plotly

Star 23,446

Dash Python > ***Advanced Callbacks***

Plotly Studio: Transform any dataset into an interactive data application in minutes with AI. **Sign up for early access now.**

Advanced Callbacks

To get the most out of this page, make sure you've read about **Basic Callbacks** in the Dash Fundamentals.

Catching Errors with `PreventUpdate`

In certain situations, you don't want to update the callback output. You can achieve this by raising a `PreventUpdate` exception in the callback function.

```
from dash import Dash, html, Input, Output, callback
from dash.exceptions import PreventUpdate

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

app = Dash(__name__, external_stylesheets=external_stylesheets)

app.layout = html.Div([
    html.Button('Click here to see the content', id='show-secret'),
    html.Div(id='body-div')
])

@callback(
    Output('body-div', 'children'),
    Input('show-secret', 'n_clicks')
)
def update_output(n_clicks):
    if n_clicks is None:
        raise PreventUpdate
    else:
        return "Elephants are the only animal that can't jump"

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

CLICK HERE TO SEE THE CONTENT

Displaying Errors with `dash.no_update`

This example illustrates how you can show an error while keeping the previous input, using `dash.no_update` to update only some of the callback outputs.

```
from dash import Dash, dcc, html, Input, Output, callback, no_update
from dash.exceptions import PreventUpdate

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

app = Dash(__name__, external_stylesheets=external_stylesheets)

app.layout = html.Div([
```

```
    html.P('Enter a composite number to see its prime factors'),
    dcc.Input(id='num', type='number', debounce=True, min=2, step=1),
    html.P(id='err', style={'color': 'red'}),
    html.P(id='out')
])

@callback(
    Output('out', 'children'),
    Output('err', 'children'),
    Input('num', 'value')
)
def show_factors(num):
    if num is None:
        # PreventUpdate prevents ALL outputs updating
        raise PreventUpdate

    factors = prime_factors(num)
    if len(factors) == 1:
        # dash.no_update prevents any single output updating
        # (note: it's OK to use for a single-output callback too)
        return no_update, '{} is prime!'.format(num)

    return '{} is {}'.format(num, ' * '.join(str(n) for n in factors)), ''

def prime_factors(num):
    n, i, out = num, 2, []
    while i * i <= n:
        if n % i == 0:
            n = int(n / i)
            out.append(i)
        else:
            i += 1 if i == 2 else 2
    out.append(n)
    return out

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

Enter a composite number to see its prime factors

Updating Component Properties when a Callback is Running

New in Dash 2.16

You can use the `running` argument on a callback to update specific component-property pairs when the callback is running. For example, you could disable the button that triggered the callback while the callback is still running.

`running` accepts a list of three element tuples, where:

- The first element of each tuple must be an `Output` dependency object referencing a property of a component in the app layout.
- The second element is the value that the property should be set to while the callback is running.
- The third element is the value the property should be set to when the callback completes.

In the following example, our `running` argument sets the `disabled` property on the `submit-button` to `True` while the callback is running and sets it back to `False` once the callback completes.

```
from dash import Dash, dcc, html, Input, Output, State, callback
import time

app = Dash()

app.layout = html.Div([
    html.Div(dcc.Input(id='input-on-submit-text', type='text')),
    html.Button('Submit', id='submit-button', n_clicks=0),
    html.Div(id='container-output-text',
```

```
            children='Enter a value and press submit')
])


@callback(
    Output('container-output-text', 'children'),
    Input('submit-button', 'n_clicks'),
    State('input-on-submit-text', 'value'),
    prevent_initial_call=True,
    running=[(Output("submit-button", "disabled"), True, False)]
)
def update_output(n_clicks, value):
    time.sleep(5)
    return 'The input value was "{}" and the button has been clicked {} times'.format(
        value,
        n_clicks
    )


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

SUBMIT

Enter a value and press submit

> There is a known issue where using `running` with a multi-pages app doesn't work as expected when a user changes page when the callback is running.

Determining which `Input` Has Fired with `dash.callback_context`

In addition to event properties like `n_clicks` that change whenever an event happens (in this case a click), there is a global variable `dash.callback_context`, available only inside a callback. Using `dash.callback_context`, you can determine which component/property pairs triggered a callback.

Below is a summary of properties of `dash.callback_context` outlining the basics of when to use them. For more detail and examples see **Determining Which Callback Input Changed**.

For more examples of minimal Dash apps that use `dash.callback_context`, go to the community-driven **Example Index**.

Properties for `callback_context`

- `triggered_id`: The `id` of the component that triggered the callback. *Available in Dash 2.4 and later*.
- `triggered_prop_ids`: A dictionary of the component ids and props that triggered the callback. Useful when multiple inputs can trigger the callback at the same time, or multiple properties of the same component can trigger the callback. *Available in Dash 2.4 and later*.
- `args_grouping`: A dictionary of the inputs used with flexible callback signatures. *Available in Dash 2.4 and later*.

  The keys are the variable names and the values are dictionaries containing:

  - "id": the component ID. If it's a pattern matching ID, it will be a dict.
  - "id_str": for pattern matching IDs, it's the stringified dict ID with no white spaces.
  - "property": the component property used in the callback.
  - "value": the value of the component property at the time the callback was fired.
  - "triggered": a boolean indicating whether this input triggered the callback.
- `triggered`: list of all the `Input` props that changed and caused the callback to execute. It is empty when the callback is called on initial load, unless an `Input` prop got its value from another initial callback. Callbacks triggered by user actions typically have one item in `triggered`, unless the same action changes

two props at once or the callback has several `Input` props that are all modified by another callback based on a single user action.

> **More about empty `triggered` lists**: For backward compatibility purposes, an empty `triggered` is not really empty. It's falsy so that you can use `if triggered` to detect the initial call, but it still has a placeholder element so that `ctx.triggered[0]["prop_id"].split(".")` yields a blank ID and prop `["", ""]` instead of an error.

- `inputs` and `states`: allow you to access the callback params by ID and prop instead of through the function args. These have the form of dictionaries `{ 'component_id.prop_name': value }`}}
- `outputs_list`, `inputs_list`, and `states_list`: lists of inputs, outputs, and state items arranged as you'll find them in the callback arguments and return value. This is mostly useful for **pattern-matching callbacks**.
- `response`: The HTTP response object being constructed, useful for changing cookies.
- `record_timing`: a method to report granular timing information, to be seen in the **Dev Tools**.
- `custom_data`: Custom data available to callbacks. Can be set with `dash.hooks.custom_data`. See **Making Custom Data Available in a Callback** for more information. *Available in Dash 3.0 and later*.

Here's an example of how this can be done:

```
import json
from dash import Dash, html, Input, Output, callback, ctx

app = Dash()

app.layout = html.Div([
    html.Button('Button 1', id='btn-1'),
    html.Button('Button 2', id='btn-2'),
    html.Button('Button 3', id='btn-3'),
    html.Div(id='container')
])


@callback(Output('container', 'children'),
              Input('btn-1', 'n_clicks'),
              Input('btn-2', 'n_clicks'),
              Input('btn-3', 'n_clicks'))
def display(btn1, btn2, btn3):
    if not ctx.triggered_id:
        button_id = 'No clicks yet'
    else:
        button_id = ctx.triggered_id

    ctx_msg = json.dumps({
        'states': ctx.states,
        'triggered': ctx.triggered,
        'inputs': ctx.inputs
    }, indent=2)

    return html.Div([
        html.Table([
            html.Tr([html.Th('Button 1'),
                     html.Th('Button 2'),
                     html.Th('Button 3'),
                     html.Th('Most Recent Click')]),
            html.Tr([html.Td(btn1 or 0),
                     html.Td(btn2 or 0),
                     html.Td(btn3 or 0),
                     html.Td(button_id)])
        ]),
        html.Pre(ctx_msg)
    ])


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

Button 1	Button 2	Button 3	Most Recent Click
0	0	0	No clicks yet


```
{
  "states": {},
  "triggered": [],
  "inputs": {
    "btn-1.n_clicks": null,
    "btn-2.n_clicks": null,
    "btn-3.n_clicks": null
  }
}
```


Improving Performance with Memoization

Memoization allows you to bypass long computations by storing the results of function calls.

To better understand how memoization works, let's start with a simple example.

```
import time
import functools32

@functools32.lru_cache(maxsize=32)
def slow_function(input):
    time.sleep(10)
    return f'Input was {input}'
```

Calling `slow_function('test')` the first time will take 10 seconds. Calling it a second time with the same argument will take almost no time since the previously computed result was saved in memory and reused.

The **Performance** section of the Dash docs delves a little deeper into leveraging multiple processes and threads in conjunction with memoization to further improve performance.

Making Callback States and Inputs Optional

New in Dash 3.1

In some cases you may have components in your app that don't exist when a callback runs, for example if you have a component that is conditionally rendered. By default, Dash will throw an error if you use a non-existent component as an `Input` or `State` for a callback. You'll see an error like: `'A nonexistent object was used...'`

You can set these `Input` or `State` objects to be optional using the `allow_optional` parameter. With `allow_optional=True`, the callback will receive `None` for any inputs or states that use it that don't exist.

The following example shows a button that is conditionally rendered from one callback and used as an optional input on another callback:

```
from dash import Dash, html, Output, Input, State, no_update

app = Dash(suppress_callback_exceptions=True)

app.layout = html.Div([
    html.Div([
        html.Button(id="optional-inputs-button-1", children="Button 1", className="button"),
        html.Div(id="optional-inputs-container"),
        html.Div(id="optional-inputs-output", className="output")
    ], className="container")
])

@app.callback(
    Output("optional-inputs-container", "children"),
    Input("optional-inputs-button-1", "n_clicks"),
    State("optional-inputs-container", "children"),
    prevent_initial_call=True
)
def add_second_button(_, current_children):
```

```
    if not current_children:
        return html.Button(id="optional-inputs-button-2", children="Button 2", className="butto
    return no_update

@app.callback(
    Output("optional-inputs-output", "children"),
    Input("optional-inputs-button-1", "n_clicks"),
    Input("optional-inputs-button-2", "n_clicks", allow_optional=True),
    prevent_initial_call=True
)
def display_clicks(n_clicks1, n_clicks2):
    return f"Button 1 clicks: {n_clicks1} - Button 2 clicks: {n_clicks2}"

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

Using Async/Await in Callbacks

New in Dash 3.1

Dash supports using `async`/`await` in callbacks. To use `async`/`await` in callbacks, `pip install "dash[async]"`.

Using `async`/`await` is useful when making requests, communicating with databases, or when using async-first libraires such as **`aiofiles`** and **`tortoise-orm`**.

The following example demonstrates using `async` / `await` in a callback.

```
import time
from dash import html, Input, Output, Dash
import asyncio

app = Dash()
server = app.server

def get_sync_data(iteration):
    time.sleep(1)
    return f"Result ({iteration}) from synchronous API call"

async def get_async_data(iteration):
    await asyncio.sleep(1)
    return f"Result ({iteration}) from asynchronous API call"

app.layout = html.Div([
    html.H2("Synchronous vs. Asynchronous Functions in Dash"),
    html.Button("Run Sync Tasks", id="sync-btn", style={'margin-right': '10px'}),
    html.Button("Run Async Tasks", id="async-btn"),
    html.Hr(),
    html.Div(id="sync-output", style={'color': 'red', 'font-weight': 'bold'}),
    html.Div(id="async-output", style={'color': 'green', 'font-weight': 'bold'}),
])

@app.callback(
    Output("sync-output", "children"),
    Input("sync-btn", "n_clicks"),
    prevent_initial_call=True,
)
def sync_callback_example(n_clicks):
    if n_clicks:
        results = [get_sync_data(i) for i in range(5)]
        return html.Div([html.Div(result) for result in results])
    return ""

@app.callback(
    Output("async-output", "children"),
    Input("async-btn", "n_clicks"),
    prevent_initial_call=True,
)
```

```
async def async_callback_example(n_clicks):
    if n_clicks:
        coros = [get_async_data(i) for i in range(5)]
        results = await asyncio.gather(*coros)
        return html.Div([html.Div(result) for result in results])
    return ""

if __name__ == "__main__":
    app.run(debug=True)
```

When you install `dash[async]`, Dash installs **Flask async's** dependencies.

Note: The `gevent` **worker class** with `gunicorn` is not supported. Using `gunicorn` with `gthread` worker class is supported. For example: `gunicorn app:app -k gthread --threads=2`

When Are Callbacks Executed?

This section describes the circumstances under which the `dash-renderer` front-end client can make a request to the Dash back-end server (or the clientside callback code) to execute a callback function.

When a Dash App First Loads

All of the callbacks in a Dash app are executed with the initial value of their inputs when the app is first loaded. This is known as the "initial call" of the callback. To learn how to suppress this behavior, see the documentation for the `prevent_initial_call` attribute of Dash callbacks.

It is important to note that when a Dash app is initially loaded in a web browser by the `dash-renderer` front-end client, its entire callback chain is introspected recursively.

This allows the `dash-renderer` to predict the order in which callbacks will need to be executed, as callbacks are blocked when their inputs are outputs of other callbacks which have not yet fired. In order to unblock the execution of these callbacks, first callbacks whose inputs are immediately available must be executed. This process helps the `dash-renderer` to minimize the time and effort it uses, and avoid unnecessarily redrawing the page, by making sure it only requests that a callback is executed when all of the callback's inputs have reached their final values.

Examine the following Dash app:

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

app = Dash()
app.layout = html.Div(
    [
        html.Button("execute callback", id="button_1"),
        html.Div(children="callback not executed", id="first_output_1"),
        html.Div(children="callback not executed", id="second_output_1"),
    ]
)


@callback(
    Output("first_output_1", "children"),
    Output("second_output_1", "children"),
    Input("button_1", "n_clicks")
)
def change_text(n_clicks):
    return ["n_clicks is " + str(n_clicks), "n_clicks is " + str(n_clicks)]

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

```
EXECUTE CALLBACK
n_clicks is None
n_clicks is None
```

Notice that when this app is finished being loaded by a web browser and ready for user interaction, the `html.Div` components do not say "callback not executed" as declared in the app's layout, but rather "n_clicks is None" as the result of the `change_text()` callback being executed. This is because the "initial call" of the callback occurred with `n_clicks` having the value of `None`.

As a Direct Result of User Interaction

Most frequently, callbacks are executed as a direct result of user interaction, such as clicking a button or selecting an item in a dropdown menu. When such interactions occur, Dash components communicate their new values to the `dash-renderer` front-end client, which then requests that the Dash server execute any callback function that has the newly changed value as input.

If a Dash app has multiple callbacks, the `dash-renderer` requests callbacks to be executed based on whether or not they can be immediately executed with the newly changed inputs. If several inputs change simultaneously, then requests are made to execute them all.

Whether or not these requests are executed in a synchronous or asynchronous manner depends on the specific setup of the Dash back-end server. If it is running in a multi-threaded environment, then all of the callbacks can be executed simultaneously, and they will return values based on their speed of execution. In a single-threaded environment however, callbacks will be executed one at a time in the order they are received by the server.

In the example application above, clicking the button results in the callback being executed.

As an Indirect Result of User Interaction

When a user interacts with a component, the resulting callback might have outputs that are themselves the input of other callbacks. The `dash-renderer` will block the execution of such a callback until the callback whose output is its input has been executed.

Take the following Dash app:

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

from datetime import datetime
import time

app = Dash()
app.layout = html.Div(
    [
        html.Button("execute fast callback", id="button_3"),
        html.Button("execute slow callback", id="button_4"),
        html.Div(children="callback not executed", id="first_output_3"),
        html.Div(children="callback not executed", id="second_output_3"),
        html.Div(children="callback not executed", id="third_output_3"),
    ]
)


@callback(
    Output("first_output_3", "children"),
    Input("button_3", "n_clicks"))
def first_callback(n):
    now = datetime.now()
    current_time = now.strftime("%H:%M:%S")
    return "in the fast callback it is " + current_time


@callback(
    Output("second_output_3", "children"), Input("button_4", "n_clicks"))
def second_callback(n):
    time.sleep(5)
    now = datetime.now()
    current_time = now.strftime("%H:%M:%S")
    return "in the slow callback it is " + current_time
```

```
@callback(
    Output("third_output_3", "children"),
    Input("first_output_3", "children"),
    Input("second_output_3", "children"))
def third_callback(n, m):
    now = datetime.now()
    current_time = now.strftime("%H:%M:%S")
    return "in the third callback it is " + current_time


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

EXECUTE FAST CALLBACK EXECUTE SLOW CALLBACK

in the fast callback it is 08:22:33
in the slow callback it is 08:22:38
in the third callback it is 08:22:39

The above Dash app demonstrates how callbacks chain together. Notice that if you first click "execute slow callback" and then click "execute fast callback", the third callback is not executed until after the slow callback finishes executing. This is because the third callback has the second callback's output as its input, which lets the `dash-renderer` know that it should delay its execution until after the second callback finishes.

When Dash Components Are Added to the Layout

It is possible for a callback to insert new Dash components into a Dash app's layout. If these new components are themselves the inputs to other callback functions, then their appearance in the Dash app's layout will trigger those callback functions to be executed.

In this circumstance, it is possible that multiple requests are made to execute the same callback function. This would occur if the callback in question has already been requested and its output returned before the new components which are also its inputs are added to the layout.

Prevent Callback Execution Upon Initial Component Render

You can use the `prevent_initial_call` attribute to prevent callbacks from firing when their inputs initially appear in the layout of your Dash application.

This attribute applies when the layout of your Dash app is initially loaded, and also when new components are introduced into the layout when a callback has been triggered.

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

from datetime import datetime
import time

app = Dash()

app.layout = html.Div(
    [
        html.Button("execute callbacks", id="button_2"),
        html.Div(children="callback not executed", id="first_output_2"),
        html.Div(children="callback not executed", id="second_output_2"),
        html.Div(children="callback not executed", id="third_output_2"),
        html.Div(children="callback not executed", id="fourth_output_2"),
    ]
)


@callback(
    Output("first_output_2", "children"),
    Output("second_output_2", "children"),
    Input("button_2", "n_clicks"), prevent_initial_call=True)
def first_callback(n):
    now = datetime.now()
    current_time = now.strftime("%H:%M:%S")
```

```
    return ["in the first callback it is " + current_time, "in the first callback it is " + cu


@callback(
    Output("third_output_2", "children"), Input("second_output_2", "children"), prevent_initia
def second_callback(n):
    time.sleep(2)
    now = datetime.now()
    current_time = now.strftime("%H:%M:%S")
    return "in the second callback it is " + current_time


@callback(
    Output("fourth_output_2", "children"),
    Input("first_output_2", "children"),
    Input("third_output_2", "children"), prevent_initial_call=True)
def third_output(n, m):
    time.sleep(2)
    now = datetime.now()
    current_time = now.strftime("%H:%M:%S")
    return "in the third callback it is " + current_time


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

EXECUTE CALLBACKS

callback not executed
callback not executed
callback not executed
callback not executed

However, the above behavior only applies if both the callback output and input are present in the app layout upon initial load of the application.

It is important to note that `prevent_initial_call` will not prevent a callback from firing in the case where the callback's input is inserted into the layout as the result of another callback after the app initially loads **unless the output is inserted alongside that input**!

In other words, if the output of the callback is already present in the app layout before its input is inserted into the layout, `prevent_initial_call` will not prevent its execution when the input is first inserted into the layout.

Consider the following example:

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

app = Dash(__name__, suppress_callback_exceptions=True)
server = app.server
app.layout = html.Div([
    dcc.Location(id='url'),
    html.Div(id='layout-div'),
    html.Div(id='content')
])

@callback(Output('content', 'children'), Input('url', 'pathname'))
def display_page(pathname):
    return html.Div([
        dcc.Input(id='input', value='hello world'),
        html.Div(id='output')
    ])

@callback(Output('output', 'children'), Input('input', 'value'), prevent_initial_call=True)
def update_output(value):
    print('>>> update_output')
    return value

@callback(Output('layout-div', 'children'), Input('input', 'value'), prevent_initial_call=True)
def update_layout_div(value):
```

```
    print('>>> update_layout_div')
    return value
```

In this case, `prevent_initial_call` will prevent the `update_output()` callback from firing when its input is first inserted into the app layout as a result of the `display_page()` callback. This is because both the input and output of the callback are already contained within the app layout when the callback executes.

However, because the app layout contains only the output of the callback, and not its input, `prevent_initial_call` will not prevent the `update_layout_div()` callback from firing. Since `suppress_callback_exceptions=True` is specified here, Dash has to assume that the input is present in the app layout when the app is initialized. From the perspective of the output element in this example, the new input component is handled as if an existing input had been provided a new value, rather than treating it as initially rendered.

Callbacks with No Outputs

New in 2.17

All previous examples have inputs that trigger callbacks as well as outputs that are updated. Callbacks also support having no outputs. This can be useful for cases where you want some code to run when an action triggers a callback, but you don't want to update any component property. For example, you may want to fetch some data and save it, send emails, or interact with other external services outside of the Dash ecosystem.

In the following example, using the `dcc.Upload` component, we allow the user to upload a file to the server.

Our example has no `return` statements. Callbacks without outputs should not return a value, because there are no outputs to update. If you return a value from the callback when there is no output to update, you'll see an error in **Dash Dev Tools**.

```
import base64

from dash import Dash, html, Input, Output, dcc, callback, State


app = Dash(__name__)

app.layout = html.Div(
    [
        html.H1("No Output Example"),
        dcc.Upload(
            id='upload-data-to-server',
            children=html.Div([
                'Drag and Drop or ',
                html.A('Select Files')
            ]),
            style={
                'width': '100%',
                'height': '60px',
                'lineHeight': '60px',
                'borderWidth': '1px',
                'borderStyle': 'dashed',
                'borderRadius': '5px',
                'textAlign': 'center',
                'margin': '10px'
            },
        ),
    ]
)


@callback(
    Input("upload-data-to-server", "contents"),
    State('upload-data-to-server', 'filename'),
    # We could also use `running` on the callback to disable the button
    # while the callback is running:
    # running=[(Output("upload-data-to-server", "disabled"), True, False)]
)
def update_output_div(contents, filename):
    if contents is not None:
        _, content_string = contents.split(',')
        decoded = base64.b64decode(content_string)
```

```
        directory = './uploaded_files'
        with open(f'{directory}/{filename}', 'wb') as f:
            f.write(decoded)


if __name__ == "__main__":
    app.run(debug=True)
```

Setting Properties Directly

New in 2.17

You can update component-property pairs directly from a callback without them being callback outputs by using `set_props`. With this approach, conditionally updating different component-property pairs is simpler because you don't need to add them all as outputs and use `dash.no_update`.

`set_props` takes the ID of the component to update as its first argument, followed by a `dict` of properties to update. Each `dict` key should be a property name, with the key's value being the value to update the component property to.

Here, we have a callback that uses `set_props` to set the modal's `is_open` property to `False` when the callback is triggered by a user selecting the `modal_close` button. Otherwise, it's set to `True` and the modal content is displayed. We have no outputs on this callback, but `set_props` can also be combined with outputs.

This example is based on the **Popup On Row Selection Example** on the Dash AG Grid page, which instead uses outputs and `dash.no_update`.

> On callbacks that don't run in the background, like in the following example, updates made using `set_props` and via outputs all happen at the same time, when the callback finishes running. On background callbacks, `set_props` updates take place immediately. See the **Background Callbacks page** for an example.

```
import dash_ag_grid as dag
from dash import Dash, Input, html, ctx, callback, set_props
import dash_bootstrap_components as dbc

app = Dash(__name__, external_stylesheets=[dbc.themes.SPACELAB])

rowData = [
    {"make": "Toyota", "model": "Celica", "price": 35000},
    {"make": "Ford", "model": "Mondeo", "price": 32000},
    {"make": "Porsche", "model": "Boxster", "price": 72000},
]

app.layout = html.Div(
    [
        dag.AgGrid(
            id="setprops-row-selection-popup",
            rowData=rowData,
            columnDefs=[{"field": i} for i in ["make", "model", "price"]],
            columnSize="sizeToFit",
            dashGridOptions={"rowSelection": "single", "animateRows": False},
        ),
        dbc.Modal(
            [
                dbc.ModalHeader("More information about selected row"),
                dbc.ModalBody(id="setprops-row-selection-modal-content"),
                dbc.ModalFooter(dbc.Button("Close", id="setprops-row-selection-modal-close", c
            ],
            id="setprops-row-selection-modal",
        ),
    ]
)

@callback(
    Input("setprops-row-selection-popup", "selectedRows"),
    Input("setprops-row-selection-modal-close", "n_clicks"),
    prevent_initial_call=True,
)
def open_modal(selection, _):
    if ctx.triggered_id == "setprops-row-selection-modal-close":
```

```
        # Close the modal
        set_props("setprops-row-selection-modal", {'is_open': False})
    elif ctx.triggered_id == "setprops-row-selection-popup" and selection:
        # Open the modal and display the selected row content
        content_to_display = "You selected " + ", ".join(
            [
                f"{s['make']} (model {s['model']} and price {s['price']})"
                for s in selection
            ]
        )
        set_props("setprops-row-selection-modal", {'is_open': True})
        set_props("setprops-row-selection-modal-content", {'children': content_to_display})
```

Make	Model	Price
Toyota	Celica	35000
Ford	Mondeo	32000
Porsche	Boxster	72000

In the example above, we use `set_props` to simplify the code. In other cases, however, using `Outputs` may lead to simpler code because the `Outputs` are defined in the callback signature, making it easier to tell which component properties a callback updates.

Limitations

- Component properties updated using `set_props` won't appear in the **callback graph** for debugging.
- Component properties updated using `set_props` won't appear as loading when they are wrapped with a **`dcc.Loading` component**.
- `set_props` doesn't validate the `id` or `property` names provided, so no error will be displayed if they contain typos. This can make apps that use `set_props` harder to debug.
- Using `set_props` with chained callbacks may lead to unexpected results.

Circular Callbacks

As of `dash v1.19.0`, you can create circular updates *within the same callback*.

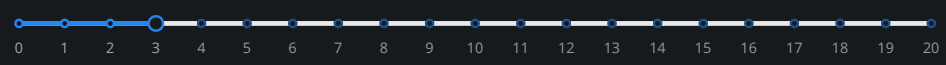
Circular callback chains that involve multiple callbacks are not supported.

Circular callbacks can be used to keep multiple inputs synchronized to each other.

Synchronizing a Slider with a Text Input Example

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

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

app = Dash(__name__, external_stylesheets=external_stylesheets)

app.layout = html.Div(
    [
        dcc.Slider(
            id="slider-circular", min=0, max=20,
            marks={i: str(i) for i in range(21)},
            value=3
        ),
```

```
        dcc.Input(
            id="input-circular", type="number", min=0, max=20, value=3
        ),
    ]
)
@callback(
    Output("input-circular", "value"),
    Output("slider-circular", "value"),
    Input("input-circular", "value"),
    Input("slider-circular", "value"),
)
def callback(input_value, slider_value):
    trigger_id = ctx.triggered[0]["prop_id"].split(".")[0]
    value = input_value if trigger_id == "input-circular" else slider_value
    return value, value

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



3

Displaying Two Inputs with Different Units Example

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

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

app = Dash(__name__, external_stylesheets=external_stylesheets)

app.layout = html.Div([
    html.Div('Convert Temperature'),
    'Celsius',
    dcc.Input(
        id="celsius",
        value=0.0,
        type="number"
    ),
    ' = Fahrenheit',
    dcc.Input(
        id="fahrenheit",
        value=32.0,
        type="number",
    ),
])

@callback(
    Output("celsius", "value"),
    Output("fahrenheit", "value"),
    Input("celsius", "value"),
    Input("fahrenheit", "value"),
)
def sync_input(celsius, fahrenheit):
    input_id = ctx.triggered[0]["prop_id"].split(".")[0]
    if input_id == "celsius":
        fahrenheit= None if celsius is None else (float(celsius) * 9/5) + 32
    else:
        celsius = None if fahrenheit is None else (float(fahrenheit) - 32) * 5/9
    return celsius, fahrenheit

if __name__ == "__main__":
    app.run(debug=True)
```

Convert Temperature

Celsius 0 = Fahrenheit 32

Synchronizing Two Checklists

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

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

app = Dash(__name__, external_stylesheets=external_stylesheets)

options = ["New York City", "Montréal", "San Francisco"]

app.layout = html.Div(
    [
        dcc.Checklist(["All"], [], id="all-checklist", inline=True),
        dcc.Checklist(options, [], id="city-checklist", inline=True),
    ]
)
@callback(
    Output("city-checklist", "value"),
    Output("all-checklist", "value"),
    Input("city-checklist", "value"),
    Input("all-checklist", "value"),
)
def sync_checklists(cities_selected, all_selected):
    ctx = callback_context
    input_id = ctx.triggered[0]["prop_id"].split(".")[0]
    if input_id == "city-checklist":
        all_selected = ["All"] if set(cities_selected) == set(options) else []
    else:
        cities_selected = options if all_selected else []
    return cities_selected, all_selected

if __name__ == "__main__":
    app.run(debug=True)
```

☐ All
☐ New York City ☐ Montréal ☐ San Francisco

Dash Python ***> Advanced Callbacks***