dcc.RangeSlider

Simple RangeSlider Example

An example of a simple dcc.RangeSlider tied to a callback. The callback takes the dcc.RangeSlider's currently selected range and outputs it to a `html.Div`.

```
from dash import Dash, html, dcc, Input, Output

external_stylesheets = ['https://codepen.io/chriddyp/pen/bWLwgP.css']

app = Dash(__name__, external_stylesheets=external_stylesheets)
app.layout = html.Div([
    dcc.RangeSlider(0, 20, 1, value=[5, 15], id='my-range-slider'),
    html.Div(id='output-container-range-slider')
])

@app.callback(
    Output('output-container-range-slider', 'children'),
    [Input('my-range-slider', 'value')])
def update_output(value):
    return 'You have selected "{}"'.format(value)

if __name__ == '__main__':
    app.run_server(debug=True)
```

You have selected "[4, 12]"

Min, Max, and Step

In the example above, the first three arguments provided (`0`, `20`, and `1`) are `min`, `max`, and `step` respectively.

`min` sets a minimum value available for selection on the dcc.RangeSlider, `max` sets a maximum, and `step` defines the points for the dcc.RangeSlider between the `min` and the `max`.

`dcc.RangeSlider` accepts these three arguments as positional arguments, but you can also provide them as keyword arguments. Using keyword arguments, the same dcc.RangeSlider component code looks like this:

```
dcc.RangeSlider(min=0, max=20, step=1, value=[5, 15], id='my-range-slider'),
```

Marks and Steps

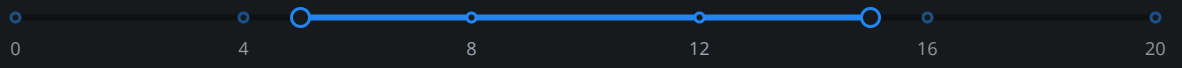
`marks` are the points displayed on the dcc.RangeSlider. In Dash >= 2.1, they are autogenerated if not explicitly provided or turned off.

Auto Generated Marks

By default, the `dcc.RangeSlider` component adds `marks` if they are not specified, as in the example above. It uses the `min` and `max` and and the `marks` correspond to the `step` if you use one.

If you don't supply `step`, `RangeSlider` automatically calculates a step and adds around five marks. Labels for autogenerated marks are SI unit formatted.

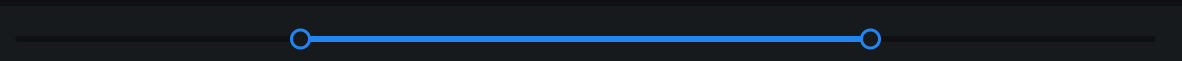
```
from dash import dcc

dcc.RangeSlider(0, 20, value=[5, 15])
```



Turn Off Marks

You can turn off marks by setting `marks=None`:

```
from dash import dcc

dcc.RangeSlider(0, 20, marks=None, value=[5, 15])
```



Custom Marks

You can also define custom marks. If `marks` are defined and `step` is set to `None` then the dcc.RangeSlider will only be able to select values that have been predefined by the `marks`. Note that the default is `step=1`, so you must explicitly specify `None` to get this behavior. `marks` is a `dict` where the keys represent the numerical values and the values represent their labels.

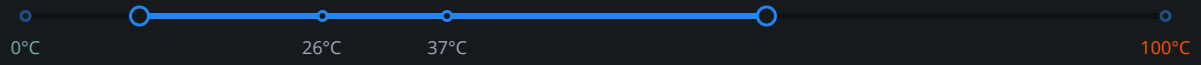
```
from dash import dcc

dcc.RangeSlider(
    min=0,
    max=10,
    step=None,
    marks={
        0: '0°F',
        3: '3°F',
        5: '5°F',
        7.65: '7.65°F',
        10: '10°F'
    },
    value=[3, 7.65]
)
```

Included and Styling Marks

By default, `included=True`, meaning the rail trailing the handle will be highlighted. To have the handle act as a discrete value, set `included=False`. To style `marks`, include a style CSS attribute alongside the key value.

```
from dash import dcc

# RangeSlider has included=True by default
```

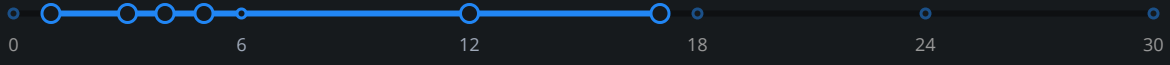
```
dcc.RangeSlider(0, 100, value=[10, 65], marks={
        0: {'label': '0°C', 'style': {'color': '#77b0b1'}},
        26: {'label': '26°C'},
        37: {'label': '37°C'},
        100: {'label': '100°C', 'style': {'color': '#f50'}}
    }
)
```



Multiple Handles

To create multiple handles, define more values for `value`.

```
from dash import dcc

dcc.RangeSlider(0, 30, value=[1, 3, 4, 5, 12, 17])
```



Pushable Handles

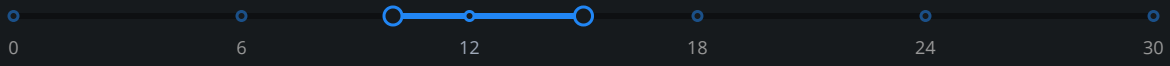
The `pushable` property is either a `boolean` or a numerical value. The numerical value determines the minimum distance between the handles. Try moving the handles around!

```
from dash import dcc

dcc.RangeSlider(0, 30, value=[8, 10, 15, 17, 20], pushable=2)
```

Non-Crossing Handles

To prevent handles from crossing each other, set `allowCross=False`.

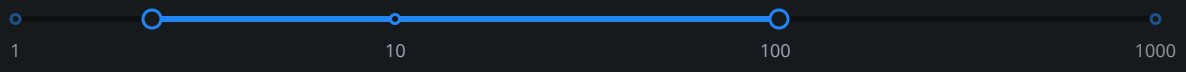
```
from dash import dcc

dcc.RangeSlider(0, 30, value=[10, 15], allowCross=False)
```



Non-Linear Slider and Updatemode

Create a logarithmic slider by setting `marks` to be logarithmic and adjusting the slider's output `value` in the callbacks. The `updatemode` property allows us to determine when we want a callback to be triggered. The following example has `updatemode='drag'` which means a callback is triggered everytime the handle is moved. Contrast the callback output with the first example on this page to see the difference.

```
from dash import Dash, dcc, html, Input, Output

external_stylesheets = ['https://codepen.io/chriddyp/pen/bWLwgP.css']

app = Dash(__name__, external_stylesheets=external_stylesheets)

# Use the following function when accessing the value of 'my-range-slider'
# in callbacks to transform the output value to logarithmic
def transform_value(value):
    return 10 ** value

app.layout = html.Div([
    dcc.RangeSlider(0, 3,
        id='non-linear-range-slider',
        marks={i: '{}'.format(10 ** i) for i in range(4)},
        value=[0.1, 2],
        dots=False,
        step=0.01,
        updatemode='drag'
    ),
    html.Div(id='output-container-range-slider-non-linear', style={'margin-top': 20})
])

@app.callback(
    Output('output-container-range-slider-non-linear', 'children'),
    Input('non-linear-range-slider', 'value'))
def update_output(value):
    transformed_value = [transform_value(v) for v in value]
    return 'Linear Value: {}, Log Value: [{:0.2f}, {:0.2f}]'.format(
        str(value),
        transformed_value[0],
        transformed_value[1]
    )

if __name__ == '__main__':
    app.run_server(debug=True)
```

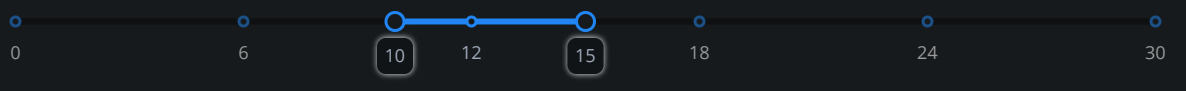


Linear Value: [0.36, 2.01], Log Value: [2.29, 102.33]

Tooltips

The `tooltips` property can be used to display the current value. The `placement` parameter controls the position of the tooltip i.e. 'left', 'right', 'top', 'bottom' and `always_visible=True` is used, then the tooltips will show always, otherwise it will only show when hovered upon.

```
from dash import dcc

dcc.RangeSlider(0, 30, value=[10, 15],
                tooltip={"placement": "bottom", "always_visible": True})
```



RangeSlider Properties

> Access this documentation in your Python terminal with:
>
> ```
> >>> help(dash.dcc.RangeSlider)
> ```

> Our recommended IDE for writing Dash apps is Dash Enterprise's **Data Science Workspaces**, which has typeahead support for Dash Component Properties. **Find out if your company is using Dash Enterprise**.

`min` (*number*; optional): Minimum allowed value of the slider.

`max` (*number*; optional): Maximum allowed value of the slider.

`step` (*number*; optional): Value by which increments or decrements are made.

`marks` (*dict*; optional): Marks on the slider. The key determines the position (a number), and the value determines what will show. If you want to set the style of a specific mark point, the value should be an object which contains style and label properties.

`marks` is a dict with strings as keys and values of type string | dict with keys:

- **`label`** (*string*; optional)
- **`style`** (*dict*; optional)

`value` (*list of numbers*; optional): The value of the input.

`drag_value` (*list of numbers*; optional): The value of the input during a drag.

`allowCross` (*boolean*; optional): allowCross could be set as True to allow those handles to cross.

`pushable` (*boolean | number*; optional): pushable could be set as True to allow pushing of surrounding handles when moving an handle. When set to a number, the number will be the minimum ensured distance between handles.

`disabled` (*boolean*; optional): If True, the handles can't be moved.

`count` (*number*; optional): Determine how many ranges to render, and multiple handles will be rendered (number + 1).

`dots` (*boolean*; optional): When the step value is greater than 1, you can set the dots to True if you want to render the slider with dots.

`included` (*boolean*; optional): If the value is True, it means a continuous value is included. Otherwise, it is an independent value.

`tooltip` (*dict*; optional): Configuration for tooltips describing the current slider values.

`tooltip` is a dict with keys:

- **`always_visible`** (*boolean*; optional): Determines whether tooltips should always be visible (as opposed to the default, visible on hover).
- **`placement`** (*a value equal to: 'left', 'right', 'top', 'bottom', 'topLeft', 'topRight', 'bottomLeft' or 'bottomRight'*; optional): Determines the placement of tooltips See **https://github.com/react-component/tooltip#api** top/bottom{*} sets the *origin* of the tooltip, so e.g. `topLeft` will in reality appear to be on the top right of the handle.

`updatemode` (*a value equal to: 'mouseup' or 'drag'*; default `'mouseup'`): Determines when the component should update its `value` property. If `mouseup` (the default) then the slider will only trigger its value when the user has finished dragging the slider. If `drag`, then the slider will update its value continuously as it is being dragged. Note that for the latter case, the `drag_value` property could be used instead.

`vertical` (*boolean*; optional): If True, the slider will be vertical.

`verticalHeight` (*number*; default `400`): The height, in px, of the slider if it is vertical.

`className` (*string*; optional): Additional CSS class for the root DOM node.

`id` (*string*; optional): The ID of this component, used to identify dash components in callbacks. The ID needs to be unique across all of the components in an app.

`loading_state` (*dict*; optional): Object that holds the loading state object coming from dash-renderer.

`loading_state` is a dict with keys:

- **`component_name`** (*string*; optional): Holds the name of the component that is loading.
- **`is_loading`** (*boolean*; optional): Determines if the component is loading or not.
- **`prop_name`** (*string*; optional): Holds which property is loading.

`persistence` (*boolean | string | number*; optional): Used to allow user interactions in this component to be persisted when the component - or the page - is refreshed. If `persisted` is truthy and hasn't changed from its previous value, a `value` that the user has changed while using the app will keep that change, as long as the new `value` also matches what was given originally. Used in conjunction with `persistence_type`.

persisted_props (*list of values equal to: 'value'*; default `['value']`): Properties whose user interactions will persist after refreshing the component or the page. Since only `value` is allowed this prop can normally be ignored.

persistence_type (*a value equal to: 'local', 'session' or 'memory'*; default `'local'`): Where persisted user changes will be stored: memory: only kept in memory, reset on page refresh. local: window.localStorage, data is kept after the browser quit. session: window.sessionStorage, data is cleared once the browser quit.