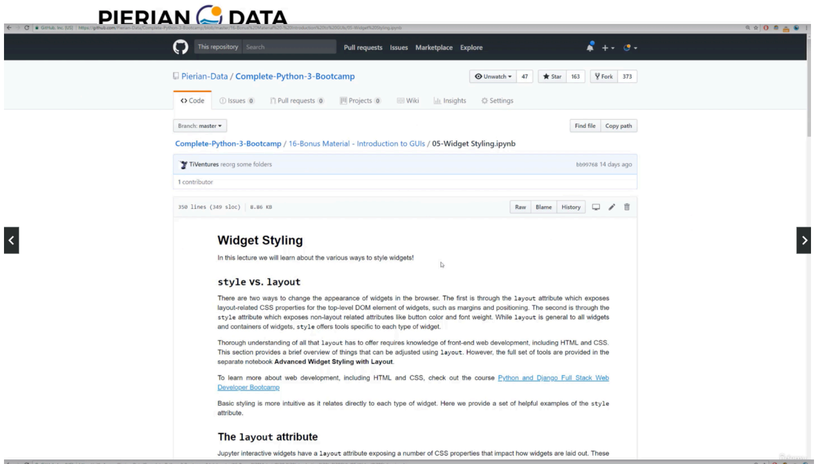
SECTION 21: GUIs - 45 minutes, 7 parts
6/7 Widget Styling and Layouts

- -> stying widgets
- -> using the style vs layout functionality of the ipywidget system
- **-> information in the notebook**
  - -> style vs layout
  - -> this assumes knowledge of CSS and HTML <- flexbox et al
  - -> django

Widget Styling and Layout

PIERIAN DATA

- **-> he first imports modules**
  - -> ipywidgets
  - -> IPython.display
- -> then creates an integer slider and displays it
- -> we first see a default
- -> we can change the size of the slider as default
- -> if you have ....., you can enter tab and see the different module options available



Pierian-Data / Complete-Python-3-Bootcamp

Branch: master

Complete-Python-3-Bootcamp / 16-Bonus Material - Introduction to GUIs / 05-Widget Styling.ipynb

TiVentures reorg some folders — bb99768 14 days ago

1 contributor

350 lines (349 sloc) | 8.86 KB

Widget Styling

In this lecture we will learn about the various ways to style widgets!

`style` vs. `layout`

There are two ways to change the appearance of widgets in the browser. The first is through the `layout` attribute which exposes layout-related CSS properties for the top-level DOM element of widgets, such as margins and positioning. The second is through the `style` attribute which exposes non-layout related attributes like button color and font weight. While `layout` is general to all widgets and containers of widgets, `style` offers tools specific to each type of widget.

Thorough understanding of all that `layout` has to offer requires knowledge of front-end web development, including HTML and CSS. This section provides a brief overview of things that can be adjusted using `layout`. However, the full set of tools are provided in the separate notebook **Advanced Widget Styling with Layout**.

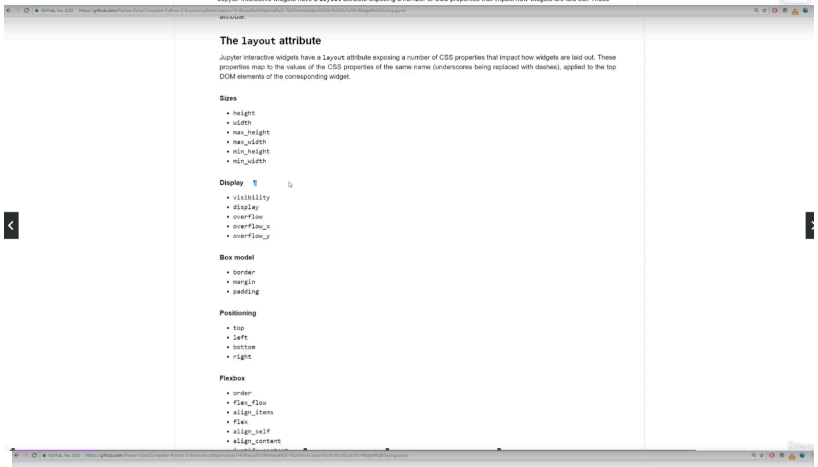
To learn more about web development, including HTML and CSS, check out the course Python and Django Full Stack Web Developer Bootcamp

Basic styling is more intuitive as it relates directly to each type of widget. Here we provide a set of helpful examples of the `style` attribute.

The `layout` attribute

Jupyter interactive widgets have a `layout` attribute exposing a number of CSS properties that impact how widgets are laid out. These

- **-> the height of the slider can be adjusted**
  - -> as can the margin
  - -> passed from one widget to another of the same type



The `layout` attribute

Jupyter interactive widgets have a `layout` attribute exposing a number of CSS properties that impact how widgets are laid out. These properties map to the values of the CSS properties of the same name (underscores being replaced with dashes), applied to the top DOM elements of the corresponding widget.

Sizes

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

Display

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

Box model

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

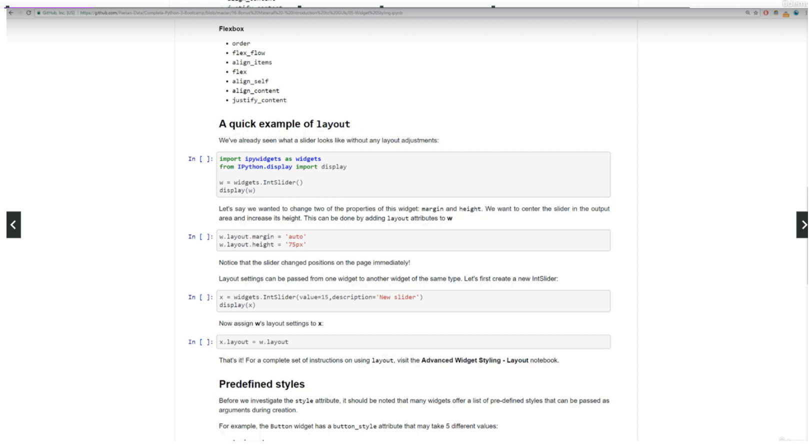
Positioning

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

Flexbox

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

- **-> he enters arguments into the new slider**
  - -> this has default styling
  - -> this can be changed
  - -> he displays the slider using display()
  - -> using x.layout
  - -> he then centres the slider



Flexbox

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

A quick example of `layout`

We've already seen what a slider looks like without any layout adjustments:

```
In [ ]: import ipywidgets as widgets
        from IPython.display import display

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

Let's say we wanted to change two of the properties of this widget: `margin` and `height`. We want to center the slider in the output area and increase its height. This can be done by adding `layout` attributes to **w**

```
In [ ]: w.layout.margin = 'auto'
        w.layout.height = '75px'
```

Notice that the slider changed positions on the page immediately!

Layout settings can be passed from one widget to another widget of the same type. Let's first create a new IntSlider:

```
In [ ]: x = widgets.IntSlider(value=15,description='New slider')
        display(x)
```

Now assign **w**'s layout settings to **x**:

```
In [ ]: x.layout = w.layout
```

That's it! For a complete set of instructions on using `layout`, visit the **Advanced Widget Styling - Layout** notebook.

Predefined styles

Before we investigate the `style` attribute, it should be noted that many widgets offer a list of pre-defined styles that can be passed as arguments during creation.

For example, the `Button` widget has a `button_style` attribute that may take 5 different values:

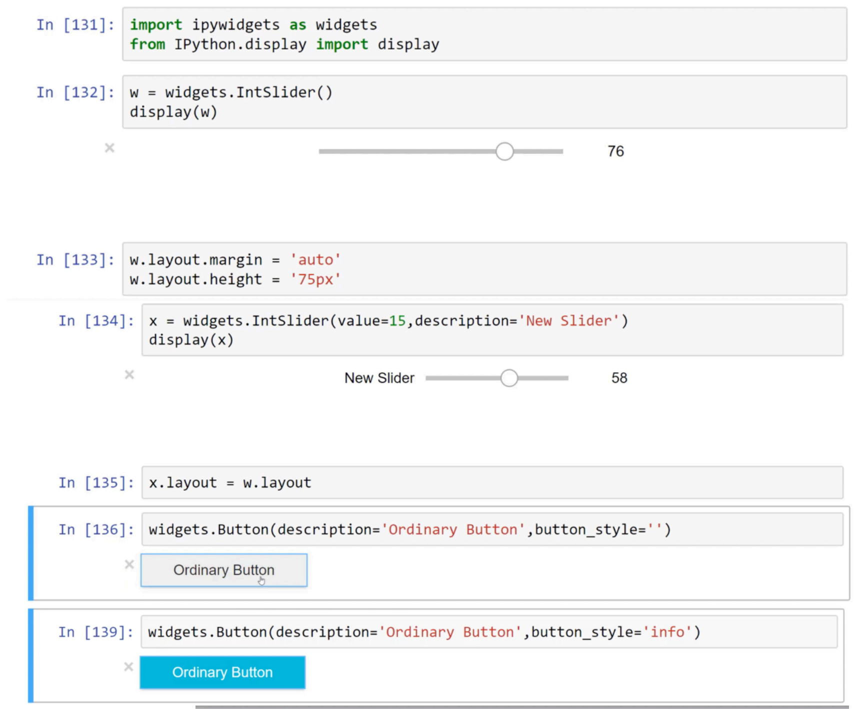
- **-> many widgets have pre-defined styles**
  - -> these are defined on bootstrap terms
  - -> e.g primary, success, info, warning and danger
- **-> using widgets.Button in this example**
  - -> this creates an ordinary button
  - -> we are using Python to create buttons
  - -> these look like ones which could be written in HTML and CSS, but there are functions which you can use to get them in HTML
- -> you can write other arguments into this
- **-> the style attribute**
  - -> this is used to expose non-layout related styling attributes

    - -> custom colours
    - -> he calls the .style method on a variable which stores a button
    - -> this button looks like something you could write in HTML or CSS
    - -> he calls the button colour method
    - -> there are multiple different style attributes available for this
    - -> using the .keys method, we can also see the different available methods for this

- **-> he defines a second button**
    - -> he sets the style of the second button equal to that for the first
    - -> this returns the same button as the first - but without a description (until he gives one)

- **-> he then creates another slider**
    - -> this has a description
    - -> there are multiple potential options for this
    - -> this can be assigned a different colour depending on what we select
    - -> button colour and font weight
    - -> progress bars
    - -> there are multiple others that only have description width as adjustable weights
    - -> Jupyter has to be able to colour the button and not only affect everything happening in the browser
    - -> we don't have that much access to all of the styling
    - -> Jupyter has to colour one thing and not the rest of the elements in the browser

- -> then how to create a custom widget
- -> all of the widgets and the functionality associated with this



```
In [131]: import ipywidgets as widgets
          from IPython.display import display

In [132]: w = widgets.IntSlider()
          display(w)
```

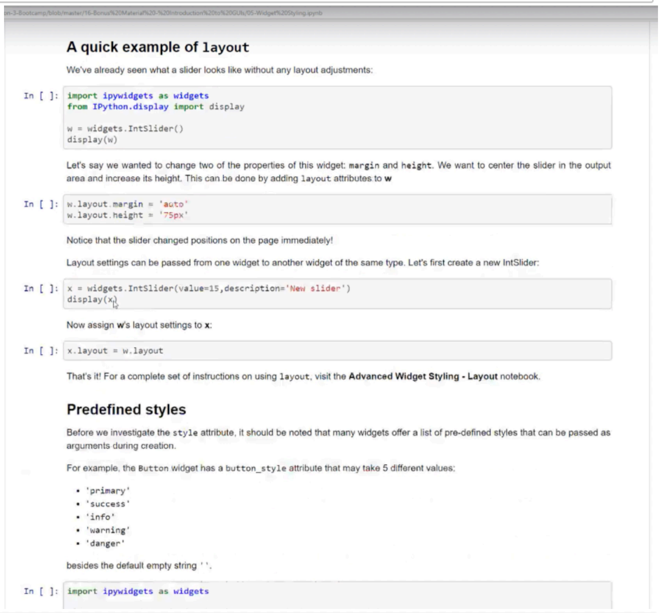
```
In [133]: w.layout.margin = 'auto'
          w.layout.height = '75px'

In [134]: x = widgets.IntSlider(value=15,description='New Slider')
          display(x)
```

```
In [135]: x.layout = w.layout

In [136]: widgets.Button(description='Ordinary Button',button_style='')
```

```
In [139]: widgets.Button(description='Ordinary Button',button_style='info')
```



on-3-Bootcamp/blob/master/16-Bonus%20Material%20-%20Introduction%20to%20GUIs/05-Widget%20Styling.ipynb

A quick example of `layout`

We've already seen what a slider looks like without any layout adjustments:

```
In [ ]: import ipywidgets as widgets
        from IPython.display import display

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

Let's say we wanted to change two of the properties of this widget: `margin` and `height`. We want to center the slider in the output area and increase its height. This can be done by adding `layout` attributes to **w**

```
In [ ]: w.layout.margin = 'auto'
        w.layout.height = '75px'
```

Notice that the slider changed positions on the page immediately!

Layout settings can be passed from one widget to another widget of the same type. Let's first create a new IntSlider:

```
In [ ]: x = widgets.IntSlider(value=15,description='New slider')
        display(x)
```

Now assign **w**'s layout settings to **x**:

```
In [ ]: x.layout = w.layout
```

That's it! For a complete set of instructions on using `layout`, visit the **Advanced Widget Styling - Layout** notebook.

Predefined styles

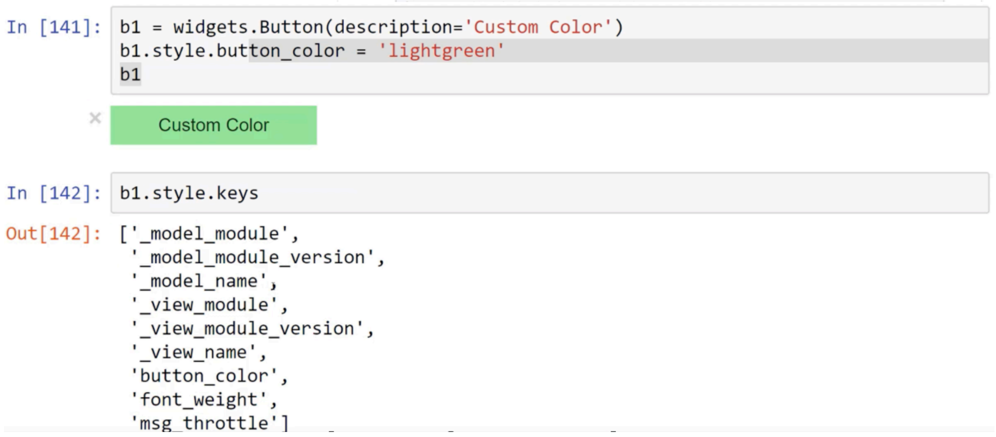
Before we investigate the `style` attribute, it should be noted that many widgets offer a list of pre-defined styles that can be passed as arguments during creation.

For example, the `Button` widget has a `button_style` attribute that may take 5 different values:

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

besides the default empty string `''`.

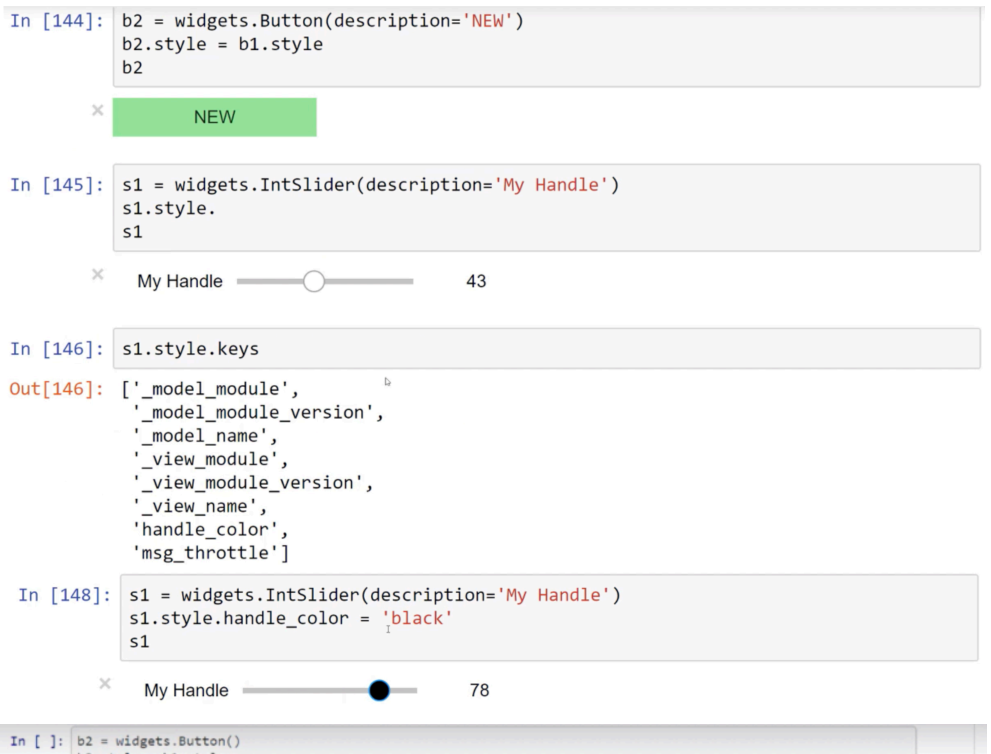
```
In [ ]: import ipywidgets as widgets
```



```
In [141]: b1 = widgets.Button(description='Custom Color')
          b1.style.button_color = 'lightgreen'
          b1
```

```
In [142]: b1.style.keys

Out[142]: ['_model_module',
           '_model_module_version',
           '_model_name',
           '_view_module',
           '_view_module_version',
           '_view_name',
           'button_color',
           'font_weight',
           'msg_throttle']
```



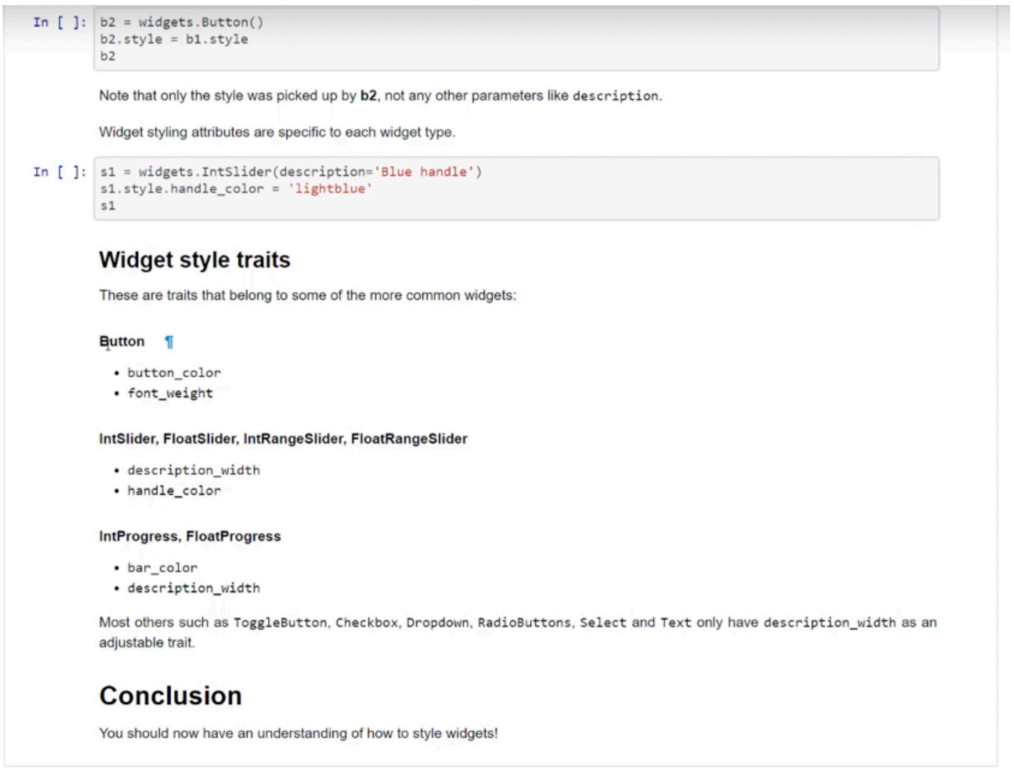
```
In [144]: b2 = widgets.Button(description='NEW')
          b2.style = b1.style
          b2
```

```
In [145]: s1 = widgets.IntSlider(description='My Handle')
          s1.style.
          s1
```

```
In [146]: s1.style.keys
```

```
Out[146]: ['_model_module',
           '_model_module_version',
           '_model_name',
           '_view_module',
           '_view_module_version',
           '_view_name',
           'handle_color',
           'msg_throttle']
```

```
In [148]: s1 = widgets.IntSlider(description='My Handle')
          s1.style.handle_color = 'black'
          s1
```



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

Note that only the style was picked up by **b2**, not any other parameters like `description`.

Widget styling attributes are specific to each widget type.

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

Widget style traits

These are traits that belong to some of the more common widgets:

Button ¶

- `button_color`
- `font_weight`

IntSlider, FloatSlider, IntRangeSlider, FloatRangeSlider

- `description_width`
- `handle_color`

IntProgress, FloatProgress

- `bar_color`
- `description_width`

Most others such as `ToggleButton`, `Checkbox`, `Dropdown`, `RadioButtons`, `Select` and `Text` only have `description_width` as an adjustable trait.

Conclusion

You should now have an understanding of how to style widgets!

Custom Widget

Exploring the Lorenz System of Differential Equations

In this Notebook we explore the Lorenz system of differential equations:

$$\dot{x} = \sigma(y - x)$$
$$\dot{y} = \rho x - y - xz$$
$$\dot{z} = -\beta z + xy$$

This is one of the classic systems in non-linear differential equations. It exhibits a range of different behaviors as the parameters (σ, β, ρ) are varied.

Imports

First, we import the needed things from IPython, NumPy, Matplotlib and SciPy. Check out the class Python for Data Science and Machine Learning Bootcamp if you're interested in learning more about this part of Python!

In [1]:
```
%matplotlib inline
```

In [ ]:
```
from ipywidgets import interact, interactive
from IPython.display import clear_output, display, HTML
```

In [ ]:
```
import numpy as np
from scipy import integrate

from matplotlib import pyplot as plt
from mpl_toolkits.mplot3d import Axes3D
from matplotlib.colors import cnames
from matplotlib import animation
```

Computing the trajectories and plotting the result

We define a function that can integrate the differential equations numerically and then plot the solutions. This function has arguments that control the parameters of the differential equation (σ, β, ρ), the numerical integration (N, max_time) and the visualization (angle).

In [ ]:
```
def solve_lorenz(N=10, angle=0.0, max_time=4.0, sigma=10.0, beta=8./3, rho=28.0):

    fig = plt.figure();
    ax = fig.add_axes([0, 0, 1, 1], projection='3d');
    ax.axis('off')

    # prepare the axes limits
    ax.set_xlim((-25, 25))
    ax.set_ylim((-35, 35))
    ax.set_zlim((5, 55))

    def lorenz_deriv(x_y_z, t0, sigma=sigma, beta=beta, rho=rho):
        """Compute the time-derivative of a Lorenz system."""
        x, y, z = x_y_z
        return [sigma * (y - x), x * (rho - z) - y, x * y - beta * z]

    # Choose random starting points, uniformly distributed from -15 to 15
    np.random.seed(1)
    x0 = -15 + 30 * np.random.random((N, 3))

    # Solve for the trajectories
    t = np.linspace(0, max_time, int(250*max_time))
    x_t = np.asarray([integrate.odeint(lorenz_deriv, x0i, t)
```