frontend doesn't currently support widgets.

1.1.2 Using Interact

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

```
[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 returns its only argument `x`.

```
[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.

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

```
interactive(children=(IntSlider(value=10, description='x', max=30, min=-10),␣
↪Output()), _dom_classes=('widget-...
```

```
[3]: <function __main__.f(x)>
```

When you move the slider, the function is called, and its return value is printed.

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

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

```
interactive(children=(Checkbox(value=True, description='x'), Output()), _dom_classes=(
↪'widget-interact',))
```

```
[4]: <function __main__.f(x)>
```

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

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

```
interactive(children=(Text(value='Hi there!', description='x'), Output()), _dom_
↪classes=('widget-interact',))
```

```
[5]: <function __main__.f(x)>
```

`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.

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

```
interactive(children=(Checkbox(value=True, description='x'), FloatSlider(value=1.0,␣
↪description='y', max=3.0, ...
```

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.

```
[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`.

```
[8]: interact(h, p=5, q=fixed(20));
     interactive(children=(IntSlider(value=5, description='p', max=15, min=-5), Output()),␣
     ↪_dom_classes=('widget-in...
[8]: <function __main__.h(p, q)>
```

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:

```
[9]: interact(f, x=widgets.IntSlider(min=-10, max=30, step=1, value=10));
     interactive(children=(IntSlider(value=10, description='x', max=30, min=-10),␣
     ↪Output()), _dom_classes=('widget-...
[9]: <function __main__.f(x)>
```

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 list of tuples 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 text 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.

```
[10]: interact(f, x=(0,4));
```
```
      interactive(children=(IntSlider(value=2, description='x', max=4), Output()), _dom_
      →classes=('widget-interact',)...
```
```
[10]: <function __main__.f(x)>
```

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

```
[11]: interact(f, x=(0,8,2));
```
```
      interactive(children=(IntSlider(value=4, description='x', max=8, step=2), Output()), _
      →dom_classes=('widget-int...
```
```
[11]: <function __main__.f(x)>
```

A float-valued slider is produced if *any* of 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).

```
[12]: interact(f, x=(0.0,10.0));
```
```
      interactive(children=(FloatSlider(value=5.0, description='x', max=10.0), Output()), _
      →dom_classes=('widget-inte...
```
```
[12]: <function __main__.f(x)>
```

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

```
[13]: interact(f, x=(0.0,10.0,0.01));
```
```
      interactive(children=(FloatSlider(value=5.0, description='x', max=10.0, step=0.01),␣
      →Output()), _dom_classes=('...
```
```
[13]: <function __main__.f(x)>
```

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`.

```
[14]: @interact(x=(0.0,20.0,0.5))
      def h(x=5.5):
          return x
```
```
      interactive(children=(FloatSlider(value=5.5, description='x', max=20.0, step=0.5),␣
      →Output()), _dom_classes=('w...
```

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.

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

```
interactive(children=(Dropdown(description='x', options=('apples', 'oranges'), value=
→'apples'), Output()), _do...
```
[15]: `<function __main__.f(x)>`

If you want a dropdown menu that passes non-string values to the Python function, you can pass a list of (`'label'`, `value`) pairs. The first items are the names in the dropdown menu UI and the second items are values that are the arguments passed to the underlying Python function.

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

```
interactive(children=(Dropdown(description='x', options=(('one', 10), ('two', 20)),
→value=10), Output()), _dom...
```
[16]: `<function __main__.f(x)>`

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.

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 is a function that displays the sum of its two arguments and returns the sum. The `display` line may be omitted if you don't want to show the result of the function.

[17]:
```python
from IPython.display import display
def f(a, b):
    display(a + b)
    return a+b
```

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

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

The widget is an `interactive`, a subclass of `VBox`, which is a container for other widgets.

[19]: `type(w)`

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

The children of the `interactive` are two integer-valued sliders and an output widget, produced by the widget abbreviations above.

[20]: `w.children`

```
[20]: (IntSlider(value=10, description='a', max=30, min=-10),
 IntSlider(value=20, description='b', max=60, min=-20),
 Output())
```

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

[21]: `display(w)`

```
interactive(children=(IntSlider(value=10, description='a', max=30, min=-10),
→IntSlider(value=20, description='...
```

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.

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

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

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

Here is the current return value of the function.

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

```
[23]: 30
```

Disabling continuous updates

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

```
[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
```

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

```
1000000 [0, 1, 2, 3, 11, 22, 101, 111, 121, 202, 212, 1001, 1111, 2002, 10001, 10101,␣
→10201, 11011, 11111, 11211, 20002, 20102, 100001, 101101, 110011, 111111, 200002]
CPU times: user 857 ms, sys: 730 µs, total: 858 ms
Wall time: 878 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:

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

```
interactive(children=(FloatSlider(value=100000.0, description='i', max=10000000.0,␣
→min=100000.0, step=100000.0...
```

```
[26]: <function __main__.slow_function(i)>
```

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.

```
[27]: interact_manual(slow_function,i=FloatSlider(min=1e5, max=1e7, step=1e5));

      interactive(children=(FloatSlider(value=100000.0, description='i', max=10000000.0,␣
      ↪min=100000.0, step=100000.0...

[27]: <function __main__.slow_function(i)>
```

You can do the same thing with `interactive` by using a `dict` as the second argument, as shown below.

```
[28]: slow = interactive(slow_function, {'manual': True}, i=widgets.FloatSlider(min=1e4,␣
      ↪max=1e6, step=1e4))
      slow

      interactive(children=(FloatSlider(value=10000.0, description='i', max=1000000.0,␣
      ↪min=10000.0, step=10000.0), B...
```

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.

```
[29]: interact(slow_function,i=FloatSlider(min=1e5, max=1e7, step=1e5, continuous_
      ↪update=False));

      interactive(children=(FloatSlider(value=100000.0, continuous_update=False,␣
      ↪description='i', max=10000000.0, mi...

[29]: <function __main__.slow_function(i)>
```

More control over the user interface: `interactive_output`

`interactive_output` provides additional flexibility: you can control how the UI elements are laid out.

Unlike `interact`, `interactive`, and `interact_manual`, `interactive_output` does not generate a user interface for the widgets. This is powerful, because it means you can create a widget, put it in a box, and then pass the widget to `interactive_output`, and have control over the widget and its layout.

```
[30]: a = widgets.IntSlider()
      b = widgets.IntSlider()
      c = widgets.IntSlider()
      ui = widgets.HBox([a, b, c])
      def f(a, b, c):
          print((a, b, c))

      out = widgets.interactive_output(f, {'a': a, 'b': b, 'c': c})

      display(ui, out)

      HBox(children=(IntSlider(value=0), IntSlider(value=0), IntSlider(value=0)))

      Output()
```

Arguments that are dependent on each other

Arguments that are dependent on 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*.

```
[31]: 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);
```

```
interactive(children=(FloatSlider(value=0.0, description='x', max=10.0, step=0.05),␣
↪FloatSlider(value=5.0, des...
```

```
[31]: <function __main__.printer(x, y)>
```

Flickering and jumping output

On occasion, you may notice interact output flickering and jumping, causing the notebook scroll position to change as the output is updated. The interactive control has a layout, so we can set its height to an appropriate value (currently chosen manually) so that it will not change size as it is updated.

```
[32]: %matplotlib inline
      from ipywidgets import interactive
      import matplotlib.pyplot as plt
      import numpy as np

      def f(m, b):
          plt.figure(2)
          x = np.linspace(-10, 10, num=1000)
          plt.plot(x, m * x + b)
          plt.ylim(-5, 5)
          plt.show()

      interactive_plot = interactive(f, m=(-2.0, 2.0), b=(-3, 3, 0.5))
      output = interactive_plot.children[-1]
      output.layout.height = '350px'
      interactive_plot
```

```
interactive(children=(FloatSlider(value=0.0, description='m', max=2.0, min=-2.0),␣
↪FloatSlider(value=0.0, descr...
```

```
[ ]:
```

1.1.3 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`.

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

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 output area below the code cell. Clearing cell output will also remove the widget.

```
[2]: widgets.IntSlider()
```

```
IntSlider(value=0)
```

display()

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

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

```
IntSlider(value=0)
```

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.

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

```
IntSlider(value=0)
```

Why does displaying the same widget twice work?

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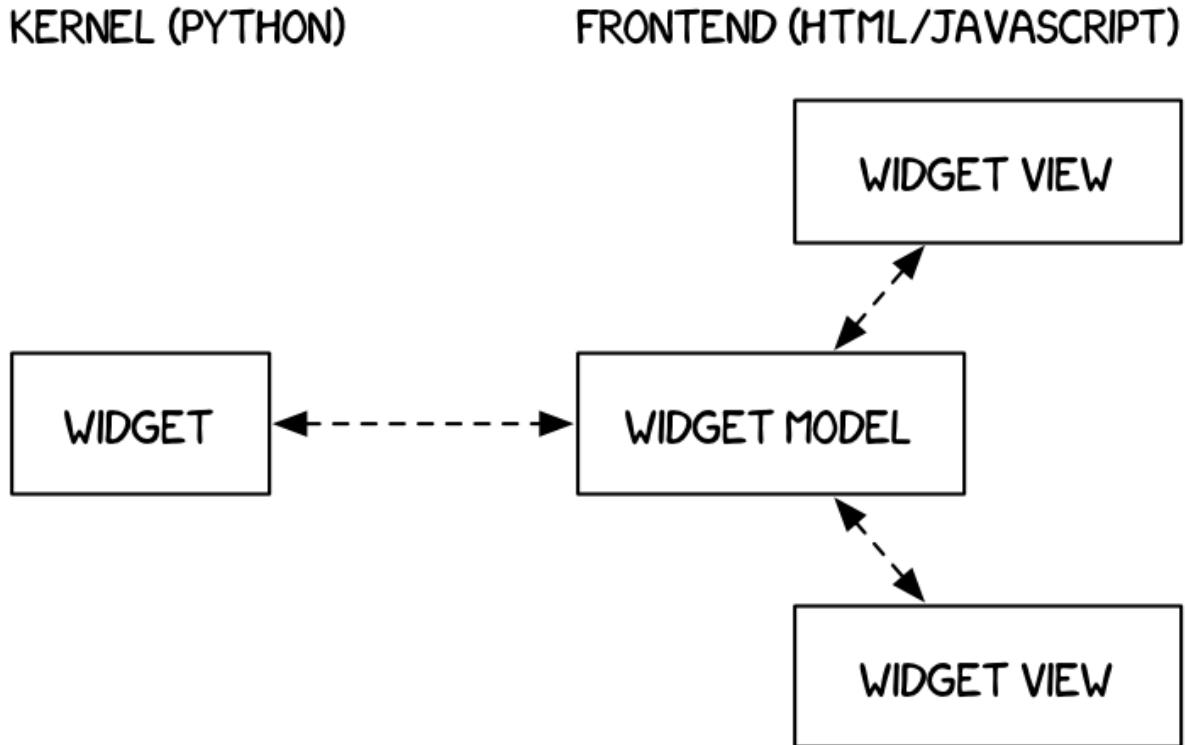
Closing widgets

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

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

```
IntSlider(value=0)
```

```
[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.

```
[7]: w = widgets.IntSlider()
     display(w)
```

```
IntSlider(value=0)
```

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

```
[8]: 0
```

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

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

Keys

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

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

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

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).

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

```
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 `link` or `jslink` function to link two properties together (the difference between these is discussed in *Widget Events*). Below, the values of two widgets are linked together.

```
[12]: a = widgets.FloatText()
      b = widgets.FloatSlider()
      display(a,b)

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

```
FloatText(value=0.0)
```

```
FloatSlider(value=0.0)
```

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.

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

1.1.4 Widget List

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

Numeric widgets

There are many widgets distributed with ipywidgets 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

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

```
IntSlider(value=7, continuous_update=False, description='Test:', max=10)
```

FloatSlider

```
[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',
     )
```

```
FloatSlider(value=7.5, continuous_update=False, description='Test:', max=10.0,␣
→readout_format='.1f')
```

Sliders can also be **displayed vertically**.

```
[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',
     )
```

```
FloatSlider(value=7.5, continuous_update=False, description='Test:', max=10.0,␣
→orientation='vertical', readout...
```

FloatLogSlider

The `FloatLogSlider` has a log scale, which makes it easy to have a slider that covers a wide range of positive magnitudes. The `min` and `max` refer to the minimum and maximum exponents of the `base`, and the `value` refers to the actual value of the slider.

```
[5]: widgets.FloatLogSlider(
         value=10,
         base=10,
         min=-10, # max exponent of base
         max=10, # min exponent of base
         step=0.2, # exponent step
         description='Log Slider'
     )
```

```
FloatLogSlider(value=10.0, description='Log Slider', max=10.0, min=-10.0, step=0.2)
```

IntRangeSlider

```
[6]: widgets.IntRangeSlider(
         value=[5, 7],
         min=0,
         max=10,
         step=1,
         description='Test:',
         disabled=False,
         continuous_update=False,
         orientation='horizontal',
         readout=True,
         readout_format='d',
     )
```

```
IntRangeSlider(value=(5, 7), continuous_update=False, description='Test:', max=10)
```

FloatRangeSlider

```
[7]: 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='.1f',
     )
```

```
FloatRangeSlider(value=(5.0, 7.5), continuous_update=False, description='Test:',
→max=10.0, readout_format='.1f...
```

IntProgress

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

```
IntProgress(value=7, description='Loading:', max=10)
```

FloatProgress

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

```
FloatProgress(value=7.5, bar_style='info', description='Loading:', max=10.0)
```

The numerical text boxes that impose some limit on the data (range, integer-only) impose that restriction when the user presses enter.

BoundedIntText

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

```
BoundedIntText(value=7, description='Text:', max=10)
```

BoundedFloatText

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

```
BoundedFloatText(value=7.5, description='Text:', max=10.0, step=0.1)
```

IntText

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

```
IntText(value=7, description='Any:')
```

FloatText

```
[13]: widgets.FloatText(
          value=7.5,
          description='Any:',
          disabled=False
      )
```

```
FloatText(value=7.5, description='Any:')
```

Boolean widgets

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

ToggleButton

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

```
ToggleButton(value=False, description='Click me', icon='check', tooltip='Description')
```

Checkbox

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

```
Checkbox(value=False, description='Check me')
```

Valid

The valid widget provides a read-only indicator.

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

```
Valid(value=False, description='Valid!')
```

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 labels are derived by calling `str`).

Dropdown

```
[17]: widgets.Dropdown(
          options=['1', '2', '3'],
          value='2',
          description='Number:',
          disabled=False,
      )
```

```
Dropdown(description='Number:', index=1, options=('1', '2', '3'), value='2')
```

The following is also valid, displaying the words `'One'`, `'Two'`, `'Three'` as the dropdown choices but returning the values `1`, `2`, `3`.

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

```
Dropdown(description='Number:', index=1, options=(('One', 1), ('Two', 2), ('Three',
→3)), value=2)
```

RadioButtons

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

```
RadioButtons(description='Pizza topping:', options=('pepperoni', 'pineapple',
→'anchovies'), value='pepperoni')
```

Select

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

```
Select(description='OS:', index=2, options=('Linux', 'Windows', 'OSX'), value='OSX')
```

SelectionSlider

```
[21]: 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
      )
```

```
SelectionSlider(continuous_update=False, description='I like my eggs ...', index=1,␣
↪options=('scrambled', 'sun...
```

SelectionRangeSlider

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

```
[22]: 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)',
          disabled=False
      )
```

```
SelectionRangeSlider(description='Months (2015)', index=(0, 11), options=(('Jan',␣
↪datetime.date(2015, 1, 1)), ...
```

ToggleButtons

```
[23]: widgets.ToggleButtons(
          options=['Slow', 'Regular', 'Fast'],
          description='Speed:',
          disabled=False,
          button_style='', # 'success', 'info', 'warning', 'danger' or ''
          tooltips=['Description of slow', 'Description of regular', 'Description of fast'],
      #     icons=['check'] * 3
      )
```

```
ToggleButtons(description='Speed:', options=('Slow', 'Regular', 'Fast'), tooltips=(
↪'Description of slow', 'Des...
```

SelectMultiple

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

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

```
SelectMultiple(description='Fruits', index=(1,), options=('Apples', 'Oranges', 'Pears
→'), value=('Oranges',))
```

String widgets

There are several widgets that can be used to display a string value. The `Text`, `Textarea`, and `Combobox` widgets accept input. The `HTML` and `HTMLMath` widgets display a string as HTML (`HTMLMath` also renders math). The `Label` widget can be used to construct a custom control label.

Text

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

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

Textarea

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

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

Combobox

```
[27]: widgets.Combobox(
          # value='John',
          placeholder='Choose Someone',
          options=['Paul', 'John', 'George', 'Ringo'],
          description='Combobox:',
          ensure_option=True,
          disabled=False
      )
```

```
Combobox(value='', description='Combobox:', ensure_option=True, options=('Paul', 'John
→', 'George', 'Ringo'), p...
```

Label

The `Label` widget is useful if you need to build a custom description next to a control using similar styling to the built-in control descriptions.

```
[28]: widgets.HBox([widgets.Label(value="The $m$ in $E=mc^2$:"), widgets.FloatSlider()])
```

```
HBox(children=(Label(value='The $m$ in $E=mc^2$:'), FloatSlider(value=0.0)))
```

HTML

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

```
HTML(value='Hello <b>World</b>', description='Some HTML', placeholder='Some HTML')
```

HTML Math

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

```
HTMLMath(value='Some math and <i>HTML</i>: \\(x^2\\) and $$\\frac{x+1}{x-1}$$',␣
→description='Some HTML', place...
```

Image

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

```
Image(value=b'\x89PNG\r\n\x1a\n\x00\x00\x00\rIHDR\x00\x00\x012\x00\x00\x01\xb1\x08\
→x06\x00\x00\x00\xe1\xe2:\xb...
```

Button

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

```
Button(description='Click me', icon='check', style=ButtonStyle(), tooltip='Click me')
```

Output

The `Output` widget can capture and display stdout, stderr and rich output generated by IPython. For detailed documentation, see the output widget examples.

Play (Animation) widget

The `Play` widget is useful to perform animations by iterating on a sequence of integers with a certain speed. The value of the slider below is linked to the player.

```
[33]: 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])
```

```
HBox(children=(Play(value=50, description='Press play'), IntSlider(value=0)))
```

Date picker

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

```
[34]: widgets.DatePicker(
          description='Pick a Date',
          disabled=False
      )
```

```
DatePicker(value=None, description='Pick a Date')
```

Color picker

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

```
ColorPicker(value='blue', description='Pick a color')
```

File Upload

The `FileUpload` allows to upload any type of file(s) as bytes.

```
[36]: widgets.FileUpload(
          accept='',  # Accepted file extension e.g. '.txt', '.pdf', 'image/*', 'image/*,.
      →pdf'
          multiple=False  # True to accept multiple files upload else False
      )
```

```
FileUpload(value={}, description='Upload')
```

Controller

The `Controller` allows a game controller to be used as an input device.

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

```
Controller()
```

Container/Layout widgets

These widgets are used to hold other widgets, called children. Each has a `children` property that may be set either when the widget is created or later.

Box

```
[38]: items = [widgets.Label(str(i)) for i in range(4)]
      widgets.Box(items)
```

```
Box(children=(Label(value='0'), Label(value='1'), Label(value='2'), Label(value='3')))
```

HBox

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

```
HBox(children=(Label(value='0'), Label(value='1'), Label(value='2'), Label(value='3
→')))
```

VBox

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

```
HBox(children=(VBox(children=(Label(value='0'), Label(value='1'))),␣
→VBox(children=(Label(value='2'), Label(val...
```

GridBox

This box uses the HTML Grid specification to lay out its children in two dimensional grid. The example below lays out the 8 items inside in 3 columns and as many rows as needed to accommodate the items.

```
[41]: items = [widgets.Label(str(i)) for i in range(8)]
      widgets.GridBox(items, layout=widgets.Layout(grid_template_columns="repeat(3, 100px)
→"))
```

```
GridBox(children=(Label(value='0'), Label(value='1'), Label(value='2'), Label(value='3
→'), Label(value='4'), La...
```

Accordion

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

```
Accordion(children=(IntSlider(value=0), Text(value='')), _titles={'0': 'Slider', '1':
→'Text'})
```

Tabs

In this example the children are set after the tab is created. Titles for the tabs are set in the same way they are for
Accordion.

```
[43]: tab_contents = ['P0', 'P1', 'P2', 'P3', 'P4']
      children = [widgets.Text(description=name) for name in tab_contents]
      tab = widgets.Tab()
      tab.children = children
      for i in range(len(children)):
          tab.set_title(i, str(i))
      tab
```

```
Tab(children=(Text(value='', description='P0'), Text(value='', description='P1'),␣
→Text(value='', description='...
```

Accordion and Tab use `selected_index`, not value

Unlike the rest of the widgets discussed earlier, the container widgets Accordion and Tab update their
selected_index attribute when the user changes which accordion or tab is selected. That means that you can both
see what the user is doing *and* programmatically set what the user sees by setting the value of selected_index.

Setting selected_index = None closes all of the accordions or deselects all tabs.

In the cells below try displaying or setting the selected_index of the tab and/or accordion.

```
[44]: tab.selected_index = 3
```

```
[45]: accordion.selected_index = None
```

Nesting tabs and accordions

Tabs and accordions can be nested as deeply as you want. If you have a few minutes, try nesting a few accordions or
putting an accordion inside a tab or a tab inside an accordion.

The example below makes a couple of tabs with an accordion children in one of them

```
[46]: tab_nest = widgets.Tab()
      tab_nest.children = [accordion, accordion]
      tab_nest.set_title(0, 'An accordion')
      tab_nest.set_title(1, 'Copy of the accordion')
      tab_nest
```

```
Tab(children=(Accordion(children=(IntSlider(value=0), Text(value='')), selected_
→index=None, _titles={'0': 'Sli...
```

1.1.5 Output widgets: leveraging Jupyter's display system

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

The `Output` widget can capture and display stdout, stderr and rich output generated by IPython. You can also append output directly to an output widget, or clear it programmatically.

```
[2]: out = widgets.Output(layout={'border': '1px solid black'})
     out
```

```
Output(layout=Layout(border='1px solid black'))
```

After the widget is created, direct output to it using a context manager. You can print text to the output area:

```
[3]: with out:
         for i in range(10):
             print(i, 'Hello world!')
```

Rich output can also be directed to the output area. Anything which displays nicely in a Jupyter notebook will also display well in the `Output` widget.

```
[4]: from IPython.display import YouTubeVideo
     with out:
         display(YouTubeVideo('eWzY2nGfkXk'))
```

We can even display complex mimetypes, such as nested widgets, in an output widget.

```
[5]: with out:
         display(widgets.IntSlider())
```

We can also append outputs to the output widget directly with the convenience methods `append_stdout`, `append_stderr`, or `append_display_data`.

```
[6]: out = widgets.Output(layout={'border': '1px solid black'})
     out.append_stdout('Output appended with append_stdout')
     out.append_display_data(YouTubeVideo('eWzY2nGfkXk'))
     out
```

```
Output(layout=Layout(border='1px solid black'), outputs=({'output_type': 'stream',
→'name': 'stdout', 'text': '...
```

Note that `append_display_data` cannot currently be used to display widgets. The status of this bug is tracked in this issue.

We can clear the output by either using `IPython.display.clear_output` within the context manager, or we can call the widget's `clear_output` method directly.

```
[7]: out.clear_output()
```

`clear_output` supports the keyword argument `wait`. With this set to `True`, the widget contents are not cleared immediately. Instead, they are cleared the next time the widget receives something to display. This can be useful when replacing content in the output widget: it allows for smoother transitions by avoiding a jarring resize of the widget following the call to `clear_output`.

Finally, we can use an output widget to capture all the output produced by a function using the `capture` decorator.

```
[8]: @out.capture()
     def function_with_captured_output():
         print('This goes into the output widget')
         raise Exception('As does this')

     function_with_captured_output()
```

`out.capture` supports the keyword argument `clear_output`. Setting this to `True` will clear the output widget every time the function is invoked, so that you only see the output of the last invocation. With `clear_output` set to `True`, you can also pass a `wait=True` argument to only clear the output once new output is available. Of course, you can also manually clear the output any time as well.

```
[9]: out.clear_output()
```

Output widgets as the foundation for interact

The output widget forms the basis of how interact and related methods are implemented. It can also be used by itself to create rich layouts with widgets and code output. One simple way to customize how an interact UI looks is to use the `interactive_output` function to hook controls up to a function whose output is captured in the returned output widget. In the next example, we stack the controls vertically and then put the output of the function to the right.

```
[10]: a = widgets.IntSlider(description='a')
      b = widgets.IntSlider(description='b')
      c = widgets.IntSlider(description='c')
      def f(a, b, c):
          print('{}*{}*{}={}'.format(a, b, c, a*b*c))

      out = widgets.interactive_output(f, {'a': a, 'b': b, 'c': c})

      widgets.HBox([widgets.VBox([a, b, c]), out])
```

```
HBox(children=(VBox(children=(IntSlider(value=0, description='a'), IntSlider(value=0,␣
→description='b'), IntSli...
```

Debugging errors in callbacks with the output widget

On some platforms, like JupyterLab, output generated by widget callbacks (for instance, functions attached to the `.observe` method on widget traits, or to the `.on_click` method on button widgets) are not displayed anywhere. Even on other platforms, it is unclear what cell this output should appear in. This can make debugging errors in callback functions more challenging.

An effective tool for accessing the output of widget callbacks is to decorate the callback with an output widget's capture method. You can then display the widget in a new cell to see the callback output.

```
[11]: debug_view = widgets.Output(layout={'border': '1px solid black'})

      @debug_view.capture(clear_output=True)
      def bad_callback(event):
          print('This is about to explode')
          return 1.0 / 0.0

      button = widgets.Button(
          description='click me to raise an exception',
```

```
        layout={'width': '300px'}
)
button.on_click(bad_callback)
button
```

```
Button(description='click me to raise an exception', layout=Layout(width='300px'),
→style=ButtonStyle())
```

```
[12]: debug_view
```

```
Output(layout=Layout(border='1px solid black'))
```

Integrating output widgets with the logging module

While using the `.capture` decorator works well for understanding and debugging single callbacks, it does not scale to larger applications. Typically, in larger applications, one might use the logging module to print information on the status of the program. However, in the case of widget applications, it is unclear where the logging output should go.

A useful pattern is to create a custom handler that redirects logs to an output widget. The output widget can then be displayed in a new cell to monitor the application while it runs.

```
[13]: import ipywidgets as widgets
      import logging

      class OutputWidgetHandler(logging.Handler):
          """ Custom logging handler sending logs to an output widget """

          def __init__(self, *args, **kwargs):
              super(OutputWidgetHandler, self).__init__(*args, **kwargs)
              layout = {
                  'width': '100%',
                  'height': '160px',
                  'border': '1px solid black'
              }
              self.out = widgets.Output(layout=layout)

          def emit(self, record):
              """ Overload of logging.Handler method """
              formatted_record = self.format(record)
              new_output = {
                  'name': 'stdout',
                  'output_type': 'stream',
                  'text': formatted_record+'\n'
              }
              self.out.outputs = (new_output, ) + self.out.outputs

          def show_logs(self):
              """ Show the logs """
              display(self.out)

          def clear_logs(self):
              """ Clear the current logs """
              self.out.clear_output()


      logger = logging.getLogger(__name__)
```

```
handler = OutputWidgetHandler()
handler.setFormatter(logging.Formatter('%(asctime)s  - [%(levelname)s] %(message)s'))
logger.addHandler(handler)
logger.setLevel(logging.INFO)
```

```
[14]: handler.show_logs()
```

```
Output(layout=Layout(border='1px solid black', height='160px', width='100%'))
```

```
[15]: handler.clear_logs()
      logger.info('Starting program')

      try:
          logger.info('About to try something dangerous...')
          1.0/0.0
      except Exception as e:
          logger.exception('An error occurred!')
```

Interacting with output widgets from background threads

Jupyter's `display` mechanism can be counter-intuitive when displaying output produced by background threads. A background thread's output is printed to whatever cell the main thread is currently writing to. To see this directly, create a thread that repeatedly prints to standard out:

```python
import threading
import time

def run():
    for i in itertools.count(0):
        time.sleep(1)
        print('output from background {}'.format(i))

t = threading.Thread(target=run)
t.start()
```

This always prints in the currently active cell, not the cell that started the background thread.

This can lead to surprising behaviour in output widgets. During the time in which output is captured by the output widget, *any* output generated in the notebook, regardless of thread, will go into the output widget.

The best way to avoid surprises is to *never* use an output widget's context manager in a context where multiple threads generate output. Instead, we can pass an output widget to the function executing in a thread, and use `append_display_data()`, `append_stdout()`, or `append_stderr()` methods to append displayable output to the output widget.

```python
[16]: import threading
      from IPython.display import display, HTML
      import ipywidgets as widgets
      import time

      def thread_func(something, out):
          for i in range(1, 5):
              time.sleep(0.3)
              out.append_stdout('{} {} {}\n'.format(i, '**'*i, something))
          out.append_display_data(HTML("<em>All done!</em>"))
```

```
display('Display in main thread')
out = widgets.Output()
# Now the key: the container is displayed (while empty) in the main thread
display(out)

thread = threading.Thread(
    target=thread_func,
    args=("some text", out))
thread.start()
```

```
'Display in main thread'
```

```
Output()
```

1.1.6 Widget Events

Special events

```
[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.

```
[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. To capture `prints` (or any other kind of output) and ensure it is displayed, be sure to send it to an `Output` widget (or put the information you want to display into an `HTML` widget).

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

     display(button, output)

     def on_button_clicked(b):
```

```
    with output:
        print("Button clicked.")
```

```
button.on_click(on_button_clicked)
```

```
Button(description='Click Me!', style=ButtonStyle())
```

```
Output()
```

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.

```
[4]: 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.

```
[5]: int_range = widgets.IntSlider()
     output2 = widgets.Output()

     display(int_range, output2)

     def on_value_change(change):
```

```
    with output2:
        print(change['new'])

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

```
IntSlider(value=0)
```

```
Output()
```

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 (these two functions are re-exported by the `ipywidgets` module for convenience). This only works if we are interacting with a live kernel.

```
[6]: 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 = widgets.link((sliders1, 'value'), (slider2, 'value'))
     display(caption, sliders1, slider2)
```

```
Label(value='The values of slider1 and slider2 are synchronized')
```

```
IntSlider(value=0, description='Slider 1')
```

```
IntSlider(value=0, description='Slider 2')
```

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

```
Label(value='Changes in source values are reflected in target1')
```

```
IntSlider(value=0, description='Source')
```

```
IntSlider(value=0, description='Target 1')
```

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

```
[8]: 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 observe 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.

```
[9]: 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)
```

```
Label(value='The values of range1 and range2 are synchronized')
```

```
IntSlider(value=1, description='Slider', max=5, min=-5)
```

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.

```
[10]: 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)
```

```
Label(value='The values of range1 and range2 are synchronized')
```

```
IntSlider(value=0, description='Range 1')
```

```
IntSlider(value=0, description='Range 2')
```

```
[11]: caption = widgets.Label(value='Changes in source_range values are reflected in target_
      →range1')
      source_range, target_range1 = widgets.IntSlider(description='Source range'),\
```

(continues on next page)

```
                                    widgets.IntSlider(description='Target range 1')
dl = widgets.jsdlink((source_range, 'value'), (target_range1, 'value'))
display(caption, source_range, target_range1)
```

```
Label(value='Changes in source_range values are reflected in target_range1')
```

```
IntSlider(value=0, description='Source range')
```

```
IntSlider(value=0, description='Target range 1')
```

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

```
[12]:  # l.unlink()
       # dl.unlink()
```

The difference between linking in the kernel and linking in the client

Linking in the kernel means linking via python. If two sliders are linked in the kernel, when one slider is changed the browser sends a message to the kernel (python in this case) updating the changed slider, the link widget in the kernel then propagates the change to the other slider object in the kernel, and then the other slider's kernel object sends a message to the browser to update the other slider's views in the browser. If the kernel is not running (as in a static web page), then the controls will not be linked.

Linking using jslink (i.e., on the browser side) means contructing the link in Javascript. When one slider is changed, Javascript running in the browser changes the value of the other slider in the browser, without needing to communicate with the kernel at all. If the sliders are attached to kernel objects, each slider will update their kernel-side objects independently.

To see the difference between the two, go to the static version of this page in the ipywidgets documentation and try out the sliders near the bottom. The ones linked in the kernel with `link` and `dlink` are no longer linked, but the ones linked in the browser with `jslink` and `jsdlink` are still linked.

Continuous updates

Some widgets offer a choice with their `continuous_update` attribute between continually updating values or only updating values when a user submits the value (for example, by pressing Enter or navigating away from the control). In the next example, we see the "Delayed" controls only transmit their value after the user finishes dragging the slider or submitting the textbox. The "Continuous" controls continually transmit their values as they are changed. Try typing a two-digit number into each of the text boxes, or dragging each of the sliders, to see the difference.

```
[13]:  a = widgets.IntSlider(description="Delayed", continuous_update=False)
       b = widgets.IntText(description="Delayed", continuous_update=False)
       c = widgets.IntSlider(description="Continuous", continuous_update=True)
       d = widgets.IntText(description="Continuous", continuous_update=True)

       widgets.link((a, 'value'), (b, 'value'))
       widgets.link((a, 'value'), (c, 'value'))
       widgets.link((a, 'value'), (d, 'value'))
       widgets.VBox([a,b,c,d])
```

```
VBox(children=(IntSlider(value=0, continuous_update=False, description='Delayed'),␣
→IntText(value=0, descriptio...
```

Sliders, `Text`, and `Textarea` controls default to `continuous_update=True`. `IntText` and other text boxes for entering integer or float numbers default to `continuous_update=False` (since often you'll want to type an entire number before submitting the value by pressing enter or navigating out of the box).

1.1.7 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.

You can jump directly to these sections:

- *The ``layout` attribute <#The-layout-attribute>`__*
- *The Flexbox layout*
- *Predefined styles*
- *The ``style` attribute <#The-style-attribute>`__*
- *The Grid layout*
- `Image layout and sizing <#Image-layout-and-sizing>`__

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` (deprecated)
- `overflow_y` (deprecated)

Box model

- `border`
- `margin`
- `padding`

Positioning

- `top`
- `left`
- `bottom`
- `right`

Flexbox

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

Grid layout

- `grid_auto_columns`
- `grid_auto_flow`
- `grid_auto_rows`
- `grid_gap`
- `grid_template`
- `grid_row`
- `grid_column`

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 `'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:

```
[1]: from ipywidgets import Button, Layout

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

```
Button(description='(50% width, 80px height) button', layout=Layout(height='80px',
↪width='50%'), style=ButtonS...
```

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

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

```
Button(description='Another button with the same layout', layout=Layout(height='80px',
↪ width='50%'), style=But...
```

Description

You may have noticed that long descriptions are truncated. This is because the description length is, by default, fixed.

```
[3]: from ipywidgets import IntSlider

IntSlider(description='A too long description')
```

```
IntSlider(value=0, description='A too long description')
```

You can change the length of the description to fit the description text. However, this will make the widget itself shorter. You can change both by adjusting the description width and the widget width using the widget's style.

```
[4]: style = {'description_width': 'initial'}
IntSlider(description='A too long description', style=style)
```

```
IntSlider(value=0, description='A too long description',
↪style=SliderStyle(description_width='initial'))
```

If you need more flexibility to lay out widgets and descriptions, you can use Label widgets directly.

```
[5]: from ipywidgets import HBox, Label

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

```
HBox(children=(Label(value='A too long description'), IntSlider(value=0)))
```

Natural sizes, and arrangements using HBox and VBox

Most of the core-widgets have default heights and widths that tile well together. This allows simple layouts based on the `HBox` and `VBox` helper functions to align naturally:

```
[6]: from ipywidgets import Button, HBox, VBox

     words = ['correct', 'horse', 'battery', 'staple']
     items = [Button(description=w) for w in words]
     left_box = VBox([items[0], items[1]])
     right_box = VBox([items[2], items[3]])
     HBox([left_box, right_box])
```

```
HBox(children=(VBox(children=(Button(description='correct', style=ButtonStyle()),␣
→Button(description='horse', ...
```

Latex

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

```
[7]: from ipywidgets import IntSlider, Label
```

```
[8]: IntSlider(description=r'\(\int_0^t f\)')
```

```
IntSlider(value=0, description='\\(\\int_0^t f\\)')
```

```
[9]: Label(value=r'\(e=mc^2\)')
```

```
Label(value='\\(e=mc^2\\)')
```

Number formatting

Sliders have a readout field which can be formatted using Python's 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

The `HBox` and `VBox` classes above are special cases of the `Box` widget.

The `Box` widget enables the entire CSS flexbox spec as well as the Grid layout 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 flexbox tutorial on the flexbox layout follows the lines of the article A Complete Guide to Flexbox by Chris Coyier, and uses text and various images from the article with permission.

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 (known as "flex items").

If regular layout is based on both block and inline flow directions, the flex layout is based on "flex-flow directions". Please have a look at this figure from the specification, explaining the main idea behind the flex layout.
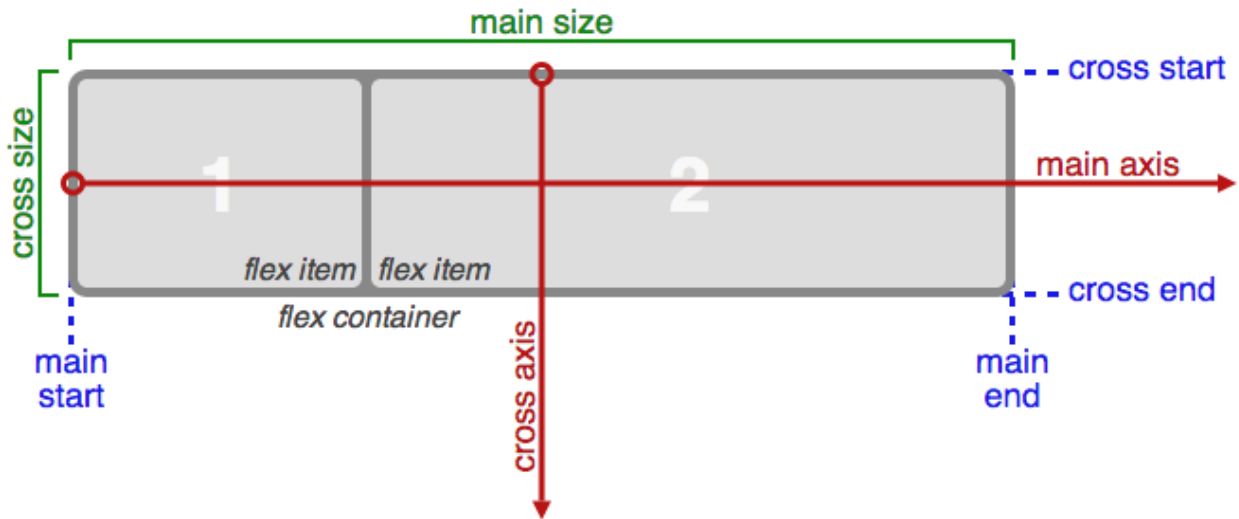
Basically, items will be laid out following either the `main axis` (from `main-start` to `main-end`) or the `cross axis` (from `cross-start` to `cross-end`).

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

- `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

display

`display` can be `flex` or `inline-flex`. This defines a flex container (block or inline).

flex-flow

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

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

- `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.

justify-content

`justify-content` can be one of `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

`align-items` can be one of `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

`align-content` can be one of `flex-start`, `flex-end`, `center`, `baseline`, `stretch`. This aligns a flex container's lines within when there is extra space in the cross-axis, similar to how justify-content aligns individual items within the main-axis.

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

Properties of the items

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

`flex` is shorthand for three properties, `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

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

The VBox and HBox helpers

The `VBox` and `HBox` helper classes provide simple defaults to arrange child widgets in vertical and horizontal boxes. They are roughly equivalent to:

```python
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
```

(continues on next page)

```python
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.

```
[10]: from ipywidgets import Layout, Button, Box

      items_layout = Layout( width='auto')     # override the default width of the button␣
      ↪to 'auto' to let the button grow

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

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

```
Box(children=(Button(button_style='danger', description='correct',␣
↪layout=Layout(width='auto'), style=ButtonSt...
```

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

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

      # Items flex proportionally to the weight and the left over space around the text
      items_auto = [
          Button(description='weight=1; auto', layout=Layout(flex='1 1 auto', width='auto'),
      ↪ button_style='danger'),
          Button(description='weight=3; auto', layout=Layout(flex='3 1 auto', width='auto'),
      ↪ button_style='danger'),
          Button(description='weight=1; auto', layout=Layout(flex='1 1 auto', width='auto'),
      ↪ button_style='danger'),
       ]

      # Items flex proportionally to the weight
      items_0 = [
          Button(description='weight=1; 0%', layout=Layout(flex='1 1 0%', width='auto'),␣
      ↪button_style='danger'),
          Button(description='weight=3; 0%', layout=Layout(flex='3 1 0%', width='auto'),␣
      ↪button_style='danger'),
          Button(description='weight=1; 0%', layout=Layout(flex='1 1 0%', width='auto'),␣
      ↪button_style='danger'),
```

```
    ]
box_layout = Layout(display='flex',
                    flex_flow='row',
                    align_items='stretch',
                    width='70%')
box_auto = Box(children=items_auto, layout=box_layout)
box_0 = Box(children=items_0, layout=box_layout)
VBox([box_auto, box_0])
```

```
VBox(children=(Box(children=(Button(button_style='danger', description='weight=1; auto
↪', layout=Layout(flex='1...
```

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..

```
[12]: from ipywidgets import Layout, Button, Box, FloatText, Textarea, Dropdown, Label,
      ↪IntSlider

      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=form_
      ↪item_layout),
          Box([Label(value='Egg style'),
              Dropdown(options=['Scrambled', 'Sunny side up', 'Over easy'])], layout=form_
      ↪item_layout),
          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
```

```
Box(children=(Box(children=(Label(value='Age of the captain'), IntSlider(value=40,
↪max=60, min=40)), layout=La...
```

A more advanced example: a carousel.

```
[13]: from ipywidgets import Layout, Button, VBox, Label

      item_layout = Layout(height='100px', min_width='40px')
      items = [Button(layout=item_layout, description=str(i), button_style='warning') for i
      ↪in range(40)]
      box_layout = Layout(overflow='scroll hidden',
                          border='3px solid black',
```

```
                        width='500px',
                        height='',
                        flex_flow='row',
                        display='flex')
carousel = Box(children=items, layout=box_layout)
VBox([Label('Scroll horizontally:'), carousel])
```

```
VBox(children=(Label(value='Scroll horizontally:'), Box(children=(Button(button_style=
→'warning', description='...
```

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 ".

```
[14]: from ipywidgets import Button

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

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

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.

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

```
      Button(description='Custom color', style=ButtonStyle(button_color='lightgreen'))
```

You can get a list of the style attributes for a widget with the `keys` property.

```
[16]: b1.style.keys
```

```
[16]: ['_model_module',
       '_model_module_version',
       '_model_name',
```

```
'_view_count',
'_view_module',
'_view_module_version',
'_view_name',
'button_color',
'font_weight']
```

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

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

```
Button(style=ButtonStyle(button_color='lightgreen'))
```

Widget styling attributes are specific to each widget type.

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

```
IntSlider(value=0, description='Blue handle', style=SliderStyle(handle_color=
→'lightblue'))
```

The Grid layout

The `GridBox` class is a special case of the `Box` widget.

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 grid layout 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.

The following flexbox tutorial on the flexbox layout follows the lines of the article A Complete Guide to Grid by Chris House, and uses text and various images from the article with permission.

Basics and browser support

To get started you have to define a container element as a grid with display: grid, set the column and row sizes with grid-template-rows, grid-template-columns, and grid_template_areas, and then place its child elements into the grid with grid-column and grid-row. Similarly to flexbox, the source order of the grid items doesn't matter. Your CSS can place them in any order, which makes it super easy to rearrange your grid with media queries. Imagine defining the layout of your entire page, and then completely rearranging it to accommodate a different screen width all with only a couple lines of CSS. Grid is one of the most powerful CSS modules ever introduced.

As of March 2017, most browsers shipped native, unprefixed support for CSS Grid: Chrome (including on Android), Firefox, Safari (including on iOS), and Opera. Internet Explorer 10 and 11 on the other hand support it, but it's an old implementation with an outdated syntax. The time to build with grid is now!

Important terminology

Before diving into the concepts of Grid it's important to understand the terminology. Since the terms involved here are all kinda conceptually similar, it's easy to confuse them with one another if you don't first memorize their meanings defined by the Grid specification. But don't worry, there aren't many of them.

Grid Container

The element on which `display: grid` is applied. It's the direct parent of all the grid items. In this example container is the grid container.

```
<div class="container">
  <div class="item item-1"></div>
  <div class="item item-2"></div>
  <div class="item item-3"></div>
</div>
```

Grid Item

The children (e.g. direct descendants) of the grid container. Here the item elements are grid items, but sub-item isn't.

```
<div class="container">
  <div class="item"></div>
  <div class="item">
    <p class="sub-item"></p>
  </div>
  <div class="item"></div>
</div>
```

Grid Line

The dividing lines that make up the structure of the grid. They can be either vertical ("column grid lines") or horizontal ("row grid lines") and reside on either side of a row or column. Here the yellow line is an example of a column grid line.
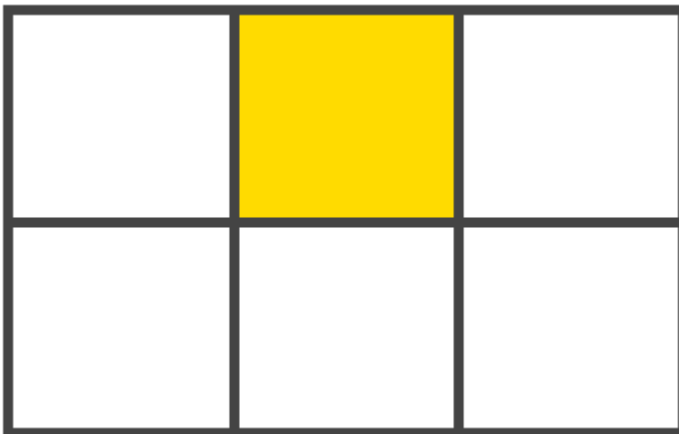

Grid Track

The space between two adjacent grid lines. You can think of them like the columns or rows of the grid. Here's the grid track between the second and third row grid lines.

Grid Cell

The space between two adjacent row and two adjacent column grid lines. It's a single "unit" of the grid. Here's the grid cell between row grid lines 1 and 2, and column grid lines 2 and 3.

Grid Area

The total space surrounded by four grid lines. A grid area may be comprised of any number of grid cells. Here's the grid area between row grid lines 1 and 3, and column grid lines 1 and 3.

Properties of the parent

grid-template-rows, grid-template-colums

Defines the columns and rows of the grid with a space-separated list of values. The values represent the track size, and the space between them represents the grid line.

Values:

- `<track-size>` - can be a length, a percentage, or a fraction of the free space in the grid (using the `fr` unit)
- `<line-name>` - an arbitrary name of your choosing

grid-template-areas

Defines a grid template by referencing the names of the grid areas which are specified with the grid-area property. Repeating the name of a grid area causes the content to span those cells. A period signifies an empty cell. The syntax itself provides a visualization of the structure of the grid.

Values:

- `<grid-area-name>` - the name of a grid area specified with `grid-area`
- `.` - a period signifies an empty grid cell
- `none` - no grid areas are defined

grid-gap

A shorthand for `grid-row-gap` and `grid-column-gap`

Values:

- `<grid-row-gap>`, `<grid-column-gap>` - length values

where `grid-row-gap` and `grid-column-gap` specify the sizes of the grid lines. You can think of it like setting the width of the gutters between the columns / rows.

- `<line-size>` - a length value

Note: The ``grid-`` prefix will be removed and ``grid-gap`` renamed to ``gap``. The unprefixed property is already supported in Chrome 68+, Safari 11.2 Release 50+ and Opera 54+.

align-items

Aligns grid items along the block (column) axis (as opposed to justify-items which aligns along the inline (row) axis). This value applies to all grid items inside the container.

Values:

- `start` - aligns items to be flush with the start edge of their cell
- `end` - aligns items to be flush with the end edge of their cell
- `center` - aligns items in the center of their cell
- `stretch` - fills the whole height of the cell (this is the default)

justify-items

Aligns grid items along the inline (row) axis (as opposed to `align-items` which aligns along the block (column) axis). This value applies to all grid items inside the container.

Values:

- `start` - aligns items to be flush with the start edge of their cell
- `end` - aligns items to be flush with the end edge of their cell

- `center` - aligns items in the center of their cell
- `stretch` - fills the whole width of the cell (this is the default)

align-content

Sometimes the total size of your grid might be less than the size of its grid container. This could happen if all of your grid items are sized with non-flexible units like `px`. In this case you can set the alignment of the grid within the grid container. This property aligns the grid along the block (column) axis (as opposed to justify-content which aligns the grid along the inline (row) axis).

Values:

- `start` - aligns the grid to be flush with the start edge of the grid container
- `end` - aligns the grid to be flush with the end edge of the grid container
- `center` - aligns the grid in the center of the grid container
- `stretch` - resizes the grid items to allow the grid to fill the full height of the grid container
- `space-around` - places an even amount of space between each grid item, with half-sized spaces on the far ends
- `space-between` - places an even amount of space between each grid item, with no space at the far ends
- `space-evenly` - places an even amount of space between each grid item, including the far ends

justify-content

Sometimes the total size of your grid might be less than the size of its grid container. This could happen if all of your grid items are sized with non-flexible units like `px`. In this case you can set the alignment of the grid within the grid container. This property aligns the grid along the inline (row) axis (as opposed to align-content which aligns the grid along the block (column) axis).

Values:

- `start` - aligns the grid to be flush with the start edge of the grid container
- `end` - aligns the grid to be flush with the end edge of the grid container
- `center` - aligns the grid in the center of the grid container
- `stretch` - resizes the grid items to allow the grid to fill the full width of the grid container
- `space-around` - places an even amount of space between each grid item, with half-sized spaces on the far ends
- `space-between` - places an even amount of space between each grid item, with no space at the far ends
- `space-evenly` - places an even amount of space between each grid item, including the far ends

grid-auto-columns, grid-auto-rows

Specifies the size of any auto-generated grid tracks (aka implicit grid tracks). Implicit tracks get created when there are more grid items than cells in the grid or when a grid item is placed outside of the explicit grid. (see The Difference Between Explicit and Implicit Grids)

Values:

- `<track-size>` - can be a length, a percentage, or a fraction of the free space in the grid (using the `fr` unit)

Properties of the items

Note: ``float``, ``display: inline-block``, ``display: table-cell``, ``vertical-align`` *and* ``column-??`` *properties have no effect on a grid item.*

grid-column, grid-row

Determines a grid item's location within the grid by referring to specific grid lines. `grid-column-start`/`grid-row-start` is the line where the item begins, and `grid-column-end`/`grid-row-end` is the line where the item ends.

Values:

- `<line>` - can be a number to refer to a numbered grid line, or a name to refer to a named grid line
- `span <number>` - the item will span across the provided number of grid tracks
- `span <name>` - the item will span across until it hits the next line with the provided name
- `auto` - indicates auto-placement, an automatic span, or a default span of one

```css
.item {
  grid-column: <number> | <name> | span <number> | span <name> | auto /
               <number> | <name> | span <number> | span <name> | auto
  grid-row: <number> | <name> | span <number> | span <name> | auto /
            <number> | <name> | span <number> | span <name> | auto
}
```

Examples:

```css
.item-a {
  grid-column: 2 / five;
  grid-row: row1-start / 3;
}
```
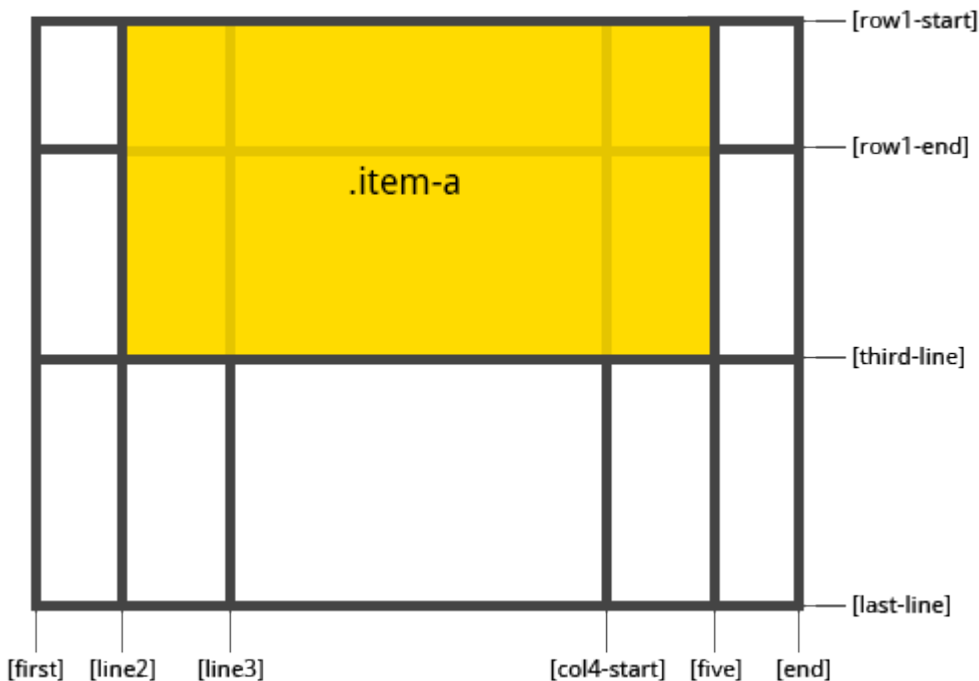
```
.item-b {
  grid-column: 1 / span col4-start;
  grid-row: 2 / span 2;
}
```


If no `grid-column` / `grid-row` is declared, the item will span 1 track by default.

Items can overlap each other. You can use `z-index` to control their stacking order.

grid-area

Gives an item a name so that it can be referenced by a template created with the `grid-template-areas` property. Alternatively, this property can be used as an even shorter shorthand for `grid-row-start` + `grid-column-start` + `grid-row-end` + `grid-column-end`.

Values:

- `<name>` - a name of your choosing

- `<row-start>` / `<column-start>` / `<row-end>` / `<column-end>` - can be numbers or named lines

```
.item {
  grid-area: <name> | <row-start> / <column-start> / <row-end> / <column-end>;
}
```
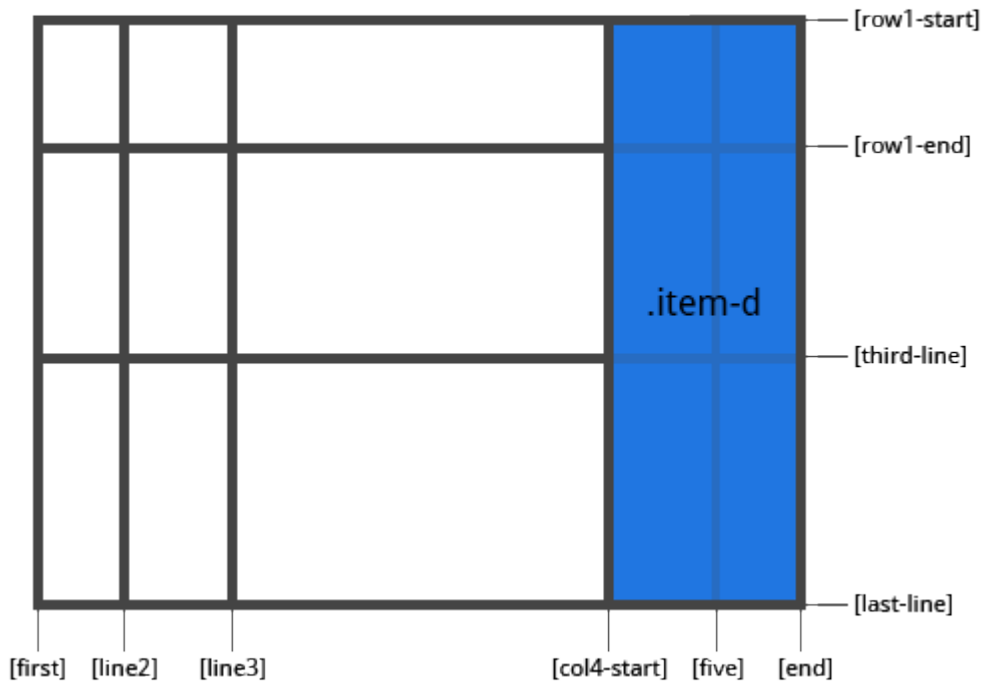
Examples:

As a way to assign a name to the item:

```
.item-d {
  grid-area: header
}
```

As the short-shorthand for `grid-row-start` + `grid-column-start` + `grid-row-end` + `grid-column-end`:

```
.item-d {
  grid-area: 1 / col4-start / last-line / 6
}
```


justify-self

Aligns a grid item inside a cell along the inline (row) axis (as opposed to `align-self` which aligns along the block (column) axis). This value applies to a grid item inside a single cell.
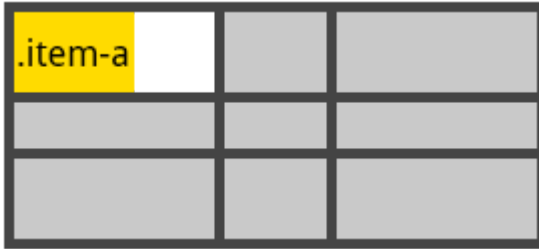
Values:

- `start` - aligns the grid item to be flush with the start edge of the cell
- `end` - aligns the grid item to be flush with the end edge of the cell
- `center` - aligns the grid item in the center of the cell
- `stretch` - fills the whole width of the cell (this is the default)

```
.item {
  justify-self: start | end | center | stretch;
}
```
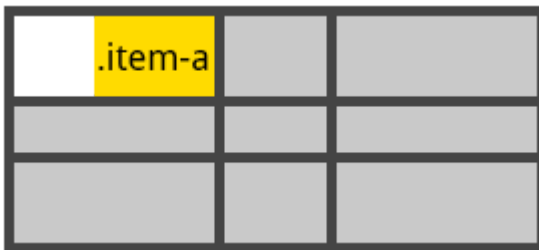
Examples:

```
.item-a {
  justify-self: start;
}
```
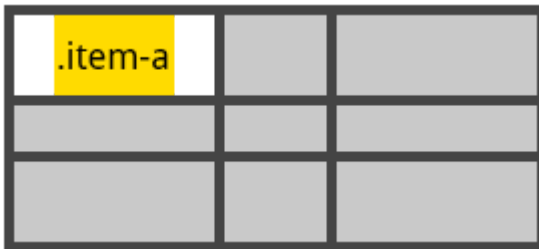
```
.item-a {
  justify-self: end;
}
```



```
.item-a {
  justify-self: center;
}
```



```
.item-a {
  justify-self: stretch;
}
```


To set alignment for *all* the items in a grid, this behavior can also be set on the grid container via the `justify-items` property.

```
[19]: from ipywidgets import Button, GridBox, Layout, ButtonStyle
```

Placing items by name:

```
[20]: header  = Button(description='Header',
                   layout=Layout(width='auto', grid_area='header'),
                   style=ButtonStyle(button_color='lightblue'))
      main    = Button(description='Main',
                   layout=Layout(width='auto', grid_area='main'),
                   style=ButtonStyle(button_color='moccasin'))
      sidebar = Button(description='Sidebar',
                   layout=Layout(width='auto', grid_area='sidebar'),
                   style=ButtonStyle(button_color='salmon'))
      footer  = Button(description='Footer',
                   layout=Layout(width='auto', grid_area='footer'),
                   style=ButtonStyle(button_color='olive'))

      GridBox(children=[header, main, sidebar, footer],
              layout=Layout(
                  width='50%',
                  grid_template_rows='auto auto auto',
                  grid_template_columns='25% 25% 25% 25%',
                  grid_template_areas='''
                  "header header header header"
                  "main main . sidebar "
                  "footer footer footer footer"
                  ''')
             )
```

```
GridBox(children=(Button(description='Header', layout=Layout(grid_area='header',␣
→width='auto'), style=ButtonSt...
```

Setting up row and column template and gap

```
[21]: GridBox(children=[Button(layout=Layout(width='auto', height='auto'),
                           style=ButtonStyle(button_color='darkseagreen')) for i in␣
      →range(9)
                       ],
              layout=Layout(
                  width='50%',
                  grid_template_columns='100px 50px 100px',
                  grid_template_rows='80px auto 80px',
                  grid_gap='5px 10px')
             )
```

```
GridBox(children=(Button(layout=Layout(height='auto', width='auto'),␣
→style=ButtonStyle(button_color='darkseagr...
```

`Image` layout and sizing

The layout and sizing of images is a little different than for other elements for a combination of historical reasons (the HTML tag `img` existed before CSS) and practical reasons (an image has an intrinsic size).

Sizing of images is particularly confusing because there are two plausible ways to specify the size of an image. The `Image` widget has attributes `width` and `height` that correspond to attributes of the same name on the HTML `img` tag. In addition, the `Image` widget, like every other widget, has a `layout`, which also has a `width` and `height`.

In addition, some CSS styling is applied to images that is not applied to other widgets: `max_width` is set to `100%` and `height` is set to `auto`.

You should not rely on `Image.width` or `Image.height` to determine the display width and height of the image widget. Any CSS styling, whether via `Image.layout` or some other source, will override `Image.width` and

`Image.height`.

When displaying an `Image` widget by itself, setting `Image.layout.width` to the desired width will display the `Image` widget with the same aspect ratio as the original image.

When placing an `Image` inside a `Box` (or `HBox` or `VBox`) the result depends on whether a width has been set on the `Box`. If it has, then the image will be stretched (or compressed) to fit within the box because `Image.layout.max_width` is `100%` so the image fills the container. This will usually not preserve the aspect ratio of image.

Controlling the display of an `Image` inside a container

Use `Image.layout.object_fit` to control how an image is scaled inside a container like a box. The possible values are:

- `'contain'`: Fit the image in its content box while preserving the aspect ratio. If any of the container is not covered by the image, the background of the container is displayed. The content box is the size of the container if the container is smaller than the image, or the size of the image if the container is larger.

- `'cover'`: Fill the content box completely while preserving the aspect ratio of the image, cropping the image if necessary.

- `'fill'`: Completely fill the content box, stretching/compressing the image as necessary.

- `'none'`: Do no resizing; image will be clipped by the container.

- `'scale-down'`: Do the same thing as either `contain` or `none`, using whichever results in the smaller dispayed image.

- None (the Python value): Remove `object_fit` from the layout; effect is same as `'fill'`.

Use `Image.layout.object_position` to control how where an image is positioned within a container like a box. The default value ensures that the image is centered in the box. The effect of `Image.layout.object_position` depends, in some cases, on the value of `Image.layout.object_fit`.

There are severl ways to specify the value for `object_position`, described below.

Examples of `object_fit`

In the example below an image is displayed inside a green box to demonstrate each of the values for `object_fit`.

To keep the example uniform, define common code here.

```
[22]: from ipywidgets import Layout, Box, VBox, HBox, HTML, Image

      fit_options = ['contain', 'cover', 'fill', 'scale-down', 'none', None]

      hbox_layout = Layout()
      hbox_layout.width = '100%'
      hbox_layout.justify_content = 'space-around'

      green_box_layout = Layout()
      green_box_layout.width = '100px'
      green_box_layout.height = '100px'
      green_box_layout.border = '2px solid green'


      def make_box_for_grid(image_widget, fit):
          """
```

(continues on next page)

ipywidgets Documentation, Release 7.6.2

(continued from previous page)

```
        Make a VBox to hold caption/image for demonstrating
        option_fit values.
        """
        # Make the caption
        if fit is not None:
            fit_str = "'{}'".format(fit)
        else:
            fit_str = str(fit)

        h = HTML(value='' + str(fit_str) + '')

        # Make the green box with the image widget inside it
        boxb = Box()
        boxb.layout = green_box_layout
        boxb.children = [image_widget]

        # Compose into a vertical box
        vb = VBox()
        vb.layout.align_items = 'center'
        vb.children = [h, boxb]
        return vb

# Use this margin to eliminate space between the image and the box
image_margin = '0 0 0 0'

# Set size of captions in figures below
caption_size = 'h4'
```

`object_fit` in a `Box` smaller than the original image

The effect of each can be seen in the image below. In each case, the image is in a box with a green border. The original image is 600x300 and the grid boxes in the image are squares. Since the image is wider than the box width, the content box is the size of the container.

```
[23]: with open('images/gaussian_with_grid.png', 'rb') as f:
          im_600_300 = f.read()
```

```
[24]: boxes = []
      for fit in fit_options:
          ib = Image(value=im_600_300)
          ib.layout.object_fit = fit
          ib.layout.margin = image_margin

          boxes.append(make_box_for_grid(ib, fit))

      vb = VBox()
      h = HTML(value='<{size}>Examples of <code>object_fit</code> with large image</{size}>
      →'.format(size=caption_size))
      vb.layout.align_items = 'center'
      hb = HBox()
      hb.layout = hbox_layout
      hb.children = boxes

      vb.children = [h, hb]
      vb
```

56 **Chapter 1. Full Table of Contents**

```
VBox(children=(HTML(value='<h4>Examples of <code>object_fit</code> with large image</
→h4>'), HBox(children=(VBo...
```

`object_fit` in a `Box` larger than the original image

The effect of each can be seen in the image below. In each case, the image is in a box with a green border. The original image is 50x25 and the grid boxes in the image are squares.

```
[25]: with open('images/gaussian_with_grid_tiny.png', 'rb') as f:
          im_50_25 = f.read()
```

```
[26]: boxes = []
      for fit in fit_options:
          ib = Image(value=im_50_25)
          ib.layout.object_fit = fit
          ib.layout.margin = image_margin
          boxes.append(make_box_for_grid(ib, fit))

      vb = VBox()
      h = HTML(value='<{size}>Examples of <code>object_fit</code> with small image</{size}>
      →'.format(size=caption_size))
      vb.layout.align_items = 'center'
      hb = HBox()
      hb.layout = hbox_layout
      hb.children = boxes

      vb.children = [h, hb]
      vb
```

```
VBox(children=(HTML(value='<h4>Examples of <code>object_fit</code> with small image</
→h4>'), HBox(children=(VBo...
```

It may be surprising, given the description of the values for `option_fit`, that in none of the cases does the image actually fill the box. The reason is that the underlying image is only 50 pixels wide, half the width of the box, so `fill` and `cover` mean "fill/cover the content box determined by the size of the image".

`object_fit` in a `Box` larger than the original image: use image layout width `100%` to fill container

If the width of the image's layout is set to `100%` it will fill the box in which it is placed. This example also illustrates the difference between `'contain'` and `'scale-down'`. The effect of `'scale-down'` is either the same as `'contain'` or `'none'`, whichever leads to the smaller displayed image. In this case, the smaller image comes from doing no fitting, so that is what is displayed.

```
[27]: boxes = []
      for fit in fit_options:
          ib = Image(value=im_50_25)
          ib.layout.object_fit = fit
          ib.layout.margin = image_margin

          # NOTE WIDTH IS SET TO 100%
          ib.layout.width = '100%'

          boxes.append(make_box_for_grid(ib, fit))
```

(continues on next page)

```
vb = VBox()
h = HTML(value='<{size}>Examples of <code>object_fit</code> with image '
                'smaller than container</{size}>'.format(size=caption_size))
vb.layout.align_items = 'center'
hb = HBox()
hb.layout = hbox_layout
hb.children = boxes

vb.children = [h, hb]
vb
```

```
VBox(children=(HTML(value='<h4>Examples of <code>object_fit</code> with image smaller␣
↪than container</h4>'), H...
```

Examples of `object_position`

There are several ways to set object position:

- Use keywords like `top` and `left` to describe how the image should be placed in the container.

- Use two positions, in pixels, for the offset from the top, left corner of the container to the top, left corner of the image. The offset may be positive or negative, and can be used to position the image outside of the box.

- Use a percentage for the offset in each direction. The percentage is a the fraction of the vertical or horizontal *whitespace* around the image if the image is smaller than the container, or the portion of the image outside the container if the image is larger than the container.

- A mix of pixel and percent offsets.

- A mix of keywords and offsets.

Image scaling as determined by `object_fit` will take precendence over the positioning in some cases. For example, if `object_fit` is `fill`, so that the image is supposed to fill the container without preserving the aspect ratio, then `object_position` will have no effect because there is effectively no positioning to do.

Another way to think about it is this: `object_position` specifies how the *white space* around an image should be distributed in a container if there is white space in a particular direction.

Specifying `object_position` with keywords

This form of `object_position` takes two keywords, one for horizontal position of the image in the container and one for the vertical position, in that order.

- The horizontal position must be one of:
  - `'left'`: the left side of the image should be aligned with the left side of the container
  - `'center'`: the image should be centered horizontally in the container.
  - `'right'`: the right side of the image should be aligned with the rigth side of the container.
- The vertical position must be one of
  - `'top'`: the top of the image should be aligned with the top of the container.
  - 'center': the image should be centered vertically in the container.
  - `'bottom'`: the bottom of the image should be aligned with the bottom of the container.

The effect of each is display below, once for an image smaller than the container and once for an image larger than the container.

In the examples below the `object_fit` is set to `'none'` so that the image is not scaled.

```
[28]: object_fit = 'none'
      image_value = [im_600_300, im_50_25]
      horz_keywords = ['left', 'center', 'right']
      vert_keywords = ['top', 'center', 'bottom']

      rows = []
      for image, caption in zip(image_value, ['600 x 300 image', '50 x 25 image']):
          cols = []
          for horz in horz_keywords:
              for vert in vert_keywords:
                  ib = Image(value=image)
                  ib.layout.object_position = '{horz} {vert}'.format(horz=horz, vert=vert)
                  ib.layout.margin = image_margin
                  ib.layout.object_fit = object_fit
                  # ib.layout.height = 'inherit'
                  ib.layout.width = '100%'
                  cols.append(make_box_for_grid(ib, ib.layout.object_position))
          hb = HBox()
          hb.layout = hbox_layout
          hb.children = cols
          rows.append(hb)

      vb = VBox()

      h1 = HTML(value='<{size}><code> object_position </code> by '
                      'keyword with large image</{size}>'.format(size=caption_size))
      h2 = HTML(value='<{size}><code> object_position </code> by '
                      'keyword with small image</{size}>'.format(size=caption_size))

      vb.children = [h1, rows[0], h2, rows[1]]
      vb.layout.height = '400px'
      vb.layout.justify_content = 'space-around'
      vb.layout.align_items = 'center'
      vb
```

```
VBox(children=(HTML(value='<h4><code> object_position </code> by keyword with large␣
→image</h4>'), HBox(childre...
```

Specifying `object_position` with offsets in pixels

One can specify the offset of the top, left corner of the image from the top, left corner of the container in pixels. The first of the two offsets is horizontal, the second is vertical and either may be negative. Using a large enough offset that the image is outside the container will result in the image being hidden.

The image is scaled first using the value of `object_fit` (which defaults to `fill` if nothing is specified) then the offset is applied.

Offsets can be specified from the bottom and/or right side by combining keywords and pixel offsets. For example, `right 10px bottom 20px` offsets the right side of the image 10px from the right edge of the container and the image bottom 20px from the bottom of the container.

```
[29]: object_fit = ['none', 'contain', 'fill', 'cover']
      offset = '20px 10px'
      image_value = [im_600_300]

      boxes = []
      for image, caption  in zip(image_value, ['600 x 300 image', ]):
          for fit in object_fit:
              ib = Image(value=image)
              ib.layout.object_position = offset
              ib.layout.margin = image_margin
              ib.layout.object_fit = fit
              # ib.layout.height = 'inherit'
              ib.layout.width = '100%'
              title = 'object_fit: {}'.format(ib.layout.object_fit)
              boxes.append(make_box_for_grid(ib, title))

      vb = VBox()
      h = HTML(value='<{size}><code>object_position</code> by '
                     'offset {offset} with several '
                     '<code>object_fit</code>s with large image</{size}>'.
       ↪format(size=caption_size,
                                                    offset=offset))
      vb.layout.align_items = 'center'
      hb = HBox()
      hb.layout = hbox_layout
      hb.children = boxes

      vb.children = [h, hb]
      vb
```

```
VBox(children=(HTML(value='<h4><code>object_position</code> by offset 20px 10px with␣
↪several <code>object_fit<...
```

Specifying `object_position` with offsets as a percentage

One can specify the offset of the top, left corner of the image from the top, left corner of the container as a percent.The first of the two offsets is horizontal, the second is vertical and either may be negative. Using a large enough offset that the image is outside the container will result in the image being hidden.

The important thing to understand is that this is a percent of the *white space* in each direction if the image is smaller than the container, so that 50% 50% centers the image.

If the image is larger than the container after scaling using the object_fit, then the offset is a percentage of the overflow of the image outside the container. That means that 50% 50% also centers an image *larger* than the container. A value of 10% 90% would put 10% of the out-of-container part of the image left of the left edge and 90% of the out-of-container part vertically above the top edge.

As with specifying the object_position by keywords, the object_fit can prevent any offset from being applied to the image.

1.1.8 Using Layout Templates

As we showed in *Layout and Styling of Jupyter widgets* multiple widgets can be aranged together using the flexible *GridBox* specification. However, use of the specification involves some understanding of CSS properties and may impose sharp learning curve. Here, we will describe layout templates built on top of `GridBox` that simplify creation of common widget layouts.

```
[1]: # Utils widgets
     from ipywidgets import Button, Layout, jslink, IntText, IntSlider

     def create_expanded_button(description, button_style):
         return Button(description=description, button_style=button_style,
     →layout=Layout(height='auto', width='auto'))

     top_left_button = create_expanded_button("Top left", 'info')
     top_right_button = create_expanded_button("Top right", 'success')
     bottom_left_button = create_expanded_button("Bottom left", 'danger')
     bottom_right_button = create_expanded_button("Bottom right", 'warning')

     top_left_text = IntText(description='Top left', layout=Layout(width='auto', height=
     →'auto'))
     top_right_text = IntText(description='Top right', layout=Layout(width='auto', height=
     →'auto'))
     bottom_left_slider = IntSlider(description='Bottom left', layout=Layout(width='auto',
     →height='auto'))
     bottom_right_slider = IntSlider(description='Bottom right', layout=Layout(width='auto
     →', height='auto'))
```

2x2 Grid

You can easily create a layout with 4 widgets aranged on 2x2 matrix using the `TwoByTwoLayout` widget:

```
[2]: from ipywidgets import TwoByTwoLayout

     TwoByTwoLayout(top_left=top_left_button,
                    top_right=top_right_button,
                    bottom_left=bottom_left_button,
                    bottom_right=bottom_right_button)

     TwoByTwoLayout(children=(Button(button_style='info', description='Top left',
     →layout=Layout(grid_area='top-left...
```

If you don't define a widget for some of the slots, the layout will automatically re-configure itself by merging neighbouring cells

```
[3]: TwoByTwoLayout(top_left=top_left_button,
                    bottom_left=bottom_left_button,
                    bottom_right=bottom_right_button)

     TwoByTwoLayout(children=(Button(button_style='info', description='Top left',
     →layout=Layout(grid_area='top-left...
```

You can pass `merge=False` in the argument of the `TwoByTwoLayout` constructor if you don't want this behavior

```
[4]: TwoByTwoLayout(top_left=top_left_button,
                    bottom_left=bottom_left_button,
```

(continues on next page)

```
                    bottom_right=bottom_right_button,
                    merge=False)
```

```
TwoByTwoLayout(children=(Button(button_style='info', description='Top left',␣
→layout=Layout(grid_area='top-left...
```

You can add a missing widget even after the layout initialization:

```
[5]: layout_2x2 = TwoByTwoLayout(top_left=top_left_button,
                                 bottom_left=bottom_left_button,
                                 bottom_right=bottom_right_button)
     layout_2x2
```

```
TwoByTwoLayout(children=(Button(button_style='info', description='Top left',␣
→layout=Layout(grid_area='top-left...
```

```
[6]: layout_2x2.top_right = top_right_button
```

You can also use the linking feature of widgets to update some property of a widget based on another widget:

```
[7]: app = TwoByTwoLayout(top_left=top_left_text, top_right=top_right_text,
                          bottom_left=bottom_left_slider, bottom_right=bottom_right_slider)

     link_left = jslink((app.top_left, 'value'), (app.bottom_left, 'value'))
     link_right = jslink((app.top_right, 'value'), (app.bottom_right, 'value'))
     app.bottom_right.value = 30
     app.top_left.value = 25
     app
```

```
TwoByTwoLayout(children=(IntText(value=25, description='Top left', layout=Layout(grid_
→area='top-left', height=...
```

You can easily create more complex layouts with custom widgets. For example, you can use bqplot Figure widget to add plots:

```
[8]: import bqplot as bq
     import numpy as np
```

```
[9]: size = 100
     np.random.seed(0)

     x_data = range(size)
     y_data = np.random.randn(size)
     y_data_2 = np.random.randn(size)
     y_data_3 = np.cumsum(np.random.randn(size) * 100.)

     x_ord = bq.OrdinalScale()
     y_sc = bq.LinearScale()

     bar = bq.Bars(x=np.arange(10), y=np.random.rand(10), scales={'x': x_ord, 'y': y_sc})
     ax_x = bq.Axis(scale=x_ord)
     ax_y = bq.Axis(scale=y_sc, tick_format='0.2f', orientation='vertical')

     fig = bq.Figure(marks=[bar], axes=[ax_x, ax_y], padding_x=0.025, padding_y=0.025,
                     layout=Layout(width='auto', height='90%'))
```

Chapter 1. Full Table of Contents

```
[10]: from ipywidgets import FloatSlider

      max_slider = FloatSlider(min=0, max=10, default_value=2, description="Max: ",
                               layout=Layout(width='auto', height='auto'))
      min_slider = FloatSlider(min=-1, max=10, description="Min: ",
                               layout=Layout(width='auto', height='auto'))
      app = TwoByTwoLayout(top_left=min_slider,
                           bottom_left=max_slider,
                           bottom_right=fig,
                           align_items="center",
                           height='700px')

      jslink((y_sc, 'max'), (max_slider, 'value'))
      jslink((y_sc, 'min'), (min_slider, 'value'))
      jslink((min_slider, 'max'), (max_slider, 'value'))
      jslink((max_slider, 'min'), (min_slider, 'value'))

      max_slider.value = 1.5
      app
```

```
TwoByTwoLayout(children=(FloatSlider(value=0.0, description='Min: ',
→layout=Layout(grid_area='top-left', heigh...
```

AppLayout

AppLayout is a widget layout template that allows you to create an application-like widget arrangements. It consist of a header, a footer, two sidebars and a central pane:

```
[11]: from ipywidgets import AppLayout, Button, Layout
```

```
[12]: header_button = create_expanded_button('Header', 'success')
      left_button = create_expanded_button('Left', 'info')
      center_button = create_expanded_button('Center', 'warning')
      right_button = create_expanded_button('Right', 'info')
      footer_button = create_expanded_button('Footer', 'success')
```

```
[13]: AppLayout(header=header_button,
                left_sidebar=left_button,
                center=center_button,
                right_sidebar=right_button,
                footer=footer_button)
```

```
AppLayout(children=(Button(button_style='success', description='Header',
→layout=Layout(grid_area='header', hei...
```

However with the automatic merging feature, it's possible to achieve many other layouts:

```
[14]: AppLayout(header=None,
                left_sidebar=None,
                center=center_button,
                right_sidebar=None,
                footer=None)
```

```
AppLayout(children=(Button(button_style='warning', description='Center',
→layout=Layout(grid_area='center', hei...
```

```
[15]: AppLayout(header=header_button,
                 left_sidebar=left_button,
                 center=center_button,
                 right_sidebar=right_button,
                 footer=None)
```

```
AppLayout(children=(Button(button_style='success', description='Header',␣
↪layout=Layout(grid_area='header', hei...
```

```
[16]: AppLayout(header=None,
                 left_sidebar=left_button,
                 center=center_button,
                 right_sidebar=right_button,
                 footer=None)
```

```
AppLayout(children=(Button(button_style='info', description='Left',␣
↪layout=Layout(grid_area='left-sidebar', he...
```

```
[17]: AppLayout(header=header_button,
                 left_sidebar=left_button,
                 center=center_button,
                 right_sidebar=None,
                 footer=footer_button)
```

```
AppLayout(children=(Button(button_style='success', description='Header',␣
↪layout=Layout(grid_area='header', hei...
```

```
[18]: AppLayout(header=header_button,
                 left_sidebar=None,
                 center=center_button,
                 right_sidebar=right_button,
                 footer=footer_button)
```

```
AppLayout(children=(Button(button_style='success', description='Header',␣
↪layout=Layout(grid_area='header', hei...
```

```
[19]: AppLayout(header=header_button,
                 left_sidebar=None,
                 center=center_button,
                 right_sidebar=None,
                 footer=footer_button)
```

```
AppLayout(children=(Button(button_style='success', description='Header',␣
↪layout=Layout(grid_area='header', hei...
```

```
[20]: AppLayout(header=header_button,
                 left_sidebar=left_button,
                 center=None,
                 right_sidebar=right_button,
                 footer=footer_button)
```

```
AppLayout(children=(Button(button_style='success', description='Header',␣
↪layout=Layout(grid_area='header', hei...
```

You can also modify the relative and absolute widths and heights of the panes using `pane_widths` and `pane_heights` arguments. Both accept a sequence of three elements, each of which is either an integer (equivalent to the weight given to the row/column) or a string in the format `'1fr'` (same as integer) or `'100px'` (absolute size).

```
[21]: AppLayout(header=header_button,
               left_sidebar=left_button,
               center=center_button,
               right_sidebar=right_button,
               footer=footer_button,
               pane_widths=[3, 3, 1],
               pane_heights=[1, 5, '60px'])
```

```
AppLayout(children=(Button(button_style='success', description='Header',␣
→layout=Layout(grid_area='header', hei...
```

Grid layout

`GridspecLayout` is a N-by-M grid layout allowing for flexible layout definitions using an API similar to matplotlib's GridSpec.

You can use `GridspecLayout` to define a simple regularly-spaced grid. For example, to create a 4x3 layout:

```python
[22]: from ipywidgets import GridspecLayout

grid = GridspecLayout(4, 3)

for i in range(4):
    for j in range(3):
        grid[i, j] = create_expanded_button('Button {} - {}'.format(i, j), 'warning')
grid
```

```
GridspecLayout(children=(Button(button_style='warning', description='Button 0 - 0',␣
→layout=Layout(grid_area='w...
```

To make a widget span several columns and/or rows, you can use slice notation:

```python
[23]: grid = GridspecLayout(4, 3, height='300px')
grid[:3, 1:] = create_expanded_button('One', 'success')
grid[:, 0] = create_expanded_button('Two', 'info')
grid[3, 1] = create_expanded_button('Three', 'warning')
grid[3, 2] = create_expanded_button('Four', 'danger')
grid
```

```
GridspecLayout(children=(Button(button_style='success', description='One',␣
→layout=Layout(grid_area='widget001'...
```

You can still change properties of the widgets stored in the grid, using the same indexing notation.

```python
[24]: grid = GridspecLayout(4, 3, height='300px')
grid[:3, 1:] = create_expanded_button('One', 'success')
grid[:, 0] = create_expanded_button('Two', 'info')
grid[3, 1] = create_expanded_button('Three', 'warning')
grid[3, 2] = create_expanded_button('Four', 'danger')

grid
```

```
GridspecLayout(children=(Button(button_style='success', description='One',␣
→layout=Layout(grid_area='widget001'...
```

```python
[25]: grid[0, 0].description = "I am the blue one"
```

Note: It's enough to pass an index of one of the grid cells occupied by the widget of interest. Slices are not supported in this context.

If there is already a widget that conflicts with the position of the widget being added, it will be removed from the grid:

```
[26]: grid = GridspecLayout(4, 3, height='300px')
      grid[:3, 1:] = create_expanded_button('One', 'info')
      grid[:, 0] = create_expanded_button('Two', 'info')
      grid[3, 1] = create_expanded_button('Three', 'info')
      grid[3, 2] = create_expanded_button('Four', 'info')

      grid

      GridspecLayout(children=(Button(button_style='info', description='One',␣
      ↪layout=Layout(grid_area='widget001', h...
```

```
[27]: grid[3, 1] = create_expanded_button('New button!!', 'danger')
```

Note: Slices are supported in this context.

```
[28]: grid[:3, 1:] = create_expanded_button('I am new too!!!!!', 'warning')
```

Creating scatter plots using GridspecLayout

In this examples, we will demonstrate how to use `GridspecLayout` and `bqplot` widget to create a multipanel scatter plot. To run this example you will need to install the bqplot package.

For example, you can use the following snippet to obtain a scatter plot across multiple dimensions:

```
[29]: import bqplot as bq
      import numpy as np
      from ipywidgets import GridspecLayout, Button, Layout

      n_features = 5
      data = np.random.randn(100, n_features)
      data[:50, 2] += 4 * data[:50, 0] **2
      data[50:, :] += 4

      A = np.random.randn(n_features, n_features)/5

      data = np.dot(data,A)

      scales_x = [bq.LinearScale() for i in range(n_features)]
      scales_y = [bq.LinearScale() for i in range(n_features)]

      gs = GridspecLayout(n_features, n_features)
      for i in range(n_features):
          for j in range(n_features):

              if i != j:
                  sc_x = scales_x[j]
                  sc_y = scales_y[i]

                  scatt = bq.Scatter(x=data[:, j], y=data[:, i], scales={'x': sc_x, 'y': sc_
      ↪y}, default_size=1)

                  gs[i, j] = bq.Figure(marks=[scatt], layout=Layout(width='auto', height=
      ↪'auto'),
```

(continues on next page)

```
                                 fig_margin=dict(top=0, bottom=0, left=0, right=0))
        else:
            sc_x = scales_x[j]
            sc_y = bq.LinearScale()

            hist = bq.Hist(sample=data[:,i], scales={'sample': sc_x, 'count': sc_y})

            gs[i, j] = bq.Figure(marks=[hist], layout=Layout(width='auto', height=
→'auto'),
                                 fig_margin=dict(top=0, bottom=0, left=0, right=0))
gs
```

```
GridspecLayout(children=(Figure(fig_margin={'top': 0, 'bottom': 0, 'left': 0, 'right':
→ 0}, layout=Layout(grid_...
```

Style attributes

You can specify extra style properties to modify the layout. For example, you can change the size of the whole layout using the `height` and `width` arguments.

```
[30]: AppLayout(header=None,
          left_sidebar=left_button,
          center=center_button,
          right_sidebar=right_button,
          footer=None,
          height="200px", width="50%")
```

```
AppLayout(children=(Button(button_style='info', description='Left',␣
→layout=Layout(grid_area='left-sidebar', he...
```

The gap between the panes can be increase or decreased with `grid_gap` argument:

```
[31]: AppLayout(header=None,
          left_sidebar=left_button,
          center=center_button,
          right_sidebar=right_button,
          footer=None,
          height="200px", width="50%",
          grid_gap="10px")
```

```
AppLayout(children=(Button(button_style='info', description='Left',␣
→layout=Layout(grid_area='left-sidebar', he...
```

Additionally, you can control the alignment of widgets within the layout using `justify_content` and `align_items` attributes:

```
[32]: from ipywidgets import Text, HTML
      TwoByTwoLayout(top_left=top_left_button, top_right=top_right_button,
                 bottom_right=bottom_right_button,
                 justify_items='center',
                 width="50%",
                 align_items='center')
```

```
TwoByTwoLayout(children=(Button(button_style='info', description='Top left',␣
→layout=Layout(grid_area='top-left...
```

For other alignment options it's possible to use common names (`top` and `bottom`) or their CSS equivalents (`flex-start` and `flex-end`):

```
[33]: TwoByTwoLayout(top_left=top_left_button, top_right=top_right_button,
                     bottom_right=bottom_right_button,
                     justify_items='center',
                     width="50%",
                     align_items='top')
```

```
TwoByTwoLayout(children=(Button(button_style='info', description='Top left',
→layout=Layout(grid_area='top-left...
```

Example

In this notebook you will find a full example using `AppLayout`.

Fictional World Weather
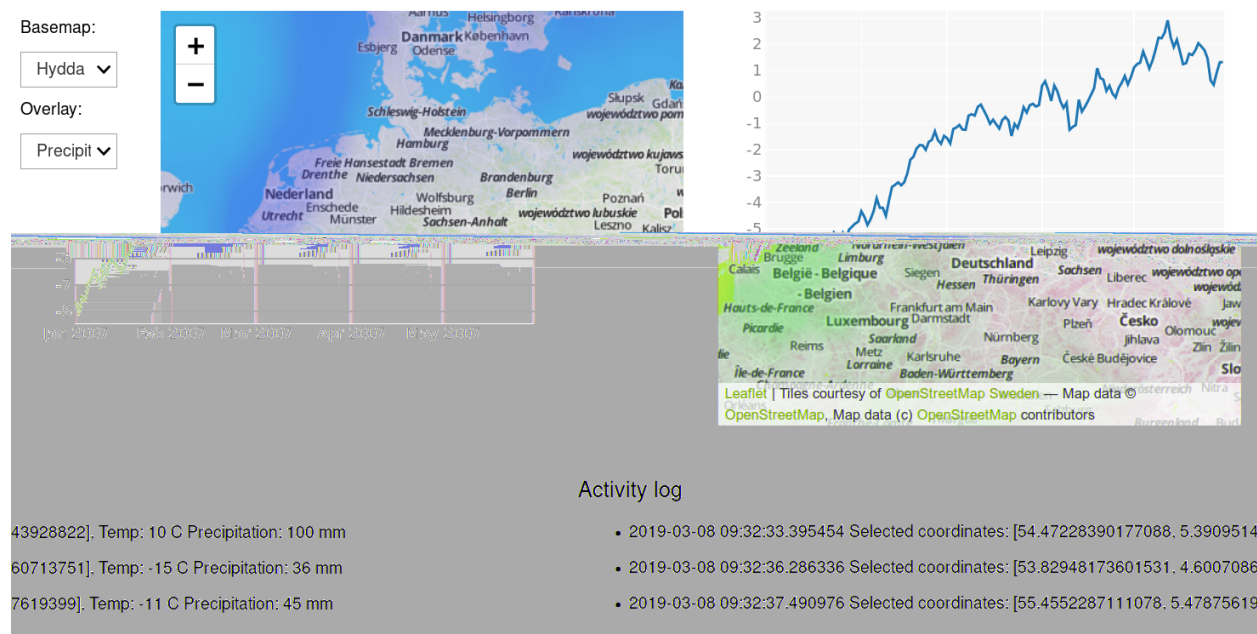

```
[1]: from __future__ import print_function
```

1.1.9 Building a Custom Widget - Email widget

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).

INTERACT

WIDGETS

COMM

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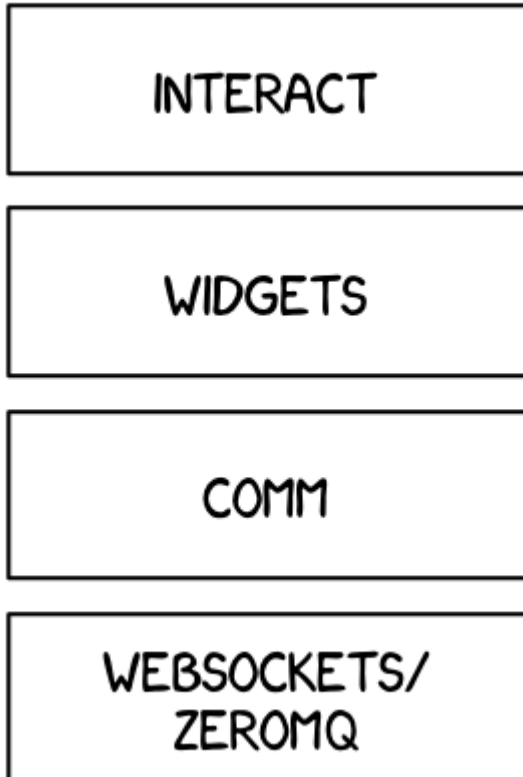
To create a custom widget, you need to define the widget both in the browser and in the python kernel.

Building a Custom Widget

To get started, you'll create a simple email widget.

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, _view_module, and _view_module_version (as seen below) and optionally _model_name and _model_module.

```
[2]: from traitlets import Unicode, Bool, validate, TraitError
     from ipywidgets import DOMWidget, register


     @register
     class Email(DOMWidget):
         _view_name = Unicode('EmailView').tag(sync=True)
         _view_module = Unicode('email_widget').tag(sync=True)
         _view_module_version = Unicode('0.1.0').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, attributes of the widget won't be synchronized with the front-end.

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-widgets/base

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

```
[3]: %%javascript
define('email_widget', ["@jupyter-widgets/base"], 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.

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

    define('email_widget', ["@jupyter-widgets/base"], function(widgets) {

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

        });

        return {
            EmailView: EmailView
        }
    });
```

```
<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.

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

    define('email_widget', ["@jupyter-widgets/base"], function(widgets) {

        var EmailView = widgets.DOMWidgetView.extend({

            // Render the view.
            render: function() {
                this.email_input = document.createElement('input');
                this.email_input.type = 'email';
                this.email_input.value = 'example@example.com';
                this.email_input.disabled = true;

                this.el.appendChild(this.email_input);
            },
        });

        return {
            EmailView: EmailView
        };
    });
```

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

Test

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

```
[6]: Email()
```

```
Email()
```

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 "example@example.com" email address, it will display an address 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.

We want to be able to avoid user to write an invalid email address, so we need a validator using traitlets.

```
[7]: from traitlets import Unicode, Bool, validate, TraitError
     from ipywidgets import DOMWidget, register


     @register
     class Email(DOMWidget):
         _view_name = Unicode('EmailView').tag(sync=True)
         _view_module = Unicode('email_widget').tag(sync=True)
         _view_module_version = Unicode('0.1.0').tag(sync=True)

         # Attributes
         value = Unicode('example@example.com', help="The email value.").tag(sync=True)
         disabled = Bool(False, help="Enable or disable user changes.").tag(sync=True)

         # Basic validator for the email value
         @validate('value')
         def _valid_value(self, proposal):
             if proposal['value'].count("@") != 1:
                 raise TraitError('Invalid email value: it must contain an "@" character')
             if proposal['value'].count(".") == 0:
                 raise TraitError('Invalid email value: it must contain at least one "."␣
     ↪character')
             return proposal['value']
```

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.

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

     define('email_widget', ["@jupyter-widgets/base"], function(widgets) {

         var EmailView = widgets.DOMWidgetView.extend({

             // Render the view.
             render: function() {
                 this.email_input = document.createElement('input');
                 this.email_input.type = 'email';
                 this.email_input.value = this.model.get('value');
                 this.email_input.disabled = this.model.get('disabled');

                 this.el.appendChild(this.email_input);
             },
         });

         return {
             EmailView: EmailView
         };
     });
```

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

```
[9]: Email(value='john.doe@domain.com', disabled=True)
```

```
Email(disabled=True, value='john.doe@domain.com')
```

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).

```
[10]: %%javascript
      require.undef('email_widget');

      define('email_widget', ["@jupyter-widgets/base"], function(widgets) {

          var EmailView = widgets.DOMWidgetView.extend({

              // Render the view.
              render: function() {
                  this.email_input = document.createElement('input');
                  this.email_input.type = 'email';
                  this.email_input.value = this.model.get('value');
                  this.email_input.disabled = this.model.get('disabled');
```

```javascript
        this.el.appendChild(this.email_input);

        // Python -> JavaScript update
        this.model.on('change:value', this.value_changed, this);
        this.model.on('change:disabled', this.disabled_changed, this);
    },

    value_changed: function() {
        this.email_input.value = this.model.get('value');
    },

    disabled_changed: function() {
        this.email_input.disabled = this.model.get('disabled');
    },
});

    return {
        EmailView: EmailView
    };
});
```

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

This allows us to update the value from the Python kernel to the views. Now to get the value updated from the front-end to the Python kernel (when the input is not disabled) we can do it using the `model.set` method.

```javascript
[11]: %%javascript
require.undef('email_widget');

define('email_widget', ["@jupyter-widgets/base"], function(widgets) {

    var EmailView = widgets.DOMWidgetView.extend({

        // Render the view.
        render: function() {
            this.email_input = document.createElement('input');
            this.email_input.type = 'email';
            this.email_input.value = this.model.get('value');
            this.email_input.disabled = this.model.get('disabled');

            this.el.appendChild(this.email_input);

            // Python -> JavaScript update
            this.model.on('change:value', this.value_changed, this);
            this.model.on('change:disabled', this.disabled_changed, this);

            // JavaScript -> Python update
            this.email_input.onchange = this.input_changed.bind(this);
        },

        value_changed: function() {
            this.email_input.value = this.model.get('value');
        },

        disabled_changed: function() {
            this.email_input.disabled = this.model.get('disabled');
        },
```

```
        input_changed: function() {
            this.model.set('value', this.email_input.value);
            this.model.save_changes();
        },
    });

    return {
        EmailView: EmailView
    };
});
```
```
<IPython.core.display.Javascript object>
```

Test

```
[12]: email = Email(value='john.doe@domain.com', disabled=False)
      email
```
```
Email(value='john.doe@domain.com')
```

```
[13]: email.value
```
```
[13]: 'john.doe@domain.com'
```

```
[14]: email.value = 'jane.doe@domain.com'
```

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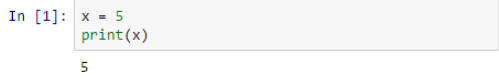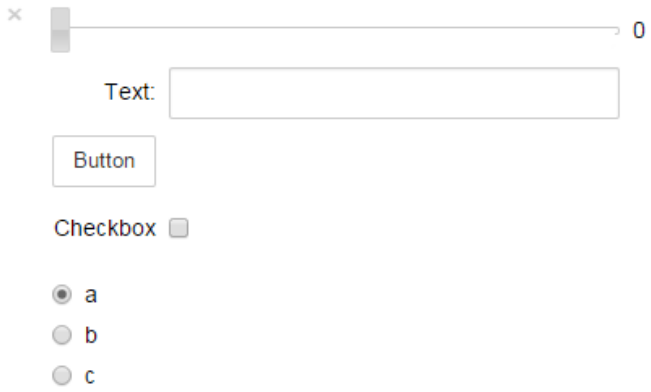
It produces a project for a Jupyter interactive widget library following the current best practices for using interactive widgets. An implementation for a placeholder "Hello World" widget is provided.

1.1.10 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.

In [1]: `x = 5` `print(x)` 5	Text: [ ] [Button] Checkbox ☐ ⦿ a ○ b ○ c
Before the widgets, this was just the segmentation of code and results from executing those segments.	Widgets further decrease the distance between the user and their data by allowing UI interactions to directly manipulate data in the kernel.

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 eachother. If the WidgetModel is changed in the front-end, the Widget receives the same change in the kernel. Vise versa, if the Widget in the kernel is changed, the WidgetModel in the front-end receives the same change. There is no single source of truth, both models have the same precedence. Although a notebook has the notion of cells, neither Widget or WidgetModel are bound to any single cell.

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.

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

```
IntSlider(value=0)
```

```
IntSlider(value=0)
```

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

```
IntSlider(value=0)
```

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 Widget-Model 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

```
[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:

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

class MyWidget(DOMWidget):
    _view_module = Unicode('mywidget').tag(sync=True)
    _view_module_version = Unicode('0.1.0').tag(sync=True)
    _view_name = Unicode('MyWidgetView').tag(sync=True)
    count = Int().tag(sync=True)
```

JavaScript:

```javascript
define('mywidget', ['@jupyter-widgets/base'], function(widgets) {
    var MyWidgetView = widgets.DOMWidgetView.extend({
```

(continues on next page)

```javascript
        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`. The `DOMWidget` class represents a widget that is represented in the page as an HTML DOM element. The `Widget` class is more general and can be used for objects that may not live on the page as a DOM element (for example, a widget inheriting from `Widget` may represent a Javascript object).

`_view_module`, `_view_module_version`, 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`, `_model_module_version`, 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`. The `DOMWidgetView` class is for widgets that render themselves into a DOM element, and the `WidgetView` class does not make this assumption.

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 must be 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 serializers 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.

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-widgets/widget-cookiecutter

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

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

1.1.11 Asynchronous Widgets

This notebook covers two scenarios where we'd like widget-related code to run without blocking the kernel from acting on other execution requests:

1. Pausing code to wait for user interaction with a widget in the frontend

2. Updating a widget in the background

Waiting for user interaction

You may want to pause your Python code to wait for some user interaction with a widget from the frontend. 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 invoke our asyncio event loop. This requires ipykernel 4.7 or later.

```
[1]: %gui asyncio
```

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

```
[2]: 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.

```
[3]: 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
```

```
IntSlider(value=0)
```

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.

```
[4]: 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.

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

     @yield_for_change(slider2, 'value')
     def f():
         for i in range(10):
             print('did work %s'%i)
             x = yield
             print('generator function continued with value %s'%x)
     f()

     slider2
```

```
did work 0
```

```
IntSlider(value=0)
```

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.

Updating a widget in the background

Sometimes you'd like to update a widget in the background, allowing the kernel to also process other execute requests. We can do this with threads. In the example below, the progress bar will update in the background and will allow the main kernel to do other computations.

```python
[6]: import threading
     from IPython.display import display
     import ipywidgets as widgets
     import time
     progress = widgets.FloatProgress(value=0.0, min=0.0, max=1.0)

     def work(progress):
         total = 100
         for i in range(total):
             time.sleep(0.2)
             progress.value = float(i+1)/total

     thread = threading.Thread(target=work, args=(progress,))
     display(progress)
     thread.start()
```

```
FloatProgress(value=0.0, max=1.0)
```

1.1.12 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

Here, we discuss embedding widgets using the custom widget manager in the `@jupyter-widgets/html-manager` npm package. Two embedders are provided:

1. A basic embedder that only embeds standard controls, but can be used on any web page
2. An embedder that uses RequireJS, and can embed standard and custom widgets

Embedding Widgets in HTML Web Pages

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

The menu provides three sets of actions

- Save Notebook Widget State and Clear Notebook Widget State
- Download Widget State
- Embed Widgets

Save Notebook Widget State

A notebook file may be saved with the current widget state as metadata. This allows the notebook file to be rendered with rendered widgets (see the section about Sphinx below, for example). To save a notebook with the current widget state, use the `Save Notebook Widget State` menu item.

In order to delete old saved state and save new state to the notebook, do the following in order:

1. Use the `Clear Notebook Widget State` menu and save the notebook. This clears the metadata from the notebook file.

2. Restart the kernel and refresh the page. This clears the old widget state from the widget manager on the page.

3. Create whatever widgets you'd like, and use `Save Notebook Widget State` and save the notebook. This saves the new widget state to the notebook file.

Embeddable HTML Snippet

The `Embed widgets` menu item provides a dialog containing an HTML page which embeds the current widgets. In order to support custom widgets, it uses the RequireJS embedder.

This HTML snippet is composed of multiple `<script>` tags embedded into an HTML document:

- The first script tag loads RequireJS from a CDN. If you already have RequireJS on the page, you can delete this script tag.

- The second script tag loads the RequireJS widget embedder. This defines appropriate modules and then sets up a function to render all of the widget views included on the page. If you are only embedding standard widgets and do not want to use RequireJS, you can replace these first two script tags with a script tag loading the standard embedder.

- The next script tag is a script tag with mime type `application/vnd.jupyter.widget-state+json` that contains the state of all the widget models currently in use. The JSON schema for the content of this script tag is found in the `@jupyter-widgets/schema` npm package.

- Then there are a number of script tags, each with mime type `application/vnd.jupyter.widget-view+json`, corresponding to the views which you want to display in the web page. These script tags must be in the body of the page, and are replaced with the rendered widgets. The JSON schema for the content of these script tags is found in the `@jupyter-widgets/schema` npm package.

  The *Embed Widgets* action currently creates one of these script tags for each view displayed in the notebook. If you'd like to lay out the views, or include only some of them, you can delete or include these script tags as you wish.

In order to clear widget state from the frontend so that it does not show up in the embedding, restart the kernel and then refresh the page, in that order.

Widget State JSON

The `Download Widget State` option triggers the downloading of a JSON file containing the serialized state of all the widget models currently in use, using the `application/vnd.jupyter.widget-state+json` format specified in the `@jupyter-widgets/schema` npm package.

Python interface

Embeddable code for the widgets can also be produced from Python. The `ipywidgets.embed` module provides several functions for embedding widgets into HTML documents programatically.

Use `embed_minimal_html` to create a simple, stand-alone HTML page:

```python
from ipywidgets import IntSlider
from ipywidgets.embed import embed_minimal_html

slider = IntSlider(value=40)
embed_minimal_html('export.html', views=[slider], title='Widgets export')
```

This creates the stand-alone file `export.html`. To view the file, either start an HTTP server, such as the HTTP server in the Python standard library, or just open it in your web browser (by double-clicking on the file, or by writing `file:///path/to/file` in your browser search bar).

You will sometimes want greater granularity than that afforded by `embed_minimal_html`. Often, you want to control the structure of the HTML document in which the widgets are embedded. For this, use `embed_data` to get JSON exports of specific parts of the widget state. You can embed these in an HTML template:

```python
import json

from ipywidgets import IntSlider
from ipywidgets.embed import embed_data

s1 = IntSlider(max=200, value=100)
s2 = IntSlider(value=40)
data = embed_data(views=[s1, s2])

html_template = """
<html>
  <head>

    <title>Widget export</title>

    <!-- Load RequireJS, used by the IPywidgets for dependency management -->
    <script
      src="https://cdnjs.cloudflare.com/ajax/libs/require.js/2.3.4/require.min.js"
      integrity="sha256-Ae2Vz/4ePdIu6ZyI/5ZGsYnb+m0JlOmKPjt6XZ9JJkA="
      crossorigin="anonymous">
    </script>

    <!-- Load IPywidgets bundle for embedding. -->
    <script
      data-jupyter-widgets-cdn="https://cdn.jsdelivr.net/npm/"
      src="https://unpkg.com/@jupyter-widgets/html-manager@*/dist/embed-amd.js"
      crossorigin="anonymous">
    </script>

    <!-- The state of all the widget models on the page -->
    <script type="application/vnd.jupyter.widget-state+json">
      {manager_state}
    </script>
  </head>

  <body>
```

```
    <h1>Widget export</h1>

    <div id="first-slider-widget">
      <!-- This script tag will be replaced by the view's DOM tree -->
      <script type="application/vnd.jupyter.widget-view+json">
        {widget_views[0]}
      </script>
    </div>

    <hrule />

    <div id="second-slider-widget">
      <!-- This script tag will be replaced by the view's DOM tree -->
      <script type="application/vnd.jupyter.widget-view+json">
        {widget_views[1]}
      </script>
    </div>

  </body>
</html>
"""

manager_state = json.dumps(data['manager_state'])
widget_views = [json.dumps(view) for view in data['view_specs']]
rendered_template = html_template.format(manager_state=manager_state, widget_
→views=widget_views)
with open('export.html', 'w') as fp:
    fp.write(rendered_template)
```

The web page needs to load RequireJS and the Jupyter widgets HTML manager. You then need to include the manager state in a `<script>` tag of type `application/vnd.jupyter.widget-state+json`, which can go in the head of the document. For each widget view, place a `<script>` tag of type `application/vnd.jupyter.widget-view+json` in the DOM element that should contain the view. The widget manager will replace each `<script>` tag with the DOM tree corresponding to the widget.

In this example, we used a Python string for the template, and used the `format` method to interpolate the state. For embedding in more complex documents, you may want to use a templating engine like Jinja2.

We also change the CDN from its default of unpkg to use jsdelivr by setting the `data-jupyter-widgets-cdn` attribute.

In all embedding functions in `ipywidgets.embed`, the state of all widgets known to the widget manager is included by default. You can alternatively pass a reduced state to use instead. This can be particularly relevant if you have many independent widgets with a large state, but only want to include the relevant ones in your export. To include only the state of specific views and their dependencies, use the function `dependency_state`:

```
from ipywidgets.embed import embed_minimal_html, dependency_state

s1 = IntSlider(max=200, value=100)
s2 = IntSlider(value=40)
embed_minimal_html('export.html', views=[s1, s2], state=dependency_state([s1, s2]))
```

Embedding Widgets in the Sphinx HTML Documentation

As of ipywidgets 6.0, Jupyter interactive widgets can be rendered 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 installed with `pip` and `conda`.

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

Then use the `jupyter-execute` directive to embed the output of code execution in your documentation

```
.. jupyter-execute::

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

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.

For notebooks that are executed by `nbsphinx` the widget state is automatically generated. For others, it is a requirement that the notebook was correctly saved with the special "Save Notebook Widget State" action in the widgets menu.

The necessary JavaScript code is automatically embedded in the generated HTML files. A custom URL or a local JavaScript file can be specified with the `nbsphinx_widgets_path` configuration option. For more configuration options, have a look at the documentation.

Rendering Interactive Widgets on nbviewer

If your notebook was saved with the special "Save Notebook Widget State" 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 npm CDN https://unpkg.com by default. 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.

While the default CDN is using https://unpkg.com it can be configured by setting the optional `data-jupyter-widgets-cdn` attribute for script tag which loads `embed-amd.js`, as shown in the example above.

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-widgets-controls` in web contexts

The core `jupyter-widgets-controls` 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-widgets/base` to provide the logic for

- where widgets should be displayed,

- how to retrieve information about their state.

Specifically:

- The `widgetsnbextension` Python package provides the implementation of a specialized widget manager for the classic Jupyter notebook, and the packaging logic as a notebook extension.

- The `@jupyter-widgets/jupyterlab-manager` npm package 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 the `@jupyter-widgets/html-manager` npm package 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 widgets in web contexts.

1. The `web1` example is a simplistic example showcasing the use of Jupyter 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 `web-tmpnb` 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.

1.1.13 Contributing

We appreciate contributions from the community.

We follow the IPython Contributing Guide and Jupyter Contributing Guides.

1.1.14 Migrating custom widget libraries

These are migration guides aimed specifically at developers of third-party widgets.

Migrating from 6.0 to 7.0

For example migrations, see these PRs:

- ipyleaflet
- jupyter-gmaps
- bqplot: PR #458, PR #497 and PR #501

To avoid tying your development cycle to ipywidgets, we recommend starting the migration on a branch and keeping that branch open until ipywidgets 7.0 is released.

We also recommend testing the migration in a completely new notebook, rather than one that contains widgets that you instantiated with ipywidgets 6.0.

Updating setup.py

Start by updating the dependency in your `setup.py` to the latest release. To find the correct version number, go to the releases page on Github and cycle through the tags until you see the latest 7.0.0 tag.

Updating package.json

Next, we should update the JavaScript dependencies. The most important change for widget developers is that the JavaScript package for jupyter-widgets has been split between `@jupyter-widgets/base` and `@jupyter-widgets/controls`:

- `@jupyter-widgets/base` contains the base widget classes and the layout classes
- `@jupyter-widgets/controls` contains the widget classes for the user-facing widgets.

In your `package.json`, remove `jupyter-js-widgets` from your dependencies and add `@jupyter-widgets/base`. To find the correct version, go to the releases page and cycle through the tags until you see a tag called `@jupyter-widgets/base@<version>`. Do the same for `@jupyter-widgets/controls` if you think you have a dependency on it (e.g. if you create widgets that inherit from `VBox` or `HBox` or another user-facing widget).

Updating Webpack configuration

If you use Webpack to build the client-side library, your Webpack configuration file (probably at `js/webpack.config.json`) declares `jupyter-js-widgets` as an external dependency. You will need to change this in both the bundle for the notebook and the embeddable bundle. If you just need `@jupyter-widgets/base`, your external dependencies would be:

```
externals: ['@jupyter-widgets/base']
```

If you need both `@jupyter-widgets/base` and `@jupyter-widgets/controls`, include both packages in the array.

The cookiecutter template provides a sample configuration.

Updating the client-side code

If you now build the client-side code of your library, you will get many errors about missing `jupyter-js-widgets` dependencies. You need to replace every import of `jupyter-js-widgets` with an import of `@jupyter-widgets/base` (or, possibly, an import of `@jupyter-widgets/controls`).

Your imports should now look like one of the following (depending on how you normally import other modules):

```
widgets = require('@jupyter-widgets/base')
```

```
require(['@jupyter-widgets/base'], function(widgets) {
})
```

```
import * as widgets from '@jupyter-widgets/base'
```

All your widget models should also declare the attributes `_view_module_version` and `_model_module_version`. A minimal model now looks like:

```
var HelloModel = widgets.DOMWidgetModel.extend({
    defaults: _.extend(widgets.DOMWidgetModel.prototype.defaults(), {
        _model_name : 'HelloModel',
        _view_name : 'HelloView',
        _model_module : 'example_module',
        _view_module : 'example_module',
        _model_module_version : '~1.0.0',
        _view_module_version : '~1.0.0'
    })
});
```

For embedding to work correctly, the module version needs to be a semantic version range that matches a release on NPM. The most common pattern is to request a version compatible with the version currently in your `package.json` by using, `~{{ version number }}` for `_model_module_version` and `_view_module_version`. See the cookiecutter template for details.

Since you probably want to avoid repeating the module version in every widget, a common pattern is to import the version from `package.json` and prepend a `~`. See ipyvolume for an example. If you do this, make sure that your webpack configuration includes a JSON loader. See the Webpack configuration for ipyvolume for an example.

Updating the notebook extension

Previously, the notebook extension (normally `js/src/extension.js`) required defining `jupyter-js-widgets` in the configuration for `requirejs`. This is no longer needed. See the cookiecutter template for an example of the correct `requirejs` configuration.

Updating the Python code

All widgets need to declare the following six traitlets:

```
class ExampleWidget(widgets.Widget):
    _model_name = Unicode('name of model in JS')
    _view_name = Unicode('name of view in JS')
    _model_module = Unicode('name your JS package')
    _view_module = Unicode('name your JS package')
```

(continues on next page)

```
    _model_module_version = Unicode('version of your JS bundle')
    _view_module_version = Unicode('version of your JS bundle')
```

It is likely that your widgets already declared a _model_name, _view_name, _model_module and _view_module. The _model_module and _view_module should be the name of your package on NPM (the value of the name field in your package.json). The _model_module_version and _view_module_version should be the version of your JavaScript client (the values of the version field in your package.json).

The _model_module_version and _view_module_version are used to find your JavaScript bundle when embedding widgets. The embed manager will look for the bundle at `https://unpkg.com/<module-name>@<module-version>/dist/index.js` when it finds a widget.

Updating embedded widgets

There are now two options for embedding widgets in an HTML page outside of the notebook.

Embedding the standard widgets

If you are just embedding the standard widgets that come with ipywidgets, then you can simply include the following script tag:

```
<script src="https://unpkg.com/@jupyter-widgets/html-manager@*/dist/embed.js"
→crossorigin="anonymous"></script>
```

If you want to use a specific version of the embedder, you replace the `@*` with a semver range, such as `@^0.9.0`

Embedding custom widgets with RequireJS

In order to embed third-party widgets, you can use the RequireJS-based embedding. First, make sure that RequireJS is loaded on the page, for example:

```
<!-- Load require.js. Delete this if your page already loads require.js -->
<script src="https://cdnjs.cloudflare.com/ajax/libs/require.js/2.3.4/require.min.js"
→integrity="sha256-Ae2Vz/4ePdIu6ZyI/5ZGsYnb+m0JlOmKPjt6XZ9JJkA=" crossorigin=
→"anonymous"></script>
```

Then include the following script, which defines the embedding libraries and runs the function to render widgets:

```
<script src="https://unpkg.com/@jupyter-widgets/html-manager@*/dist/embed-amd.js"
→crossorigin="anonymous"></script>
```

If you want to use a specific version of the embedder, you replace the `@*` with a semver range, such as `@^0.9.0`

If you need to embed custom widgets without using RequireJS, you'll need to compile your own embedding javascript that includes the third-party libraries.

1.1.15 ipywidgets changelog

A summary of changes in ipywidgets. For more detailed information, see the issues and pull requests for the appropriate milestone on GitHub.

7.6

To see the full list of pull requests and issues, see the 7.6.0 milestone on GitHub.

The main change in this release is that installing `ipywidgets` 7.6.0 will now automatically enable ipywidgets support in JupyterLab 3.0—a user has no extra JupyterLab installation step and no rebuild of JupyterLab, nor do they need Node.js installed. Simply install the python ipywidgets package with pip (`pip install ipywidgets==7.6.0`) or conda/mamba (`conda install -c conda-forge ipywidgets=7.6.0`) and ipywidgets will automatically work in classic Jupyter Notebook and in JupyterLab 3.0.

This is accomplished with the new python package `jupyterlab_widgets` version 1.0, on which `ipywidgets` 7.6.0 now depends (similar to how `ipywidgets` already depends on the `widgetsnbextension` package to configure ipywidgets for the classic Jupyter Notebook). The `jupyterlab_widgets` Python package is a JupyterLab 3.0 prebuilt extension, meaning that it can be installed into JupyterLab 3.0 without rebuilding JupyterLab and without needing Node.js installed.

Updates for Widget Maintainers

Custom widget maintainers will need to make two changes to update for JupyterLab 3:

1. Update the `@jupyter-widgets/base` dependency version to include `^4` to work in JupyterLab 3.0. For example, if you had a dependency on `@jupyter-widgets/base` version `^2 || ^3`, update to `^2 || ^3 || ^4` for your widget to work in classic Jupyter Notebook, JupyterLab 1, JupyterLab 2, and JupyterLab 3. See #2472 for background.

2. In the `package.json`, add the following `sharedPackages` configuration inside the `jupyterlab` key. See the JupyterLab extension documentation for more information.

```
"jupyterlab": {
  "sharedPackages": {
    "@jupyter-widgets/base": {
      "bundled": false,
      "singleton": true
    }
  }
}
```

Separate from these two steps to update for JupyterLab 3, we also recommend that you make your widget's JupyterLab extension a prebuilt extension for JupyterLab 3.0. Users will be able to install your JupyterLab 3.0 prebuilt extension without rebuilding JupyterLab or needing Node.js. See the JupyterLab 3 extension developer documentation or the new widget extension cookiecutter for more details.

7.5

To see the full list of pull requests and issues, see the 7.5 milestone on GitHub.

Changes include:

- New `AppLayout` and `GridLayout` templates for positioning interactive widgets. #2333
- New `FileUpload` widget allowing users to upload files from the browser. #2258
- New `ComboBox` widget. #2390
- JupyterLab CSS variables are now exposed by default even in the case of the classic notebook. #2418

Updates for Widget Maintainers

Custom widget maintainers will need to update their `@jupyter-widgets/base` dependency version to work in JupyterLab 1.0. For example, if you had a dependency on `@jupyter-widgets/base` version `^1.1`, update to `^1.1 || ^2` for your widget to work in classic notebook, JupyterLab 0.35, and JupyterLab 1.0. See #2472 for background.

7.4

To see the full list of pull requests and issues, see the 7.4 milestone on GitHub.

Changes include:

- New `Video` and `Audio` widgets have been introduced. #2162 We updated the `@jupyter-widgets/controls` widget specification version to `1.4.0`, leading to the version bump to 7.4.
- The use of mappings for the `options` attribute of selection widgets is deprecated. #2130

7.3

To see the full list of pull requests and issues, see the 7.3 milestone on GitHub.

Changes include:

- A new `GridBox` widget is introduced and associated CSS grid properties are added to the layout. This enables using the CSS Grid spec for laying out widgets. See the Widget Styling documentation for some examples. Because of this and other model specification changes, the view and module versions of widgets was incremented in both the base and controls packages. (#2107, #2064, #1942)
- Widgets with a `description` attribute now also have a `description_tooltip` attribute to set a tooltip on the description. The tooltip defaults to the description text. Setting `description_tooltip` to `''` removes it, and setting it to `None` makes the tooltip default to the description text. (#2070)
- `'transparent'` is now a valid color for color attributes. (#2128)
- Dropdowns now have extra padding to make room for the dropdown arrow. (#2052, #2101)
- Image widget `repr` now truncates the image value to prevent huge amounts of output in notebooks. (#2111)
- Python 3.3 support is dropped. Python 3.3 support was dropped in the Python community in September 2017. (#2129)
- The license information has been consolidated into the LICENSE file, and the COPYING.md file is removed. If you are repackaging ipywidgets or widgetsnbextension, please make sure to include LICENSE instead of COPYING.md. (#2133, #2048, #1701, #1706)

7.2

To see the full list of pull requests and issues, see the 7.2 milestone on GitHub.

User-visible changes include:

- A new `FloatLogSlider` widget is a slider with a log scale, suitable for exploring a wide range of magnitudes. (#1928, #2014)

```
from ipywidgets import FloatLogSlider
FloatLogSlider()
```

- `link` and `dlink` are now exported from ipywidgets for convenience, so that you can import them directly from ipywidgets instead of needing to import them from traitlets. (#1923)

- A new option `manual_name` has been added to `interact_manual()` to change the name of the update button, for example `interact_manual(manual_name='Update')`. (#1924)

- The Output widget now has a `.capture()` method, which returns a decorator to capture the output of a function.

```
from ipywidgets import Output
out = Output()

@out.capture()
def f():
  print('This output is captured')
```

The `.capture()` method has a `clear_output` boolean argument to automatically clear the output every time the function is run, as well as a `wait` argument corresponding to the `clear_output` wait argument. (#1934)

- The Output widget has much more comprehensive documentation in its own section. (#2020)

- Installing `widgetsnbextension` now automatically enables the nbextension in Jupyter Notebook 5.3 or later. (#1911)

- The default rendering of a widget if widgets are not installed is now a short description of the widget in text instead of a much longer HTML message. (#2007)

Changes for developers include:

- The JavaScript base widget manager class now has a `resolveUrl` method to resolve a URL relative to the current notebook location. (#1993)

- The html manager now exposes a way to specify which JavaScript file is fetched for a package and the loader used to fetch the library. (#1995, #1998)

- The `@jupyter-widgets/controls` widget specification version was bumped to `1.2.0`. Changes include the FloatLogSlider widget and more specific documentation about array element types. (#2017)

7.1

To see the full list of pull requests and issues, see the 7.1 milestone on GitHub.

We updated the `@jupyter-widgets/controls` widget specification version to `1.1.0`, leading to the version bump to 7.1. The new widget model specification now includes new `description_width` and `font_weight` attributes for the `ToggleButtonsStyle` widget. There are also other bugfixes in this release.

7.0.x patch releases

See the GitHub milestones for the 7.0.1, 7.0.2, 7.0.3, 7.0.4, and 7.0.5 releases for bugfixes in these releases.

7.0

To see the full list of pull requests and issues, see the 7.0 milestone on GitHub.

Major user-visible changes in ipywidgets 7.0 include:

- 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)

- 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)

- The documentation theme is changed to the new standard Jupyter theme. (#1363)

- The `layout` and `style` traits can be set with a dictionary for convenience, which will automatically converted to a Layout or Style object, like `IntSlider(layout={'width': '100%'}, style={'handle_color': 'lightgreen'})`. (#1253)

- 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`, `ToggleButtons`, etc.) have new `.value`, `.label`, and `.index` traits to make it easier to access or change the selected option. (#1262, #1513)

- Selection container widgets (`Accordion`, `Tabs`) can have their `.selected_index` set to `None` to deselect all items. (#1495)

- The `Play` widget range is now inclusive (max value is max, instead of max-1), to be consistent with Sliders

- The `Play` widget now has an optional repeat toggle button (visible by default). (#1190)

- A refactoring of the text, slider, slider range, and progress widgets in resulted in the progress widgets 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)

- 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 `Label` widget now has no width restriction. ([#1269](#))

- The description width is now configurable with the `.style.description_width` attribute ([#1376](#))

- ToggleButtons have a new `.style.button_width` attribute to set the CSS width of the buttons. Set this to `'initial'` to have buttons that individually size to the content width. ([#1257](#))

- The `readout_format` attribute of number sliders now validates its argument. ([#1550](#))

- The `IntRangeSlider` widget now has a `.readout_format` trait to control the formatting of the readout. ([#1446](#))

- The `Text`, `Textarea`, `IntText`, `BoundedIntText`, `FloatText`, and `BoundedFloatText` widgets all gained a `continuous_update` attribute (defaults to `True` for `Text` and `TextArea`, and `False` for the others). ([#1545](#))

- The `IntText`, `BoundedIntText`, `FloatText`, and `BoundedFloatText` widgets are now rendered as HTML number inputs, and have a `step` attribute that controls the resolution. ([#1545](#))

- The `Text.on_submit` callback is deprecated; instead, set `continuous_update` to `False` and observe the `value` attribute: `mywidget.observe(callback, 'value')`. The `Textarea.scroll_to_bottom` method was removed. ([#1545](#))

- The `msg_throttle` attribute on widgets is now gone, and the code has a hardcoded message throttle equivalent to `msg_throttle=1`. ([#1557](#))

- Using function annotations to specify interact controls for a function is now deprecated and will be removed in a future version of ipywidgets. ([#1292](#))

- There are now two simple ways to embed widgets in an HTML page: with a simple script tag that does not use require.js and does not support anything but the basic widgets, and a require module that does support custom widgets. See the migration guide for more details. ([#1615](#), [#1629](#), [#1630](#))

If you are developing a custom widget or widget manager, here are some major changes that may affect you. The *migration guide* also walks through how to upgrade a custom widget.

- On the Python/kernel side:

  - 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 Python `ipywidget.domwidget.LabeledWidget` is now `ipywidget.widget_description.DescriptionWidget`, and there is a new `ipywidget.widget_description.DescriptionStyle` that lets the user set the CSS width of the description.

  - Custom serializers can now return a structure that contains binary objects (`memoryview`, `bytearray`, or Python 3 `bytes` object). In this case, the sync message will be a binary message, which is much more efficient for binary data than base64-encoding. The Image widget now uses this binary synchronization. ([#1194](#), [#1595](#), [#1643](#))

- On the Javascript side:

  - The `jupyter-js-widgets` Javascript package has been split into `@jupyter-widgets/base` package (containing base widget classes, the DOM widget, and the associated layout and style classes), and the `@jupyter-widgets/controls` package (containing the rest of the Jupyter widgets controls). Authors of custom widgets will need to update to depend on `@jupyter-widgets/base` instead of

jupyter-js-widgets (if you use a class from the controls package, you will also need to depend on @jupyter-widgets/controls). See the [cookie cutter](#) to generate a simple example custom widget using the new packages.

- 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 can now return a structure that contains binary objects (`ArrayBuffer`, `DataView`, or a typed array such as `Int8Array`, `Float64Array`, etc.). In this case, the sync message will be a binary message, which is much more efficient for binary data than base64-encoding. The Image widget now uses this binary synchronization. ([#1194](#), [#1643](#))

- 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 Javascript model `.id` attribute has been renamed to `.model_id` to avoid conflicting with the Backbone `.id` attribute. ([#1410](#))

- Regarding widget managers and the syncing message protocol:

  - The widget protocol was significantly overhauled. The new widget messaging protocol (version 2) is specified in the [version 2 protocol documentation](#).

  - Widgets are now displayed with a `display_data` message instead of with a custom comm message. See the [ipywidgets](#) implementation for an example. ([#1274](#))

  - Custom widget managers are now responsible completely for loading widget model and view classes. Widget managers should provide an output model and view class appropriate for their environment so that the `Output` widget works. ([#1313](#))

  - 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.

1.2 Developer Docs

The Jupyter widgets packages are developed in the https://github.com/jupyter-widgets/ipywidgets git repository. See the issue tracker, README, and other Github documents for the most recent information.

1.2.1 Scope of ipywidgets

`ipywidgets` is a framework to provide eventful python objects that have a representation in the browser (see examples/Widget Basics.ipynb for more on the definition of widgets). This requires two components:

1. The framework for widget interactions between the widgets represented in the Python kernel and the (javascript) representation of the widgets in the browser.

2. A basic, lightweight set of form controls that *use* this framework, based on standard HTML form controls. These included controls include a text area, text box, select and multiselect controls, checkbox, etc. A few more advanced controls that are very popular are also included, such as a slider and basic tab panels.

The framework for building rich interactive objects is the foremost purpose of the ipywidgets project, and the set of included reference form controls is purposefully kept small and self-contained to serve as something like a reference implementation. We encourage and support a robust ecosystem of packages built on top of the ipywidgets framework to provide more complicated interactive objects, such as maps or 2d and 3d visualizations, or other form control systems built on a variety of popular Javascript frameworks such as Material or Vue.

1.2.2 Additional Developer Docs

Developer Install

Prerequisites

To install ipywidgets from git, you will need:

- yarn package manager ** version 1.2.1 or later **
- 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
- the latest JupyterLab release

Installing With Conda

```
conda create -c conda-forge -n ipywidgets yarn notebook
conda activate ipywidgets
ipython kernel install --name ipywidgets --display-name "ipywidgets" --sys-prefix
pip install --pre jupyterlab
git clone https://github.com/jupyter-widgets/ipywidgets.git
cd ipywidgets
./dev-install.sh
```

Notice that we install the prerelease of JupyterLab currently, which is only available via `pip`.

Rebuilding after making changes

To build and test changes, run the following commands in the ipywidgets repository root directory, empty your browser's cache, and refresh the page.

```
    yarn run clean
    yarn run build
```

If your changes are confined to one package (for example, just in the widgetsnbextension package), then it may be sufficient to just run `yarn run build` in that specific directory.

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

Updating widget model specification

To update the widget model specification with changes, do something like this in the repo root:

```
python ./packages/schema/generate-spec.py > packages/schema/jupyterwidgetmodels.
↪latest.md
```

Releasing new versions

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

```
pytest --cov=ipywidgets ipywidgets
```

To run the Javascript tests in each package directory:

```
yarn 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.

Cleaning notebooks for docs

Notebook output and metadata should be stripped with nbstripoutput before commiting. For example:

```
nbstripoutput docs/source/examples/Widget\ List.ipynb
```

Developer Release Procedure

To release a new version of the widgets on PyPI and npm, first checkout master and cd into the repo root.

```
cd release
conda deactivate
conda remove --all -y -n releasewidgets
rm -rf ipywidgets

conda create -c conda-forge --override-channels -y -n releasewidgets notebook nodejs
→yarn twine jupyterlab=3 jupyter-packaging
conda activate releasewidgets

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

Fix the widget spec

If there were changes in the widget model specification (i.e., any change made to any widget attributes), we need to update the model specification version and record the documented attributes.

First, update the relevant model specification versions. For example, the commit https://github.com/jupyter-widgets/ipywidgets/commit/fca6f355605dc9e04062ce0eec4a7acbb5632ae2 updated the controls model version. We follow the semver spec for model version numbers, so model changes that are backwards-incompatible should be major version bumps, while backwards-compatible additions should be minor version bumps.

Next, regenerate the model spec with the new version numbers by doing something like this in the repository root directory:

```
python ./packages/schema/generate-spec.py > packages/schema/jupyterwidgetmodels.
→latest.md
```

Copy `packages/schema/jupyterwidgetmodels.latest.md` to an appropriately-named markdown file (see the existing model spec files in that directory for the naming convention). This documents the widget model specification for a specific ipywidget release.

Commit the changes (don't forget to `git add` the new model spec file).

Publish the npm modules

```
# clean out all dirty files
git checkout master
git pull origin master
git reset --hard origin/master
git clean -fdx
yarn install
yarn run publish
```

Lerna will prompt you for version numbers for each of the changed npm packages. Lerna will then change the versions appropriately (including the interdependency versions), commit, tag, and publish the new packages to npm.

jupyterlab_widgets

Go into the `jupyterlab_widgets` directory. Change `jupyterlab_widgets/_version.py` to reflect the new version number.

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

Verify that the package is uploaded.

```
curl -s https://pypi.org/pypi/jupyterlab-widgets/json | jq  -r '[.releases[][] | [.
→upload_time, .digests.sha256, .filename] | join(" ")] | sort '
```

widgetsnbextension

Go into the `widgetsnbextension` directory. Change `widgetsnbextension/_version.py` to reflect the new version number.

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

Verify that the package is uploaded.

```
curl -s https://pypi.org/pypi/widgetsnbextension/json | jq  -r '[.releases[][] | [.
→upload_time, .digests.sha256, .filename] | join(" ")] | sort '
```

ipywidgets

Change `ipywidgets/_version.py` to reflect the new version number, and if necessary, a new `__html_manager_version__`. Change the `install_requires` parameter in `setup.py` reference the new widgetsnbextension version.

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

Verify that the package is uploaded:

```
curl -s https://pypi.org/pypi/ipywidgets/json | jq  -r '[.releases[][] | [.upload_
↪time, .digests.sha256, .filename] | join(" ")] | sort '
```

Push changes back

Calculate the hashes of the uploaded files. You could use a small shell script, for example, like this on macOS:

```
#!/bin/sh
for f in $@
do
  echo "$f"
  echo md5: `md5 -q "$f"`
  echo sha1: `shasum -a 1 "$f" | awk '{print $1}'`
  echo sha256: `shasum -a 256 "$f" | awk '{print $1}'`
  echo
done
```

Using the above script, you can do:

```
hashes dist/*
hashes widgetsnbextension/dist/*
hashes jupyterlab_widgets/dist/*
```

Commit the changes you've made above, and include the uploaded files hashes in the commit message. Tag the release if ipywidgets was released. Push to origin master (and include the tag in the push).

Update conda-forge packages (if the requirements changed to ipywidgets, make sure to update widgetsnbextension first).

Release Notes

Here is an example of the release statistics for ipywidgets 7.0.

It has been 157 days since the last release. In this release, we closed 127 issues and 216 pull requests with 1069 commits, of which 851 are not merges.

Here are some commands used to generate some of the statistics above.

```
# merges since in 6.0.0, but not 7.0.0, which is a rough list of merged PRs
git log --merges 6.0.0...master --pretty=oneline

# To really make sure we get all PRs, we could write a program that
# pulled all of the PRs, examined a commit in each one, and did
```

(continues on next page)

```
# `git tag --contains <commit number>` to see if that PR commit is included
# in a previous release.

# issues closed with no milestone in the time period
# is:issue is:closed closed:"2016-07-14 .. 2017-02-28"

# date of 6.0.0 tag
git show -s --format=%cd --date=short 6.0.0^{commit}

# Non-merge commits in 7.0.0 not in any 6.x release
git log --pretty=oneline --no-merges ^6.0.0 master | wc -l

# Authors of non-merge commits
git shortlog -s  6.0.0..master --no-merges | cut -c8- | sort -f

# New committers: authors unique in the 6.0.0..7.0.0 logs, but not in the 6.0.0 log
comm -23 <(git shortlog -s -n 6.0.0..master --no-merges | cut -c8- | sort) <(git
→shortlog -s -n 6.0.0 --no-merges | cut -c8- | sort) | sort -f
```