DATA SCIENCE IN THE REAL WORLD

Bring your Jupyter Notebook to life with interactive widgets

How to create dynamic dashboards using ipywidgets

Semi Koen May 3, 2019 · 10 min read★

Extending Jupyter's User Interface

Traditionally, every time you need to modify the output of your notebook cells, you need to change the code and rerun the affected cells. This can be cumbersome, inefficient and error prone and in the case of a non-technical user it may even be impracticable. This is where the ipywidgets come into play: they can be embedded in the notebook and provide a **user friendly** interface to collect the user input and see

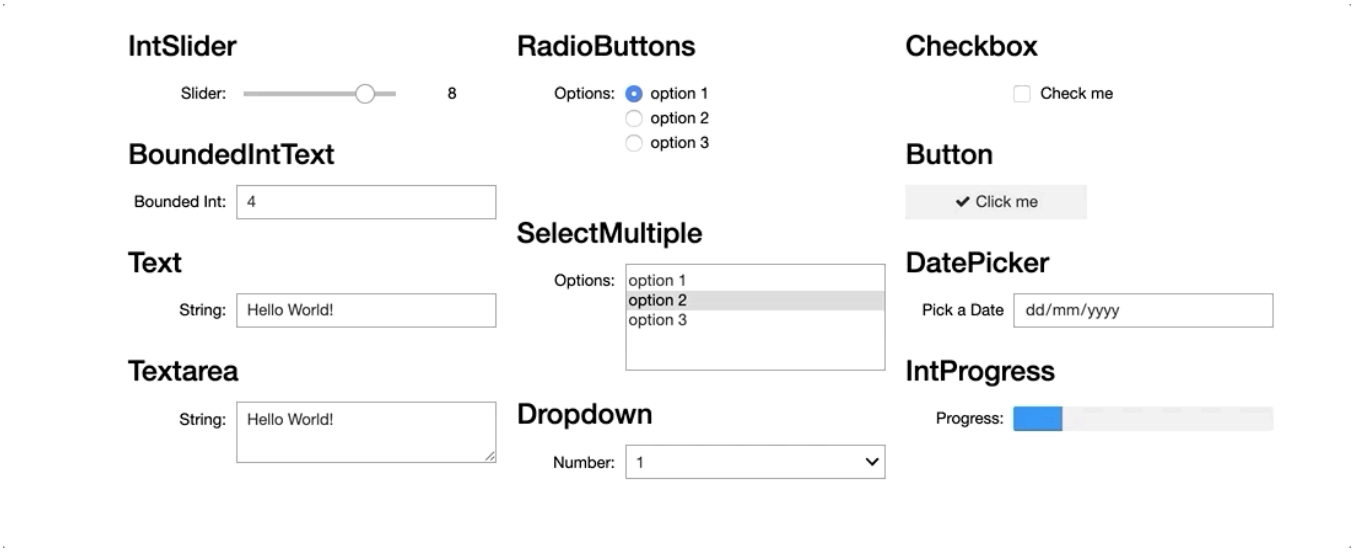
the impact the changes have on the data/results, without having to interact with the code; your notebooks can be transformed from static documents to dynamic dashboards — ideal for showcasing your data story!

> ⚠️ ***Scope:*** *There are limited resources on ipywidgets and the very few tutorials out there are either incomplete or focusing on the* `interact` *function/decorator. This is a* ***complete tutorial*** *of how you can take full control of the widgets to create powerful dashboards.*
>
> *We will start with the basics: adding a widget and explaining how the events work and we will progressively develop a dashboard.*
> *I will guide you step by step, building on the examples as we go.*

What is a widget?

If you have ever created a **graphical user interface** (GUI) then you already know what a widget is. But let's give a quick definition anyway:
A widget is a GUI element, such as a button, dropdown or textbox, which resides in the browser and allows us to control the code and the data by responding to events and invoking specified handlers.

These GUI elements can be assembled and customised to create complex dashboards.



Demo: A few of the most popular widgets

Throughout this article we will see some of them in action.

Ready? 🏁

❶ Getting started

To start using the library we need to install the **ipywidgets** extension. If using conda, we type this command in the terminal:

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

For pip, it will be a two-step process: 1. install and 2. enable:

```
pip install ipywidgets

jupyter nbextension enable --py widgetsnbextension
```

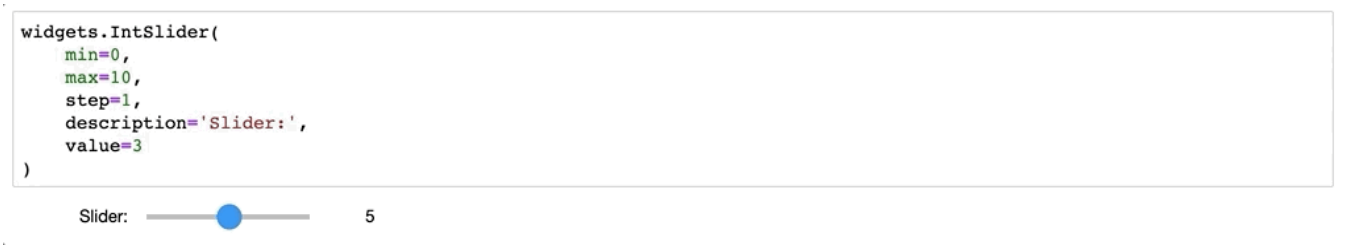
Adding a widget

In order to incorporate widgets in the notebook we have to import the module, as shown below:

```
import ipywidgets as widgets
```

To add a slider, we can define the minimum and maximum values, the interval size (step), a description and an initial value:

```
widgets.IntSlider(
    min=0,
    max=10,
    step=1,
    description='Slider:',
    value=3
)
```



Demo: Slider

Displaying it

The `display()` function renders a widget object in an input cell.

First import:

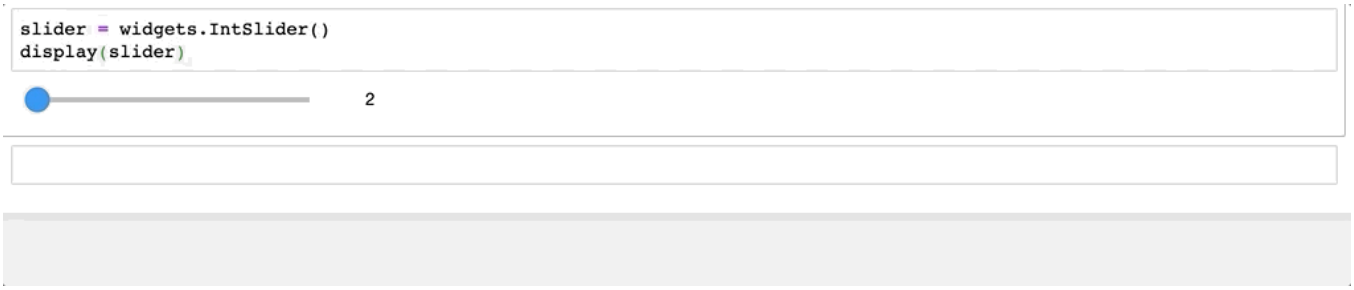
```
from IPython.display import display
```

Then pass the widget as a parameter in the display() function:

```
slider = widgets.IntSlider()
display(slider)
```

Getting/Setting its value

To read the value of a widget, we will query its `value` property. Similarly, we can set a widget's value:

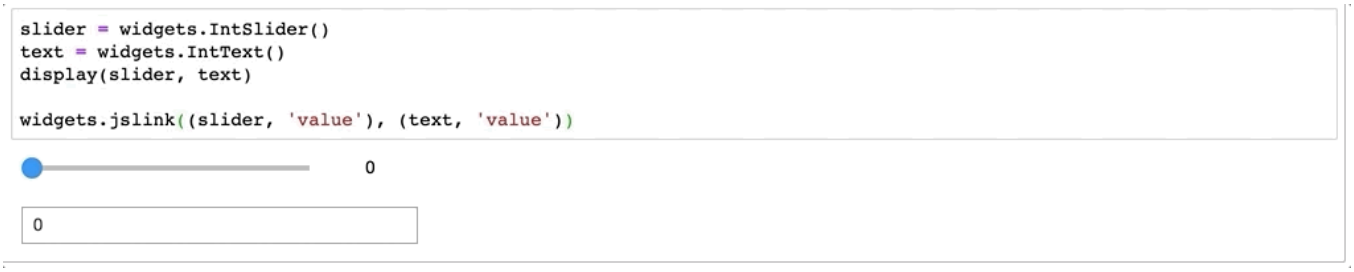


Demo: Value

Linking two widgets

We can synchronise the values of two widgets by using the `jslink()` function.

```
slider = widgets.IntSlider()
text = widgets.IntText()
display(slider, text)

widgets.jslink((slider, 'value'), (text, 'value'))
```



Demo: Linking

Widgets List

For a full list of widgets you can check out the documentation, or run the following command:

```
print(dir(widgets))
```

❷ Handling Widget Events

The widgets can respond to events, which are raised when a user interacts with them. A simple example is clicking on a button — we are expecting an action to take place.

Let's see how this works…

Depending on its specific features, each widget exposes different events. An **event handler** will be executed every time the event is fired.

> *Event handler is a callback function in response to an event, that operates asynchronously and handles the inputs received.*

Here we will create a simple button called `btn`. The `on_click` method is invoked when the button is clicked.

Our event handler, `btn_eventhandler`, will print a short message with the button's caption — note that the input argument of the handler, `obj`, is the button object itself which allows us to access its properties.

To bind the event with the handler, we assign the latter to the button's `on_click` method.

```
btn = widgets.Button(description='Medium')
display(btn)

def btn_eventhandler(obj):
    print('Hello from the {} button!'.format(obj.description))

btn.on_click(btn_eventhandler)
```

Demo: Button Event Handler

What will bring us nicely to the next section is that the output appears in the **same cell** as the button itself. So let's move on to see how we can add more flexibility to our notebook!

❸ Controlling Widget Output

In this section we will explore how to use widgets to **control a dataframe**. The sample dataset I have chosen is 'Number of International Visitors to London' which shows totals of London's visitors with regards to nights, visits and spend, broken down by year, quarter, purpose, duration, mode and country.

Initially, we will get the data and load it into a dataframe:

```
import pandas as pd
import numpy as np

url = "https://data.london.gov.uk/download/number-international-visitors-london/b1e0f953-4c8a-4b45-95f5-e0d143d5641e/international-visitors-london-raw.csv"

df_london = pd.read_csv(url)
```

	year	quarter	market	dur_stay	mode	purpose	area	visits	spend	nights	sample
44934	2015	Q1	Hong Kong	1-3 nights	Air	Holiday	LONDON	1.851748	1.801944	4.702943	3
16328	2006	Q4	Belgium	8-14 nights	Tunnel	Holiday	LONDON	0.463540	0.142171	6.026020	1
3548	2003	Q1	Greece	4-7 nights	Air	Holiday	LONDON	4.223560	3.193614	20.103650	9
36834	2012	Q4	Switzerland	8-14 nights	Tunnel	Business	LONDON	1.296450	1.301640	14.261000	1
44876	2015	Q1	Canada	4-7 nights	Air	Miscellaneous	LONDON	1.451995	0.405052	3.819025	3

df_london.sample(5)

Suppose we would like to **filter** the dataframe by *year*. We will first define a dropdown and populate it with the list of unique year values.

In order to do this, we will create a generic function, `unique_sorted_values_plus_ALL`, which will find the unique values, sort them and then add the `ALL` item at the start, so the user could remove the filter.

```
ALL = 'ALL'

def unique_sorted_values_plus_ALL(array):
    unique = array.unique().tolist()
    unique.sort()
    unique.insert(0, ALL)
    return unique
```

Now we will initialise the dropdown:

```
dropdown_year = widgets.Dropdown(options =
unique_sorted_values_plus_ALL(df_london.year))
```

The dropdown widget exposes the `observe` method, which takes a function that will be invoked when the value of the dropdown changes. As such, we will next create the observer handler to filter the dataframe by the selected values — note that the input argument of the handler, `change`, contains information about the changes that took place which allows us to access the `new` value (`change.new`).
If the new value is `ALL` we remove the filter, otherwise we apply it:

```
def dropdown_year_eventhandler(change):
    if (change.new == ALL):
        display(df_london)
    else:
        display(df_london[df_london.year == change.new])
```

We will then bind the handler to the dropdown:

```
dropdown_year.observe(dropdown_year_eventhandler, names='value')
```

```
ALL = 'ALL'
def unique_sorted_values_plus_ALL(array):
    unique = array.unique().tolist()
    unique.sort()
    unique.insert(0, ALL)
    return unique
```

```
dropdown_year = widgets.Dropdown(options = unique_sorted_values_plus_ALL(df_london.year))

def dropdown_year_eventhandler(change):
    if (change.new == ALL):
        display(df_london)
    else:
        display(df_london[df_london.year == change.new])

dropdown_year.observe(dropdown_year_eventhandler, names='value')

display(dropdown_year)
```

2018

	year	quarter	market	dur_stay	mode	purpose	area	visits	spend	nights	sample
54627	2018	Q1	Belgium	1-3 nights	Air	Holiday	LONDON	1.359576	0.522388	2.488439	2
54628	2018	Q1	Belgium	1-3 nights	Air	Business	LONDON	2.370728	0.443170	3.233408	4
54629	2018	Q1	Belgium	1-3 nights	Air	Miscellaneous	LONDON	1.519122	0.538684	2.449016	2
54630	2018	Q1	Belgium	1-3 nights	Sea	Holiday	LONDON	2.627515	0.427922	3.938132	2
54631	2018	Q1	Belgium	1-3 nights	Sea	Miscellaneous	LONDON	1.389915	0.333465	1.369789	1
54632	2018	Q1	Belgium	1-3 nights	Tunnel	Holiday	LONDON	52.457723	32.697137	118.259651	32

Using a dropdown to filter a dataframe

So far so good, but the output of all the queries is *accumulating* in this very same cell; i.e. if we select a new year from the dropdown, a new dataframe will render underneath the first one, on the same cell.
The desired behaviour though, is to **refresh** the contents of the dataframe each time.

Capturing Widget Output

The solution to this is to capture the cell output in a special kind of widget, namely `Output`, and then display it in another cell.

We will slightly tweak the code to:

- create a new instance of `Output`

```
output_year = widgets.Output()
```

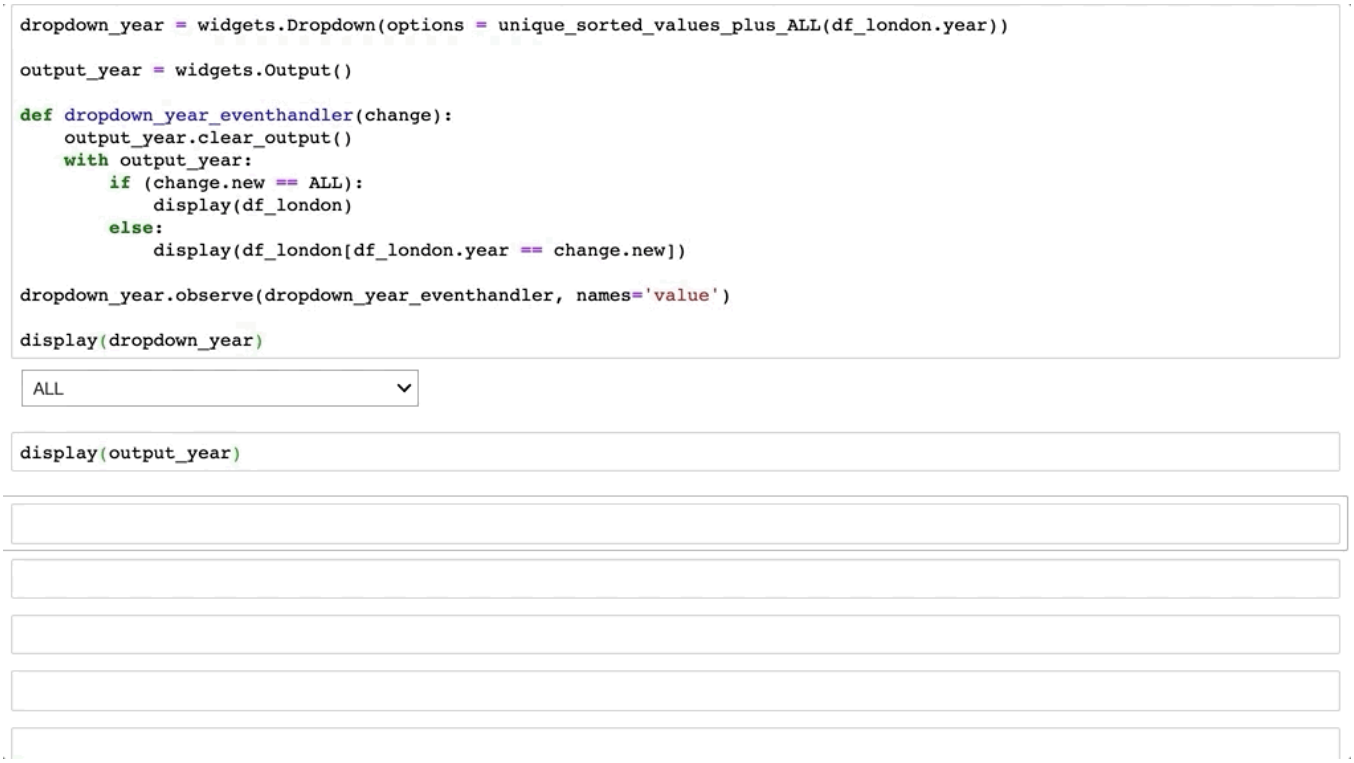
- call the `clear_output` method within the event handler to clear the previous selection on each iteration, and capture the output of the dataframe in a `with` block.

```
def dropdown_year_eventhandler(change):
    output_year.clear_output()
    with output_year:
        display(df_london[df_london.year == change.new])
```

We will then display the output in a new cell:

```
display(output_year)
```

This is how it works:



Demo: Capturing output in a new cell

As you can see the output is rendered in a new cell and the filtering is working as expected! 👍

❹ Linking Widget Outputs

Continuing the previous example, let's assume we would also like to filter by *purpose* too.

If we go ahead and add another dropdown, we will quickly realise the dataframe only responds to the filter by the dropdown which was recently changed. What we need to do is to **link** the two together so it can work on both values (i.e. year and purpose).

Let's see how it should work:

Firstly we need a common output for both dropdowns:

```
output = widgets.Output()
```

Here are the two dropdowns:

```
dropdown_year = widgets.Dropdown(options =
unique_sorted_values_plus_ALL(df_london.year))

dropdown_purpose = widgets.Dropdown(options =
unique_sorted_values_plus_ALL(df_london.purpose))
```

Then we create a new function, `common_filtering`, that will be called by both the event handlers. This function will apply a filter on the dataframe for both year *AND* purpose:
We are clearing the output, then we check if any of the values is `ALL`, in which case we consider that the respective filter is removed. When both filters are present, in the `else` statement, we apply the `&` operation in both filters. Finally we capture the output:

```
def common_filtering(year, purpose):
    output.clear_output()

    if (year == ALL) & (purpose == ALL):
        common_filter = df_london
    elif (year == ALL):
        common_filter = df_london[df_london.purpose == purpose]
```

```
    elif (purpose == ALL):
        common_filter = df_london[df_london.year == year]
    else:
        common_filter = df_london[(df_london.year == year) &
                                  (df_london.purpose == purpose)]

    with output:
        display(common_filter)
```

We amend the event handlers to call the `common_filtering` function and pass the `change.new` value as well as the current `value` of the other dropdown:

```
def dropdown_year_eventhandler(change):
    common_filtering(change.new, dropdown_purpose.value)

def dropdown_purpose_eventhandler(change):
    common_filtering(dropdown_year.value, change.new)
```

We bind the handlers to the dropdowns, and that's it!

```
dropdown_year.observe(
dropdown_year_eventhandler, names='value')

dropdown_purpose.observe(
dropdown_purpose_eventhandler, names='value')
```

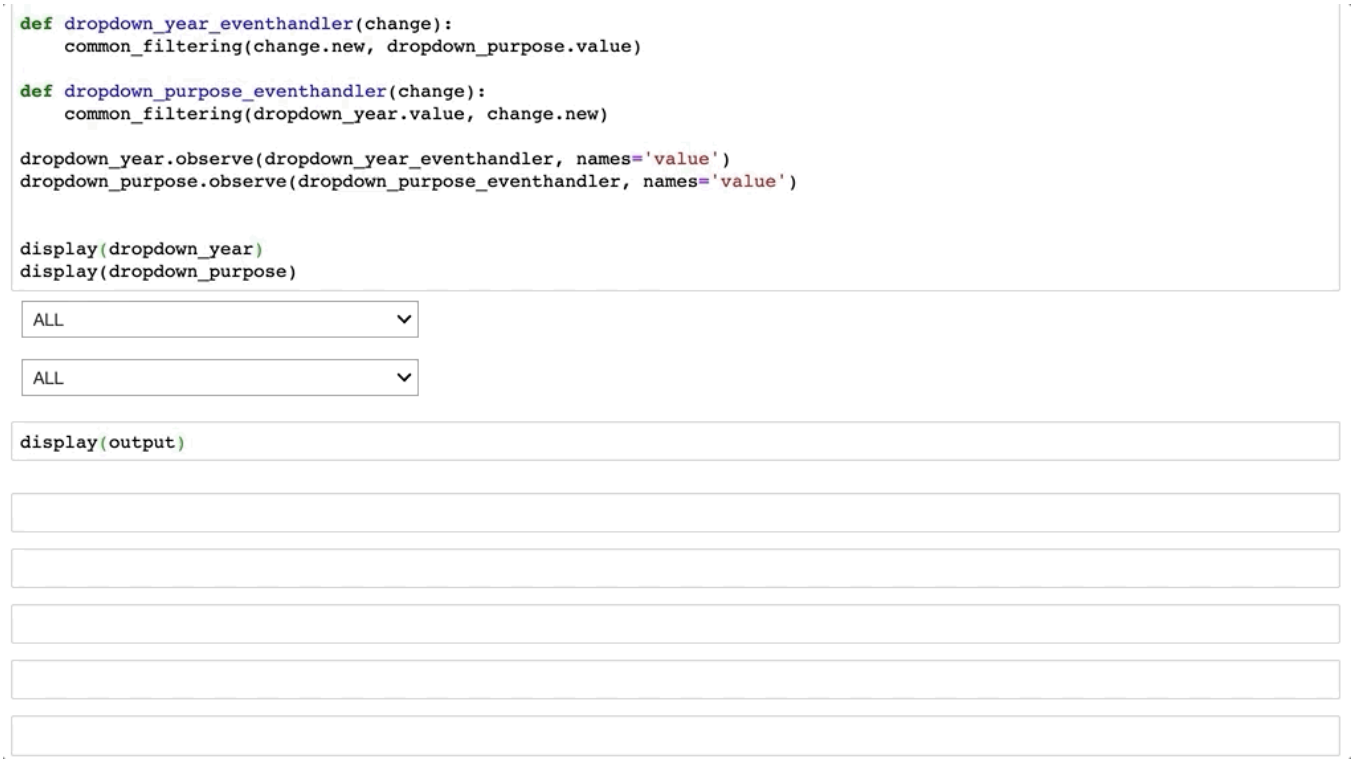
Code snippet:

```
output = widgets.Output()

dropdown_year = widgets.Dropdown(options = unique_sorted_values_plus_ALL(df_london.year))
dropdown_purpose = widgets.Dropdown(options = unique_sorted_values_plus_ALL(df_london.purpose))

def common_filtering(year, purpose):
    output.clear_output()

    if (year == ALL) & (purpose == ALL):
        common_filter = df_london
    elif (year == ALL):
        common_filter = df_london[df_london.purpose == purpose]
    elif (purpose == ALL):
        common_filter = df_london[df_london.year == year]
    else:
        common_filter = df_london[(df_london.year == year) & (df_london.purpose == purpose)]

    with output:
        display(common_filter)


def dropdown_year_eventhandler(change):
```

```
    common_filtering(change.new, dropdown_purpose.value)

def dropdown_purpose_eventhandler(change):
    common_filtering(dropdown_year.value, change.new)

dropdown_year.observe(dropdown_year_eventhandler, names='value')
dropdown_purpose.observe(dropdown_purpose_eventhandler, names='value')


display(dropdown_year)
display(dropdown_purpose)
```

Filter a dataframe based on two values

Here is the demo:



Demo: Filter a dataframe based on two values

❺ Creating a Dashboard

We have put the basis for our dashboard so far by filtering and displaying the data of the London dataset. We will carry on by colouring the numeric values based on a user selected value.

A useful numeric widget is the `BoundedFloatText`; we will give it a `min`, `max` and initial `value`, and the incremental `step`.

```
bounded_num = widgets.BoundedFloatText(
min=0, max=100000, value=5, step=1)
```

In order to colour the dataframe cells, we will define this function:

```
def colour_ge_value(value, comparison):
    if value >= comparison:
        return 'color: red'
    else:
        return 'color: black'
```

Now we will minimally amend the `common_filtering` function to:

- add new `num` input parameter:

```
def common_filtering(year, purpose, num):
```

- apply the styling by calling the `colour_ge_value` function for the three numeric columns:

```
with output:
        display(common_filter
                .style.applymap(
                    lambda x: colour_ge_value(x, num),
                    subset=['visits','spend', 'nights']))
```

The existing event handlers need to be adjusted to pass the `bounded_num.value`:

```
def dropdown_year_eventhandler(change):
    common_filtering(change.new, dropdown_purpose.value,
                     bounded_num.value)

def dropdown_purpose_eventhandler(change):
    common_filtering(dropdown_year.value, change.new,
                     bounded_num.value)
```

And finally we will plug-in the event handler of the new widget:

```
def bounded_num_eventhandler(change):
    common_filtering(dropdown_year.value, dropdown_purpose.value,
                     change.new)

bounded_num.observe(bounded_num_eventhandler, names='value')
```

Code snippet:

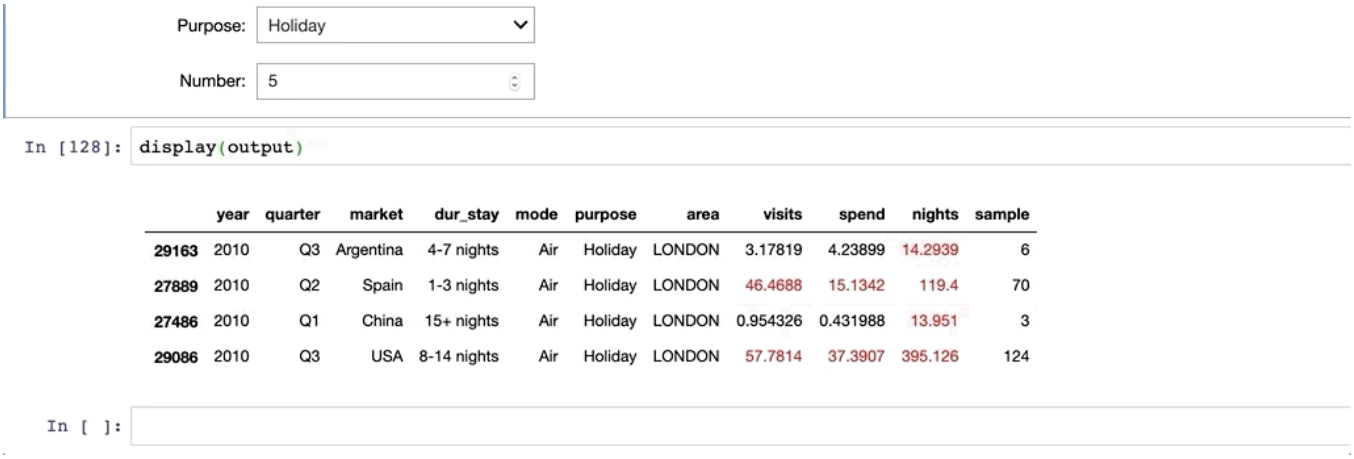
```
output = widgets.Output()

dropdown_year = widgets.Dropdown(options = unique_sorted_values_plus_ALL(df_london.year), description='Year:')
dropdown_purpose = widgets.Dropdown(options = unique_sorted_values_plus_ALL(df_london.purpose), description='Purpose:')
bounded_num = widgets.BoundedFloatText(min=0, max=100000, value=5, step=1, description='Number:')

def common_filtering(year, purpose, num):
    output.clear_output()

    if (year == ALL) & (purpose == ALL):
        common_filter = df_london
    elif (year == ALL):
        common_filter = df_london[df_london.purpose == purpose]
    elif (purpose == ALL):
        common_filter = df_london[df_london.year == year]
    else:
        common_filter = df_london[(df_london.year == year) & (df_london.purpose == purpose)]

    with output:
        display(common_filter.style.applymap(lambda x: colour_ge_value(x, num), subset=['visits','spend', 'nights']))

def dropdown_year_eventhandler(change):
    common_filtering(change.new, dropdown_purpose.value,bounded_num.value)

def dropdown_purpose_eventhandler(change):
    common_filtering(dropdown_year.value, change.new, bounded_num.value)

def bounded_num_eventhandler(change):
    common_filtering(dropdown_year.value, dropdown_purpose.value, change.new)

dropdown_year.observe(dropdown_year_eventhandler, names='value')
dropdown_purpose.observe(dropdown_purpose_eventhandler, names='value')
bounded_num.observe(bounded_num_eventhandler, names='value')

display(dropdown_year)
display(dropdown_purpose)
display(bounded_num)
```

Colour dataframe values

Here is the demo:

```
dropdown_year.observe(dropdown_year_eventhandler, names='value')
dropdown_purpose.observe(dropdown_purpose_eventhandler, names='value')
bounded_num.observe(bounded_num_eventhandler, names='value')

display(dropdown_year)
display(dropdown_purpose)
display(bounded_num)
```

Year: 2010



	year	quarter	market	dur_stay	mode	purpose	area	visits	spend	nights	sample
29163	2010	Q3	Argentina	4-7 nights	Air	Holiday	LONDON	3.17819	4.23899	14.2939	6
27889	2010	Q2	Spain	1-3 nights	Air	Holiday	LONDON	46.4688	15.1342	119.4	70
27486	2010	Q1	China	15+ nights	Air	Holiday	LONDON	0.954326	0.431988	13.951	3
29086	2010	Q3	USA	8-14 nights	Air	Holiday	LONDON	57.7814	37.3907	395.126	124

Demo: Colour dataframe values

Plotting

Next we will be adding a new graph to plot a basic univariate density of the number of visits (KDE → Kernel Density Estimation). We will use seaborn, so let's import the libraries:

```
import seaborn as sns
import matplotlib.pyplot as plt
```

Continuing the previous use-case, we will capture the plot in a new output variable:

```
plot_output = widgets.Output()
```

We will now amend the `common_filtering` function to plot the new diagram:

- first we clear the output:

```
plot_output.clear_output()
```

- and then we call the `kdeplot` method of seaborn by passing the number of visits:

```
with plot_output:
    sns.kdeplot(common_filter['visits'], shade=True)
```

```
        plt.show()
```

Lastly, the only thing we need to do is to display the outputs in a new cell:

```
display(output)
display(plot_output)
```

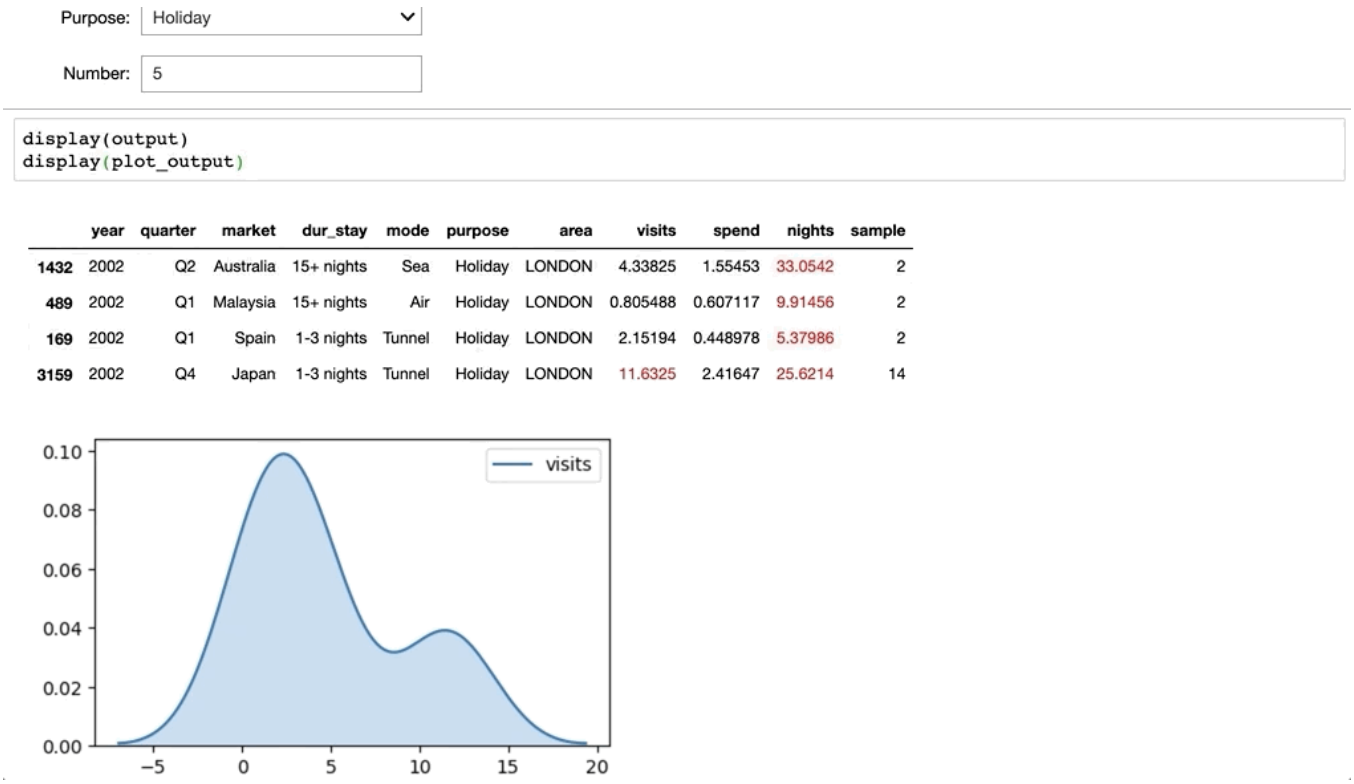
Code snippet:

```
output = widgets.Output()
plot_output = widgets.Output()

dropdown_year = widgets.Dropdown(options = unique_sorted_values_plus_ALL(df_london.year), description='Year:')
dropdown_purpose = widgets.Dropdown(options = unique_sorted_values_plus_ALL(df_london.purpose), description='Purpose:')
bounded_num = widgets.BoundedFloatText(min=0, max=100000, value=5, step=1, description='Number:')

def common_filtering(year, purpose, num):
    output.clear_output()
    plot_output.clear_output()

    if (year == ALL) & (purpose == ALL):
        common_filter = df_london
    elif (year == ALL):
        common_filter = df_london[df_london.purpose == purpose]
    elif (purpose == ALL):
        common_filter = df_london[df_london.year == year]
    else:
        common_filter = df_london[(df_london.year == year) & (df_london.purpose == purpose)]

    with output:
        display(common_filter.style.applymap(lambda x: colour_ge_value(x, num), subset=['visits','spend', 'nights']))

    with plot_output:
        sns.kdeplot(common_filter['visits'], shade=True)
        plt.show()

def dropdown_year_eventhandler(change):
    common_filtering(change.new, dropdown_purpose.value,bounded_num.value)

def dropdown_purpose_eventhandler(change):
    common_filtering(dropdown_year.value, change.new, bounded_num.value)

def bounded_num_eventhandler(change):
    common_filtering(dropdown_year.value, dropdown_purpose.value, change.new)

dropdown_year.observe(dropdown_year_eventhandler, names='value')
dropdown_purpose.observe(dropdown_purpose_eventhandler, names='value')
bounded_num.observe(bounded_num_eventhandler, names='value')

display(dropdown_year)
display(dropdown_purpose)
display(bounded_num)
```

Controlling a graph

Here is the demo:

```
display(dropdown_year)
display(dropdown_purpose)
display(bounded_num)
```



Demo: Controlling a graph

❻ Dashboard Layout

Up until now our user interface is functional but is taking up a lot of real estate.

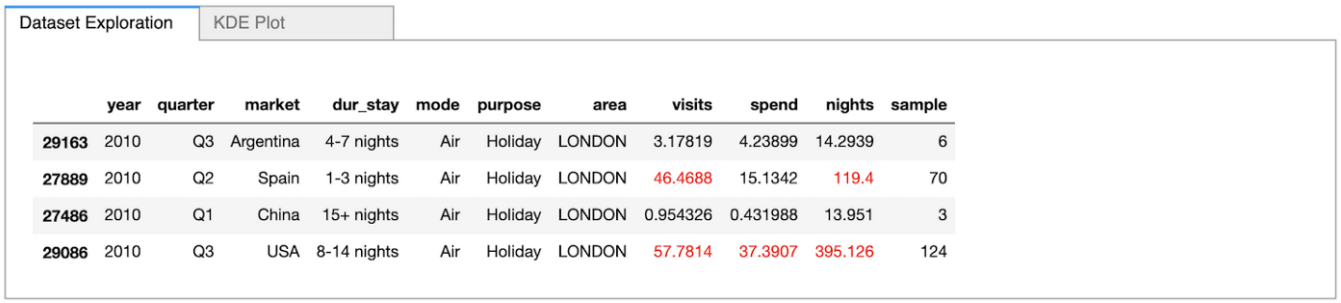
We will first **arrange** the input widgets horizontally. The `HBox` will add widgets to it one at a time from left-to-right:

```
input_widgets = widgets.HBox(
[dropdown_year, dropdown_purpose, bounded_num])

display(input_widgets)
```

HBox

Next we will create a **container** for the output. `Tab` is great for this. The 1st tab will host the dataframe and the 2nd one the graph.

```
tab = widgets.Tab([output, plot_output])
tab.set_title(0, 'Dataset Exploration')
tab.set_title(1, 'KDE Plot')

display(tab)
```

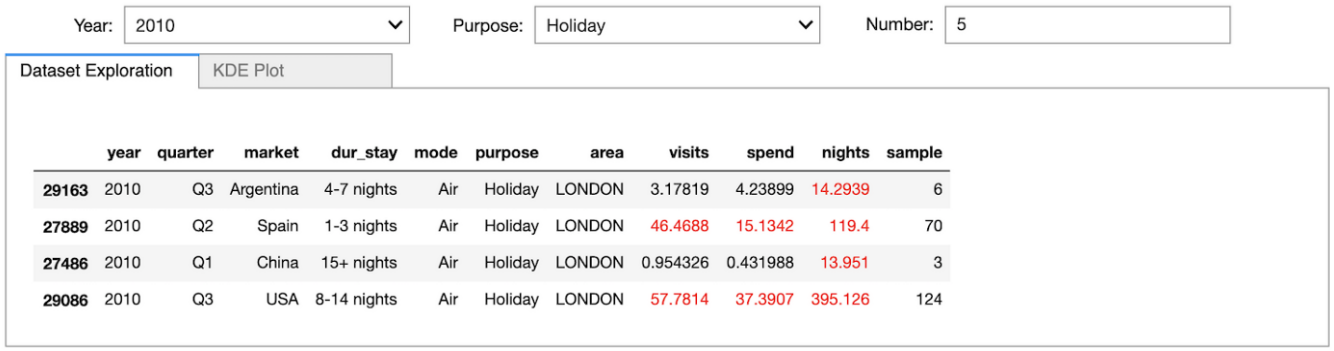


	year	quarter	market	dur_stay	mode	purpose	area	visits	spend	nights	sample
29163	2010	Q3	Argentina	4-7 nights	Air	Holiday	LONDON	3.17819	4.23899	14.2939	6
27889	2010	Q2	Spain	1-3 nights	Air	Holiday	LONDON	46.4688	15.1342	119.4	70
27486	2010	Q1	China	15+ nights	Air	Holiday	LONDON	0.954326	0.431988	13.951	3
29086	2010	Q3	USA	8-14 nights	Air	Holiday	LONDON	57.7814	37.3907	395.126	124

Tab

Finally we will stack the input widgets and the tab on top of each other with a `VBox`.

```
dashboard = widgets.VBox([input_widgets, tab])

display(dashboard)
```



	year	quarter	market	dur_stay	mode	purpose	area	visits	spend	nights	sample
29163	2010	Q3	Argentina	4-7 nights	Air	Holiday	LONDON	3.17819	4.23899	14.2939	6
27889	2010	Q2	Spain	1-3 nights	Air	Holiday	LONDON	46.4688	15.1342	119.4	70
27486	2010	Q1	China	15+ nights	Air	Holiday	LONDON	0.954326	0.431988	13.951	3
29086	2010	Q3	USA	8-14 nights	Air	Holiday	LONDON	57.7814	37.3907	395.126	124

VBox

It feels a bit 'jammed', so as a last step, we will **polish** our dashboard by adding some space. We will define a `Layout` giving 50px margin between the items.

```
item_layout = widgets.Layout(margin='0 0 50px 0')
```

We will call this layout for each item:

```
input_widgets = widgets.HBox(
[dropdown_year, dropdown_purpose, bounded_num],
layout=item_layout)

tab = widgets.Tab([output, plot_output],
layout=item_layout)
```

and ta da…. Our finished dashboard:

```
item_layout = widgets.Layout(margin='0 0 50px 0')
```

```
input_widgets = widgets.HBox([dropdown_year, dropdown_purpose, bounded_num],
                             layout=item_layout)
```

```
tab = widgets.Tab([output, plot_output], layout=item_layout)
tab.set_title(0, 'Dataset Exploration')
tab.set_title(1, 'KDE Plot')
```

```
dashboard = widgets.VBox([input_widgets, tab])
display(dashboard)
```

Year: 2010 Purpose: Holiday Number: 5

Dataset Exploration | KDE Plot

	year	quarter	market	dur_stay	mode	purpose	area	visits	spend	nights	sample
29163	2010	Q3	Argentina	4-7 nights	Air	Holiday	LONDON	3.17819	4.23899	14.2939	6
27889	2010	Q2	Spain	1-3 nights	Air	Holiday	LONDON	46.4688	15.1342	119.4	70
27486	2010	Q1	China	15+ nights	Air	Holiday	LONDON	0.954326	0.431988	13.951	3
29086	2010	Q3	USA	8-14 nights	Air	Holiday	LONDON	57.7814	37.3907	395.126	124

Dashboard

Final demo

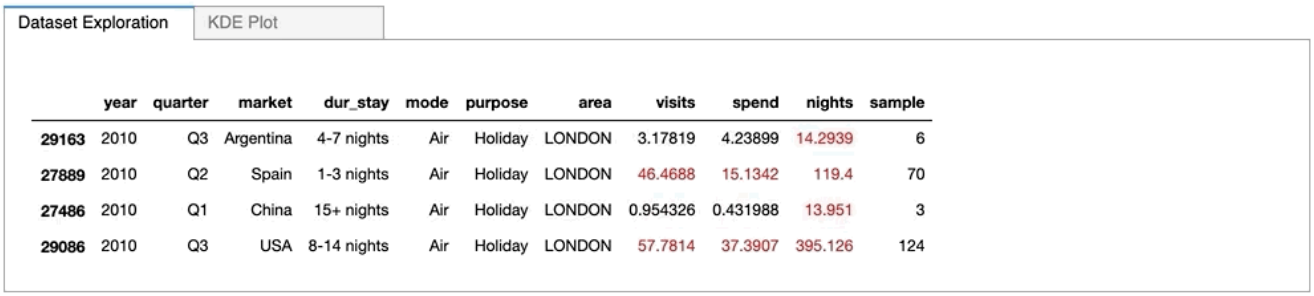

```
item_layout = widgets.Layout(margin='0 0 50px 0')
```

```
input_widgets = widgets.HBox([dropdown_year, dropdown_purpose, bounded_num],
                             layout=item_layout)
```

```
tab = widgets.Tab([output, plot_output], layout=item_layout)
tab.set_title(0, 'Dataset Exploration')
tab.set_title(1, 'KDE Plot')
```

```
dashboard = widgets.VBox([input_widgets, tab])
display(dashboard)
```

Year: 2010 Purpose: Holiday Number: 5



	year	quarter	market	dur_stay	mode	purpose	area	visits	spend	nights	sample
29163	2010	Q3	Argentina	4-7 nights	Air	Holiday	LONDON	3.17819	4.23899	14.2939	6
27889	2010	Q2	Spain	1-3 nights	Air	Holiday	LONDON	46.4688	15.1342	119.4	70
27486	2010	Q1	China	15+ nights	Air	Holiday	LONDON	0.954326	0.431988	13.951	3
29086	2010	Q3	USA	8-14 nights	Air	Holiday	LONDON	57.7814	37.3907	395.126	124

Demo: Final Dashboard

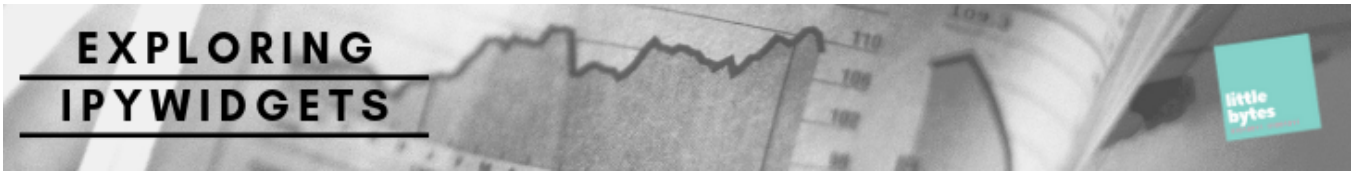
> ***PS**: For presentation purposes, in some of these demos I have used a subset of the dataset i.e.:* `df_london = df_london.sample(250)`.

Go further

There are a few third party widgets you can use too, with most popular ones being:

- 2-D charting: bqplot
- 3-D visualisation: pythreejs and ipyvolume
- Mapping: ipyleaflet and gmaps.

You can also build your own custom widgets! For more info take a look here.



Recap

We saw a fairly wide range of widgets in action but we still have only scratched the surface here — we can build really complex and extensive GUIs using ipywidgets. I hope you all agree they deserve a place in any Data Scientist's toolbox as they enhance our productivity and add a lot of value during data exploration.

Thanks for reading!