# Widget Events

In this lecture we will discuss widget events, such as button clicks!

## Special events

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

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

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

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

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

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

### Example

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

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

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

button.on_click(on_button_clicked)
```

```
Button clicked.
Button clicked.
Button clicked.
Button clicked.
Button clicked.
Button clicked.
```

## on_submit

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

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

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

text.on_submit(handle_submit)
```

```
hello
press enter
```

## Traitlet events

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

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

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

        This is used to setup dynamic notifications of trait changes.

        Static handlers can be created by creating methods on a HasTraits
        subclass with the naming convention '_[traitname]_changed'.  Thus,
        to create static handler for the trait 'a', create the method
        _a_changed(self, name, old, new) (fewer arguments can be used, see
        below).

        Parameters
        ----------
        handler : callable
            A callable that is called when a trait changes.  Its
            signature can be handler(), handler(name), handler(name, new)
            or handler(name, old, new).
        name : list, str, None
            If None, 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.
        remove : bool
            If False (the default), then install the handler.  If True
            then unintall it.
```

## Signatures

Mentioned in the doc string, the callback registered can have 4 possible signatures:

- callback()
- callback(trait_name)
- callback(trait_name, new_value)
- callback(trait_name, old_value, new_value)

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

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

def on_value_change(name, value):
    print(value)

int_range.on_trait_change(on_value_change, 'value')
```

# Linking Widgets

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

## Linking traitlets attributes from the server side

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

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

In [17]:
```
# Create Caption
caption = widgets.Latex(value = 'The values of slider1 and slider2 are synchroniz

# Create IntSlider
slider1 = widgets.IntSlider(description='Slider 1')
slider2 =  widgets.IntSlider(description='Slider 2')

# Use trailets to link
l = traitlets.link((slider1, 'value'), (slider2, 'value'))

# Display!
display(caption, slider1, slider2)
```

In [16]:
```
# Create Caption
caption = widgets.Latex(value = 'Changes in source values are reflected in target

# Create Sliders
source = widgets.IntSlider(description='Source')
target1 = widgets.IntSlider(description='Target 1')

# Use dlink
dl = traitlets.dlink((source, 'value'), (target1, 'value'))
display(caption, source, target1)
```

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

In [18]:
```
# May get an error depending on order of cells being run!
l.unlink()
dl.unlink()
```

## Linking widgets attributes from the client side

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

In [19]:
```
# NO LAG VERSION
caption = widgets.Latex(value = 'The values of range1 and range2 are synchronized

range1 = widgets.IntSlider(description='Range 1')
range2 =  widgets.IntSlider(description='Range 2')

l = widgets.jslink((range1, 'value'), (range2, 'value'))
display(caption, range1, range2)
```

In [25]:
```
# NO LAG VERSION
caption = widgets.Latex(value = 'Changes in source_range values are reflected in

source_range = widgets.IntSlider(description='Source range')
target_range1 = widgets.IntSlider(description='Target range ')

dl = widgets.jsdlink((source_range, 'value'), (target_range1, 'value'))
display(caption, source_range, target_range1)
```

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

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

## Conclusion

You should now feel comfortable linking Widget events!