plotly

Star 23,446

Dash Python > ***Partial Property Updates***

Partial Property Updates

New in Dash 2.9. To get the most out of this page, make sure you've read about **Basic Callbacks** in the Dash Fundamentals.

Most of the callback examples we've seen in earlier chapters updated an entire property when the callback ran. In cases where we only want to update a small part of a property, for example, a title on a graph, this is inefficient. It means all the property's data (the complete figure in this case) is sent back across the network, even though most of the data hasn't changed. Improve your app performance by making partial property updates when a full update is not required.

The Dash Patch Class

Partial property updates use the `Patch` class, which you can import with `from dash import Patch`. Instantiate a `Patch` object in a callback to make partial updates to a callback output. This `Patch` object defines the changes that should be made to the property. Possible changes include assigning a new value, merging a dictionary, and appending an item to a list. The operations are defined by the `Patch` object within your callback but executed in the browser client in Dash's frontend.

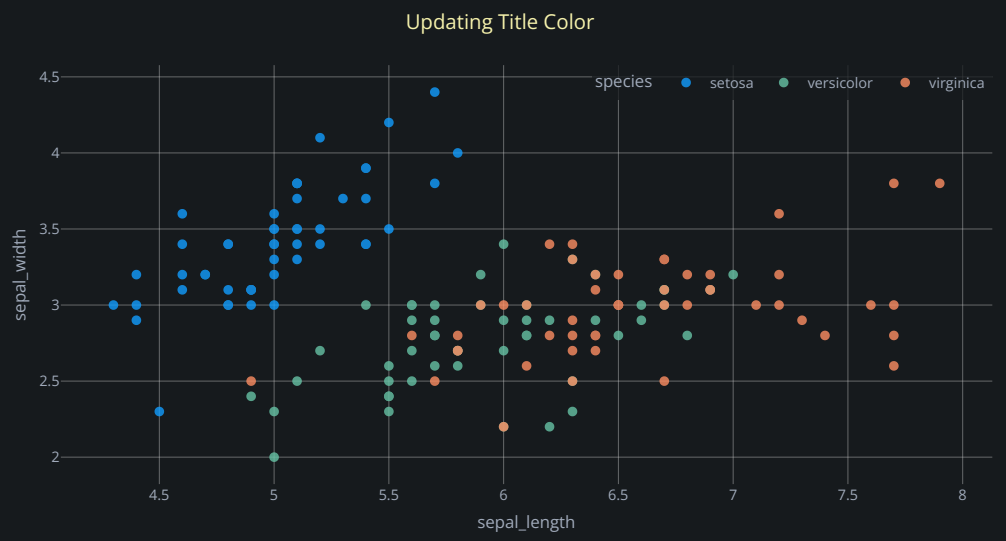
Updating Title Color with a Partial Update

Here's an example of using a `Patch` to update only the graph's title font color:

```
from dash import Dash, html, dcc, Input, Output, Patch, callback
import plotly.express as px
import random

app = Dash()

df = px.data.iris()
fig = px.scatter(
    df, x="sepal_length", y="sepal_width", color="species", title="Updating Title Color"
)

app.layout = html.Div(
    [
        html.Button("Update Graph Color", id="update-color-button-2"),
        dcc.Graph(figure=fig, id="my-fig"),
    ]
)


@callback(Output("my-fig", "figure"), Input("update-color-button-2", "n_clicks"))
def my_callback(n_clicks):
    # Defining a new random color
    red = random.randint(0, 255)
    green = random.randint(0, 255)
    blue = random.randint(0, 255)
    new_color = f"rgb({red}, {green}, {blue})"

    # Creating a Patch object
    patched_figure = Patch()
    patched_figure["layout"]["title"]["font"]["color"] = new_color
```

```
    return patched_figure

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

UPDATE GRAPH COLOR



Let's take a closer look at what's happening here:

1. First, we import the `Patch` class:

```
from dash import Patch
```

2. We define our callback inputs and outputs in the same way as with a standard callback:

```
@callback(
    Output('my-fig', 'figure'),
    Input('update-color-button-2', 'n_clicks')
)
```

3. In our callback, we create a `Patch` object. Here, we call it `patched_figure`, but the naming is arbitrary. This `patched_figure` will define the changes that Dash should make to the `figure`.

```
patched_figure = Patch()
```

4. Then, we define an assignment change, telling Dash that we want this part of the figure to be the value in `new_color`.

```
patched_figure['layout']['title']['font']['color'] = new_color
```

5. We return our `patched_figure`:

```
return patched_figure
```

6. Once the callback returns the `Patch` object, in the frontend, Dash assigns the value in `new_color` to `['layout']['title']['font']['color']` on the figure.

> In this example, we updated the **title's font color** by instructing Dash to update `['layout']['title']['font']['color']` on the figure. For more details on how to explore the property attributes of a Graph Objects Figure see the "Exploring the Structure of Properties" section below.

▶ Here's how you would update the title font color without partial property updates

Performance

Let's look at the difference when doing a partial update instead of a full update. The graph in our earlier example uses a small data set of 150 rows. Here the size difference is relatively small — 9.5 kB for the full update vs 380 B for the partial update.

Full Output Size

Name	Initiator	Size
_dash-update-component	dash_renderer.v2_8_0m1676304142.min.js:2	9.5 kB

Patch Output Size

Name	Initiator	Size
_dash-update-component	dash_renderer.v2_8_0m1676304142.min.js:2	380 B

However, in a graph with a much larger data set, the size of a full output increases, whereas the size of the response for our partial update to the title stays the same. Here's the response size of the same example, but using a data set of 45000 rows in the graph. The full output size is 368 kB, while the partial update stays the same at 380 B.

Full Output Size

Name	Initiator	Size
_dash-update-component	dash_renderer.v2_8_0m1676304142.min.js:2	368 kB

Patch Output Size

Name	Initiator	Size
_dash-update-component	dash_renderer.v2_8_0m1676304142.min.js:2	380 B

Partial Update Methods

There are multiple ways you can use `Patch` objects to make partial updates.

> Although all the methods outlined below are always available when using a `Patch` object, they won't work with every property or attribute type. For, example, `prepend`, `extend`, and `append` work with lists while `update` works with dictionaries. See the Exploring the Structure of Properties section below for more details on understanding the property you are updating.

Assign

In the example above, we use assignment with the `Patch` object. We assign `new_color` like this:

```
patched_figure['layout']['title']['font']['color'] = new_color
```

You can also use dot notation. So you could assign `new_color` like this:

```
patched_figure.layout.title.font.color = new_color
```

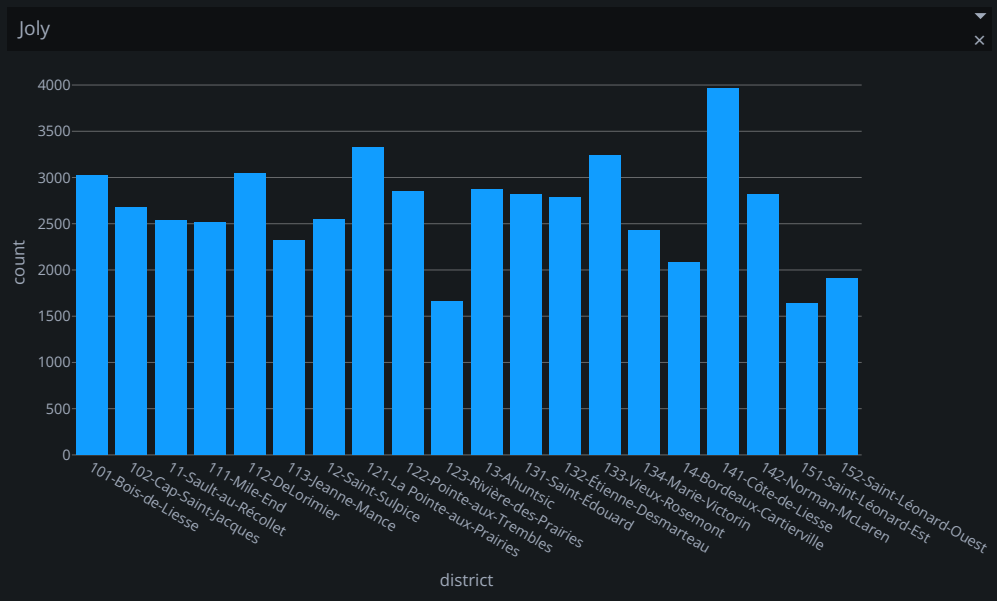
> To use dot notation to assign to a property attribute, the attribute name must be a **valid Python identifier**.

Examples of Assigning New Data

In this example, we update the values on the y-axis based on the dropdown selection.

```
from dash import Dash, html, dcc, Input, Output, Patch, callback
import plotly.express as px
```

```
app = Dash()

# Get data
df = px.data.election()[:20]
# Create figure based on data
fig = px.bar(df, x="district")

app.layout = html.Div(
    [
        dcc.Dropdown(
            ["Coderre", "Joly", "Bergeron"], id="candidate-select", value="Joly"
        ),
        dcc.Graph(figure=fig, id="new-data-graph"),
    ]
)


@callback(Output("new-data-graph", "figure"), Input("candidate-select", "value"))
def update_figure(value):
    patched_fig = Patch()
    patched_fig["data"][0]["y"] = df[value].values
    return patched_fig


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

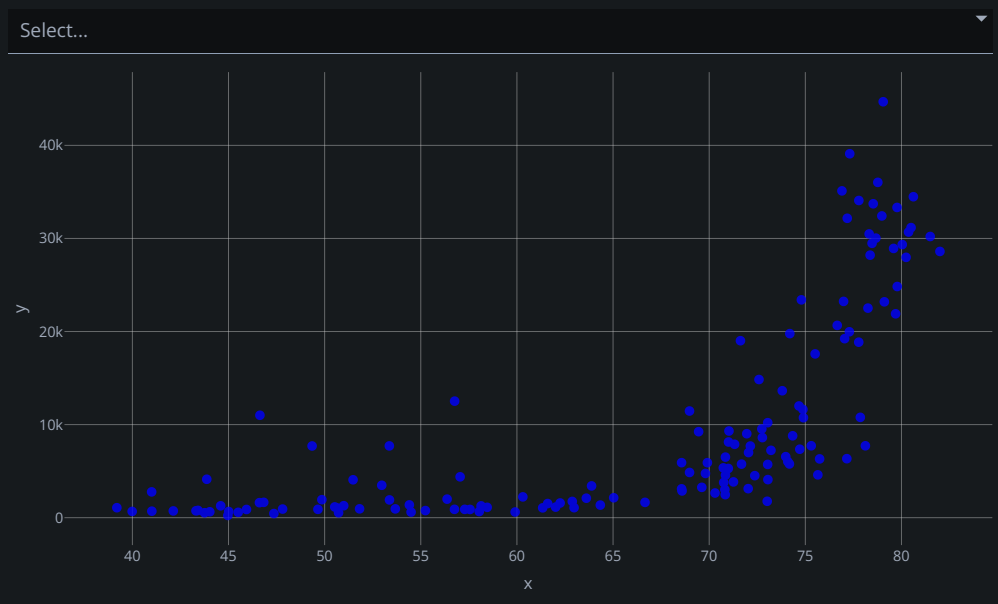


Here's another example. Here we update the marker color based on the selected values in a dropdown. In this example, we check which values the user has selected and then return a list with the color of those data points as red and any other data points as blue.

```
from dash import Dash, dcc, html, Input, Output, Patch, callback
import plotly.express as px

app = Dash()

# Getting our data
df = px.data.gapminder()
df = df.loc[df.year == 2002].reset_index()

# Creating our figure
fig = px.scatter(x=df.lifeExp, y=df.gdpPercap, hover_name=df.country)
fig.update_traces(marker=dict(color="blue"))

app.layout = html.Div(
    [
        html.H4("Updating Point Colors"),
        dcc.Dropdown(id="dropdown", options=df.country.unique(), multi=True),
        dcc.Graph(id="graph-update-example", figure=fig),
    ]
```

```
)


@callback(
    Output("graph-update-example", "figure"), Input("dropdown", "value"), prevent_initial_call=
)
def update_markers(countries):
    country_count = list(df[df.country.isin(countries)].index)
    patched_figure = Patch()
    updated_markers = [
        "red" if i in country_count else "blue" for i in range(len(df) + 1)
    ]
    patched_figure['data'][0]['marker']['color'] = updated_markers
    return patched_figure

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

Updating Point Colors



Append

`Patch` has an `append` method you can use to add to property attributes that are lists. It works like appending to a list in Python, adding the item to the end. It is useful for adding to a component's children and for adding data to axes on a figure.

Example with X and Y Axes of a Graph

By appending to the X and Y data arrays of a graph, we can add additional data points. In this example, we start with a graph with no data and append to both the X and Y axes each time the button is selected.

```
from dash import Dash, html, dcc, Input, Output, Patch, callback
import plotly.graph_objects as go

import datetime
import random

app = Dash()

fig = go.Figure()

app.layout = html.Div(
    [
        html.Button("Append", id="append-new-val"),
```

```
        dcc.Graph(figure=fig, id="append-example-graph"),
    ]
)

@callback(
    Output("append-example-graph", "figure"),
    Input("append-new-val", "n_clicks"),
    prevent_initial_call=True,
)
def add_data_to_fig(n_clicks):
    current_time = datetime.datetime.now()
    random_value = random.randrange(1, 30, 1)
    patched_figure = Patch()
    patched_figure["data"][0]["x"].append(current_time)
    patched_figure["data"][0]["y"].append(random_value)
    return patched_figure


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

APPEND

Example of Append with Pattern-Matching Callbacks

With **Pattern-Matching Callbacks**, we can add content dynamically to our layout. Often, we'll want to add to one component's `children`. In this example, the `children` of `'dropdown-container-2'` starts out as an empty list. Each time the `display_dropdowns` callback runs, a new dropdown is appended to the `children` of `'dropdown-container-2'` using a `Patch` object.

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

app = Dash(__name__, suppress_callback_exceptions=True)

app.layout = html.Div(
    [
        html.Button("Add Filter", id="add-filter-2", n_clicks=0),
        html.Div(id="dropdown-container-2", children=[]),
        html.Div(id="dropdown-container-output-2"),
    ]
)


@callback(
    Output("dropdown-container-2", "children"),
    Input("add-filter-2", "n_clicks"),
)
def display_dropdowns(n_clicks):
    patched_children = Patch()
    new_dropdown = dcc.Dropdown(
```

```
        ["NYC", "MTL", "LA", "TOKYO"],
        id={"type": "filter-dropdown-2", "index": n_clicks},
    )
    patched_children.append(new_dropdown)
    return patched_children


@callback(
    Output("dropdown-container-output-2", "children"),
    Input({"type": "filter-dropdown-2", "index": ALL}, "value"),
)
def display_output(values):
    return html.Div(
        [
            html.Div("Dropdown {} = {}".format(i + 1, value))
            for (i, value) in enumerate(values)
        ]
    )

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

ADD FILTER

Select...

Dropdown 1 = None

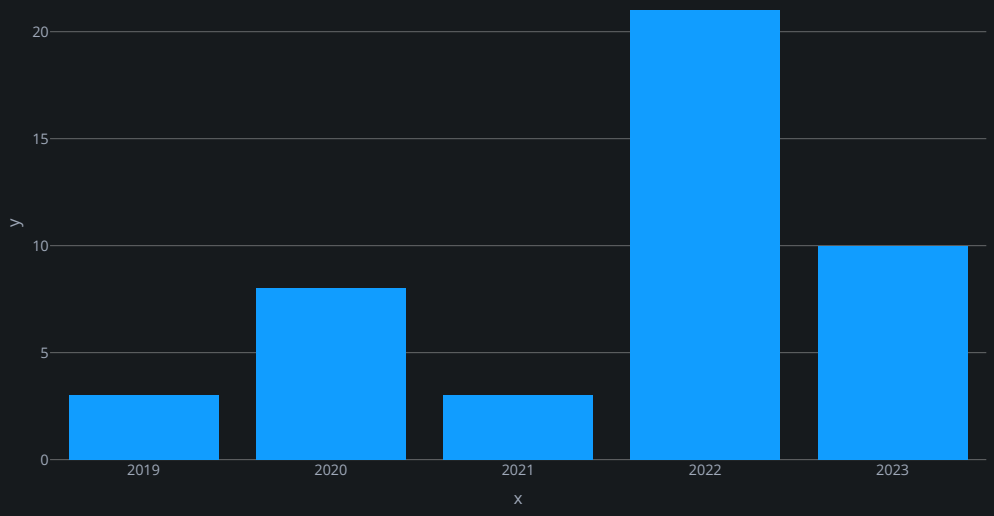
For additional examples, see the **Pattern-Matching Callbacks** page.

Prepend

The `prepend` method adds the provided value to the start of the list. Here we prepend to both the X and Y axes.

```
from dash import Dash, html, dcc, Input, Output, Patch, callback
import plotly.express as px

import random

app = Dash()

# Creating some starter x and y data
x_values = [2019, 2020, 2021, 2022, 2023]
y_values = [random.randrange(1, 30, 1) for i in range(5)]

fig = px.bar(x=x_values, y=y_values)

app.layout = html.Div(
    [
        html.Button("Prepend", id="prepend-new-val"),
        dcc.Graph(figure=fig, id="prepend-example-graph"),
    ]
)


@callback(
    Output("prepend-example-graph", "figure"),
    Input("prepend-new-val", "n_clicks"),
    prevent_initial_call=True,
)
def add_data_to_fig(n_clicks):
    random_value = random.randrange(1, 30, 1)
    patched_figure = Patch()
    patched_figure["data"][0]["x"].prepend(2019 - n_clicks)
    patched_figure["data"][0]["y"].prepend(random_value)
    return patched_figure


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

PREPEND

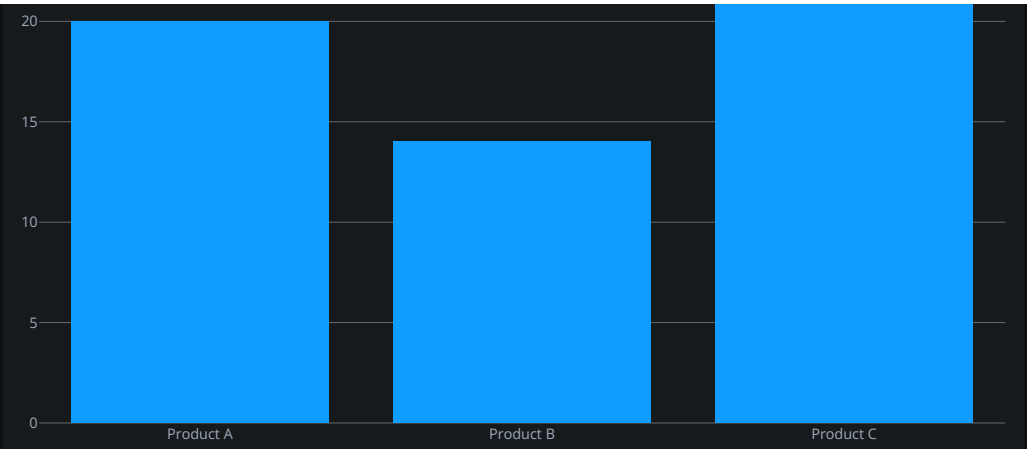


Extend

The `extend` method works like extending a list in Python. Use it by providing an iterable whose values will be added to the end of the list.

```
from dash import Dash, html, dcc, Input, Output, Patch, callback
from plotly import graph_objects as go

app = Dash()

x = ["Product A", "Product B", "Product C"]
y = [20, 14, 23]

additional_products_x = ["Product D", "Product E", "Product F"]
additional_products_y = [10, 24, 8]

fig = go.Figure(data=[go.Bar(x=x, y=y)])

app.layout = html.Div(
    [
        html.Button("Update products", id="additional-products"),
        dcc.Graph(figure=fig, id="extend-example"),
    ]
)


@callback(
    Output("extend-example", "figure"),
    Input("additional-products", "n_clicks"),
    prevent_initial_call=True,
)
def add_data_to_fig(n_clicks):
    if n_clicks % 2 != 0:
        patched_figure = Patch()
        patched_figure["data"][0]["x"].extend(additional_products_x)
        patched_figure["data"][0]["y"].extend(additional_products_y)
        return patched_figure
    else:
        return fig

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

UPDATE PRODUCTS



Maximum Number of Values

Each time you `extend` a list, it adds to the existing data. It's not currently possible to limit the maximum number of values in the list. This is something we hope to support in the future. To do this currently, you'll need to use assignment and replace the entire list with a new one with the exact number of values you want.

Example with a Dataframe

Here's another example of using `extend`. In this example, we add rows from a dataframe to a `Datatable`'s data on each click (after 10 clicks we stop updating the table).

```
from dash import Dash, html, Input, Output, Patch, dash_table, no_update, callback
import plotly.express as px

app = Dash()

df = px.data.iris()

app.layout = html.Div(
    [
        html.Button("Add Rows", id="add-data-rows"),
        dash_table.DataTable(
            data=df.to_dict("records"),
            columns=[{"name": i, "id": i} for i in df.columns],
            page_size=10,
            id="df-table",
        ),
    ]
)


@callback(
    Output("df-table", "data"),
    Input("add-data-rows", "n_clicks"),
    prevent_initial_call=True,
)
def add_data_to_fig(n_clicks):
    if n_clicks < 10:
        patched_table = Patch()
        patched_table.extend(df.to_dict("records"))
        return patched_table
    else:
        return no_update

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

ADD ROWS

sepal_length	sepal_width	petal_length	petal_width	species	species_id
5.1	3.5	1.4	0.2	setosa	1
4.9	3	1.4	0.2	setosa	1

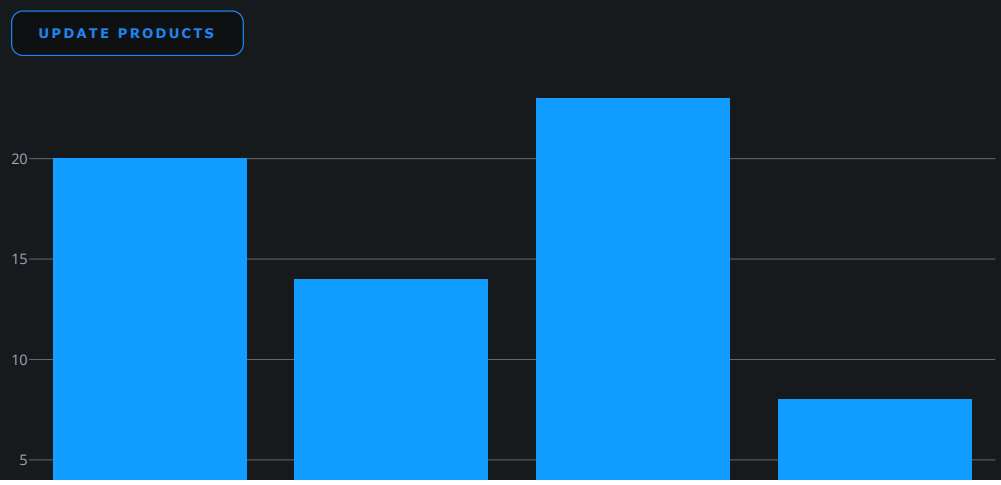
4.7	3.2	1.3	0.2	setosa	1
4.6	3.1	1.5	0.2	setosa	1
5	3.6	1.4	0.2	setosa	1
5.4	3.9	1.7	0.4	setosa	1
4.6	3.4	1.4	0.3	setosa	1
5	3.4	1.5	0.2	setosa	1
4.4	2.9	1.4	0.2	setosa	1
4.9	3.1	1.5	0.1	setosa	1

« ‹ 1 / 15 › »

Insert

Use `insert` to add to a list at a specific index. The `insert` method takes two arguments: the index to insert at and the data to add. In this example, we add "Product B" after "Product A" by inserting its X and Y data at index 1.

```
from dash import Dash, html, dcc, Input, Output, Patch, callback
from plotly import graph_objects as go

app = Dash()

x = ["Product A", "Product C", "Product D", "Product E",]
y = [20, 14, 23, 8]

fig = go.Figure(data=[go.Bar(x=x, y=y)])

app.layout = html.Div(
    [
        html.Button("Update products", id="additional-product-insert"),
        dcc.Graph(figure=fig, id="insert-example"),
    ]
)


@callback(
    Output("insert-example", "figure"),
    Input("additional-product-insert", "n_clicks"),
    prevent_initial_call=True,
)
def add_data_to_fig(n_clicks):
    if n_clicks % 2 != 0:
        patched_figure = Patch()
        patched_figure["data"][0]["x"].insert(1, "Product B")
        patched_figure["data"][0]["y"].insert(1, 10)
        return patched_figure
    else:
        return fig

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

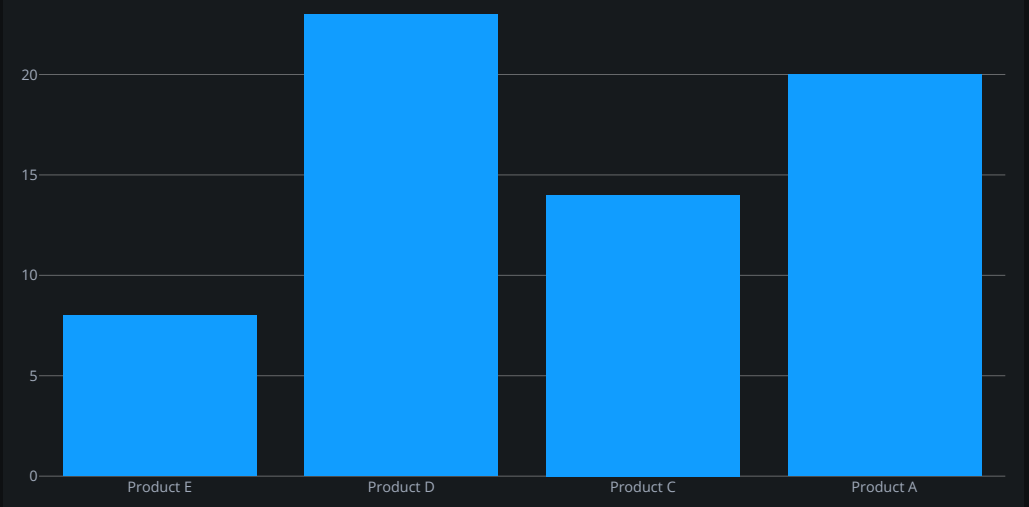


Reverse

Using `reverse` you can reverse the order of items in a list. Here, we reverse the data on the X and Y axes of the figure.

```
from dash import Dash, html, dcc, Input, Output, Patch, callback
from plotly import graph_objects as go

app = Dash()

x = ["Product A", "Product C", "Product D", "Product E",]
y = [20, 14, 23, 8]

fig = go.Figure(data=[go.Bar(x=x, y=y)])

app.layout = html.Div(
    [
        html.Button("Reverse Items", id="reverse-button"),
        dcc.Graph(figure=fig, id="reverse-example"),
    ]
)


@callback(
    Output("reverse-example", "figure"),
    Input("reverse-button", "n_clicks")
)
def add_data_to_fig(n_clicks):
    patched_figure = Patch()
    patched_figure["data"][0]["x"].reverse()
    patched_figure["data"][0]["y"].reverse()
    return patched_figure

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

REVERSE ITEMS



Clear

Use `clear` to remove all items in a list. `annotations` is a list of dictionaries on the graph's `layout`. In this example, we clear the list when the button is selected. The next time the button is selected we add the

`annotations` back.

```
from dash import Dash, html, dcc, Input, Output, Patch, callback
import plotly.graph_objects as go

app = Dash()

fig = go.Figure(
    [
        go.Scatter(x=[0, 1, 2, 3, 4, 5, 6, 7, 8], y=[0, 1, 3, 2, 4, 3, 4, 6, 5]),
        go.Scatter(x=[0, 1, 2, 3, 4, 5, 6, 7, 8], y=[0, 4, 5, 1, 2, 2, 3, 4, 2]),
    ],
    go.Layout(
        dict(
            annotations=[
                dict(
                    x=2,
                    y=5,
                    text="Text annotation with arrow",
                    showarrow=True,
                    arrowhead=1,
                ),
                dict(
                    x=4,
                    y=4,
                    text="Text annotation without arrow",
                    showarrow=False,
                    yshift=10,
                ),
            ]
        ),
        showlegend=False,
    ),
)

app.layout = html.Div(
    [
        html.Button("Show/Clear Annotations", id="clear-button"),
        dcc.Graph(id="clear-example", figure=fig),
    ]
)


@callback(Output("clear-example", "figure"), Input("clear-button", "n_clicks"))
def add_data_to_fig(n_clicks):
    patched_figure = Patch()
    if n_clicks and n_clicks % 2 != 0:
        patched_figure["layout"]["annotations"].clear()
    else:
        patched_figure["layout"]["annotations"].extend(
            [
                dict(
                    x=2,
                    y=5,
```

SHOW/CLEAR ANNOTATIONS

Update

For a property attribute that is a dictionary, you can use the `update` method to merge another dictionary into it. In this example, the `RadioItems` component's options are created as a dictionary. The initial dictionary has three key-value pairs, and when the button is selected, three additional key-value pairs are merged into it.

Note: `update` performs a single-level merge, not a deep merge.

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

app = Dash()

app.layout = html.Div(
    [
        html.Button("Add options", id="add-options"),
        dcc.RadioItems(
            options={
                "New York City": "New York City",
                "Montreal": "Montreal",
                "San Francisco": "San Francisco",
            },
            value="Montreal",
            id="city-dd",
        ),
        html.Div(id="city-output-container"),
    ]
)


@callback(Output("city-output-container", "children"), Input("city-dd", "value"))
def update_output(value):
    return f"You have selected {value}"


@callback(
    Output("city-dd", "options"),
    Input("add-options", "n_clicks"),
    prevent_initial_call=True,
)
def update_output(n_clicks):
    patched_dropdown = Patch()
    european_cities = {"Paris": "Paris", "London": "London", "Berlin": "Berlin"}
    patched_dropdown.update(european_cities)
    return patched_dropdown

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

ADD OPTIONS

- New York City
- Montreal
- San Francisco

You have selected Montreal

Delete

You can also make partial updates that delete parts of a property's data. In this example, we delete the first row of the table by deleting the element in `data` at index 0:

```
from dash import Dash, dash_table, html, Input, Output, Patch, callback
import pandas as pd

df = pd.read_csv("https://raw.githubusercontent.com/plotly/datasets/master/solar.csv")
```

```
app = Dash()

app.layout = html.Div(
    [
        html.Button("Delete first row", id="delete-button-1"),
        dash_table.DataTable(
            df.to_dict("records"),
            [{"name": i, "id": i} for i in df.columns],
            id="table-example-for-delete-1",
        ),
    ]
)

# Deleting row at index 0 in the data when the delete button is clicked
@callback(
    Output("table-example-for-delete-1", "data"),
    Input("delete-button-1", "n_clicks")
)
def delete_records(n_clicks):
    patched_table = Patch()
    del patched_table[0]
    return patched_table

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

DELETE FIRST ROW

State	Number of Solar Plants	Installed Capacity (MW)	Average MW Per Plant	Generation (GWh)
Arizona	48	1078	22.5	2550
Nevada	11	238	21.6	557
New Mexico	33	261	7.9	590
Colorado	20	118	5.9	235
Texas	12	187	15.6	354
North Carolina	148	669	4.5	1162
New York	13	53	4.1	84

Remove

You can also make partial updates that `remove` parts of a property's data that matches a given value. In this example, when the button is selected, we remove all items in the `Checklist` that are in the list `canandian_cities`.

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

app = Dash()

app.layout = html.Div(
    [
        html.Button("Remove items", id="remove-button"),
        dcc.Checklist(id="checklist-remove-items"),
    ]
)

@callback(
    Output("checklist-remove-items", "options"),
    Input("remove-button", "n_clicks"),
)
def remove_records(n_clicks):
    if not n_clicks:
        return [
            "Boston",
            "Montreal",
            "New York",
            "Toronto",
            "San Francisco",
            "Vancouver",
```

```
        ]
    else:
        canadian_cities = ["Montreal", "Toronto", "Vancouver"]
        patched_list = Patch()
        for x in canadian_cities:
            patched_list.remove(x)
        return patched_list


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

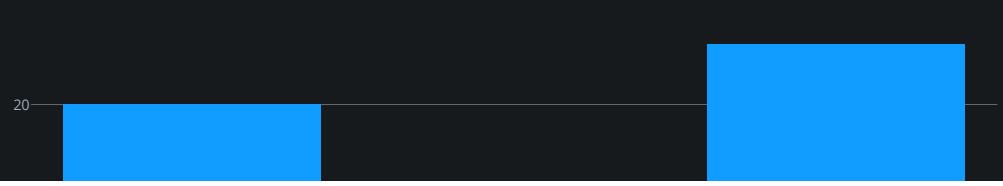
REMOVE ITEMS

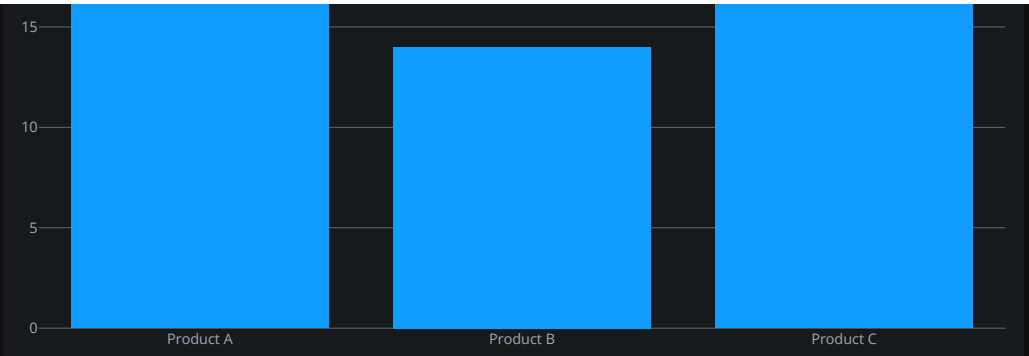
- [ ] Boston
- [ ] Montreal
- [ ] New York
- [ ] Toronto
- [ ] San Francisco
- [ ] Vancouver

Math Operations

`Patch` also supports math operations, for example, to increment, decrement, multiply, and divide values. In this example, we increment the value on the y-axis when a bar is selected.

```
from dash import Dash, html, dcc, Input, Output, Patch, callback
from plotly import graph_objects as go

app = Dash()

x = ["Product A", "Product B", "Product C"]
y = [20, 14, 23]

fig = go.Figure(data=[go.Bar(x=x, y=y)])

app.layout = html.Div(
    [
        dcc.Graph(figure=fig, id="increment-example-graph"),
    ]
)


@callback(
    Output("increment-example-graph", "figure"),
    Input("increment-example-graph", "clickData"),
    prevent_initial_call=True,
)
def check_selected_data(click_data):
    selected_product = click_data["points"][0]["label"]
    patched_figure = Patch()
    if selected_product == "Product A":
        patched_figure["data"][0]["y"][0] += 1
    elif selected_product == "Product B":
        patched_figure["data"][0]["y"][1] += 1
    elif selected_product == "Product C":
        patched_figure["data"][0]["y"][2] += 1
    return patched_figure


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





Method Summary

- **There are multiple ways you can use Patch objects to make partial updates:**

  Lists: `prepend`, `extend`, `append`, `reverse`, `insert`, `clear`, `remove`, and assignment using `=`

  Dictionaries: `update`, and assignment using `=`

  Strings: assignment using `=`

  Numbers: assignment using `=`

- **The location of the update can be specified with square bracket or dot notation:** For example, both of these are valid:

```
patched_figure = Patch()
patched_figure['layout']['title'] = 'New Title'
```

```
patched_figure = Patch()
patched_figure.layout.title = 'New Title'
```

- **But dot notation does not work for list indices, as it can only be used with valid Python identifiers.**

```
patched_figure = Patch()
patched_figure['data'][0]['y'] = [1, 2, 3]
```

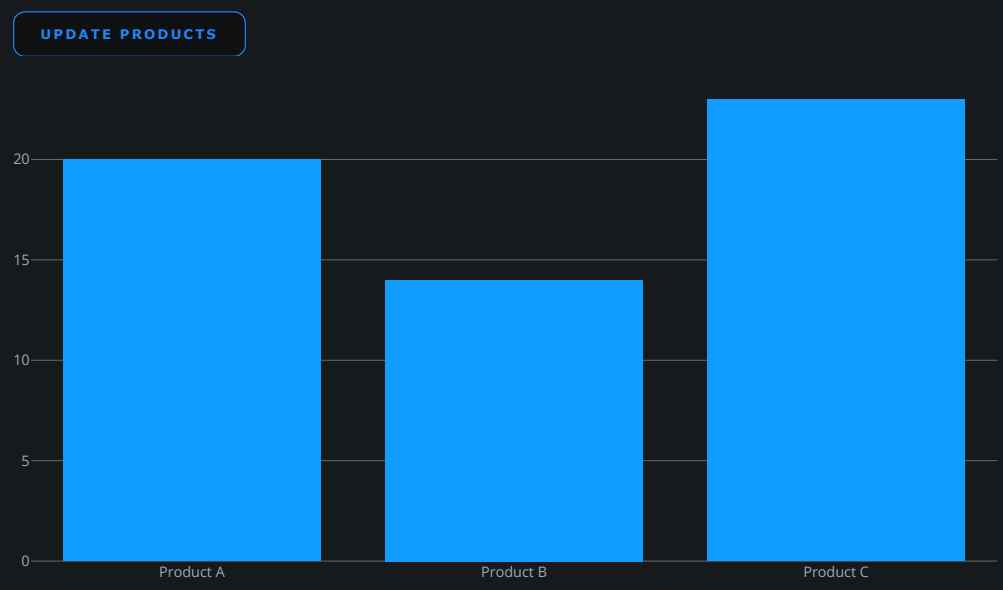
  Replacing `patched_figure['data'][0]['y'] = [1, 2, 3]` with `patched_figure.data.0.y = [1, 2, 3]` in the above example would not work.

Combining Patch Methods

You can combine the different `Patch` methods mentioned above. In this example, we use `prepend` and `append` with the same object to update the same attribute. The operations are applied in the order that they are defined in the callback.

```
from dash import Dash, html, dcc, Input, Output, Patch, callback
from plotly import graph_objects as go

app = Dash()

x = ["Product A", "Product B", "Product C"]
y = [20, 14, 23]

fig = go.Figure(data=[go.Bar(x=x, y=y)])

app.layout = html.Div(
    [
        html.Button("Update products", id="add-product-d-e"),
        dcc.Graph(figure=fig, id="prepend-append-example-graph"),
    ]
)
```

```
@callback(
    Output("prepend-append-example-graph", "figure"),
    Input("add-product-d-e", "n_clicks"),
    prevent_initial_call=True,
)
def add_data_to_fig(n_clicks):
    if n_clicks % 2 != 0:
        patched_figure = Patch()
        patched_figure["data"][0]["x"].prepend("Product D")
        patched_figure["data"][0]["y"].prepend(34)
        patched_figure["data"][0]["x"].append("Product E")
        patched_figure["data"][0]["y"].append(34)
        return patched_figure
    else:
        return fig

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



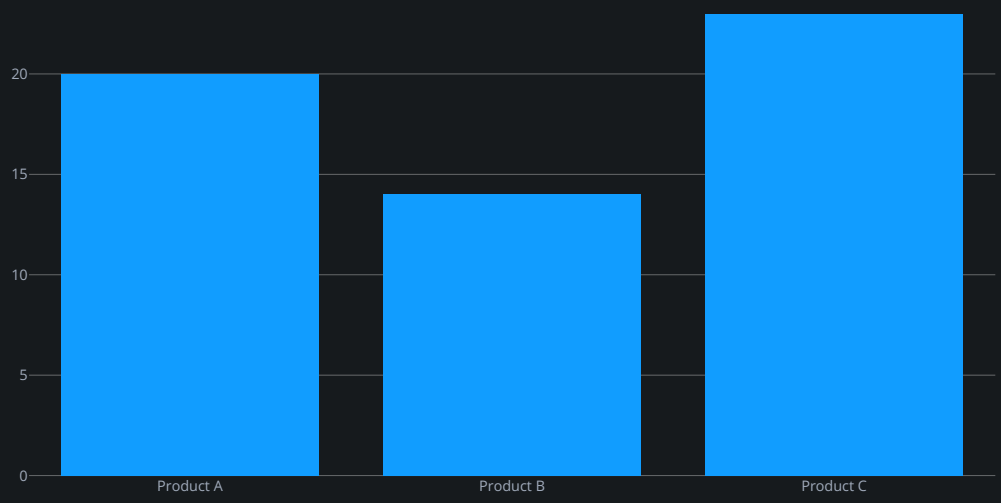
Using Patches on Multiple Outputs

You can use multiple `Patch` objects within a callback. In this example, we create one `Patch` object, `patched_figure` to append to the figure data, and we create another `Patch` object, `patched_table`, to update the `DataTable`'s `data` property.

```
from dash import Dash, html, dcc, Input, Output, Patch, dash_table, callback
from plotly import graph_objects as go

app = Dash()

# Initial data for the figure and datatable
x = ["Product A", "Product B", "Product C"]
y = [20, 14, 23]

table_data = [{"Product": x_value, "Value": y_value} for x_value, y_value in zip(x, y)]

# Additional data for the figure and datatable
additional_products_x = ["Product D", "Product E", "Product F"]
additional_products_y = [10, 24, 8]

fig = go.Figure(data=[go.Bar(x=x, y=y)])

app.layout = html.Div(
    [
        html.Button("Update Products", id="add-additional-products"),
        dcc.Graph(figure=fig, id="multiple-outputs-fig"),
```

```
        dash_table.DataTable(data=table_data, id="multiple-outputs-table"),
    ]
)


@callback(
    Output("multiple-outputs-fig", "figure"),
    Output("multiple-outputs-table", "data"),
    Input("add-additional-products", "n_clicks"),
    prevent_initial_call=True,
)
def add_data_to_fig(n_clicks):
    if n_clicks % 2 != 0:
        patched_figure = Patch()
        patched_table = Patch()

        additional_table_data = [
            {"Product": x_value, "Value": y_value}
            for x_value, y_value in zip(additional_products_x, additional_products_y)
        ]

        patched_table.extend(additional_table_data)

        patched_figure["data"][0]["x"].extend(additional_products_x)
        patched_figure["data"][0]["y"].extend(additional_products_y)

        return patched_figure, patched_table
    else:
        return fig, table_data
```

UPDATE PRODUCTS



Product	Value
Product A	20
Product B	14
Product C	23

Allowing Duplicate Callback Outputs

Sometimes you'll want to update the same component-property pair from multiple callback outputs. For example, you could have one callback output to update the color of a graph, and another callback output to update the data. You can do this by setting `allow_duplicate=True` on any outputs that are used more than once.

You can also use `allow_duplicate` to do full updates on one callback output and a partial update on another. In this example, clicking one button deletes the first row of the data. The second button sends the full data back to the component.

> When using duplicate callback outputs (with `allow_duplicate=True`), the order in which callbacks that run at the same time are updated is not guaranteed. See **Duplicate Callback Outputs**.

```python
from dash import Dash, dash_table, html, Input, Output, Patch, callback
import pandas as pd

df = pd.read_csv("https://raw.githubusercontent.com/plotly/datasets/master/solar.csv")

app = Dash()

app.layout = html.Div(
    [
        html.Button("Delete first row", id="delete-button"),
        html.Button("Reload data", id="reload-button"),
        dash_table.DataTable(
            df.to_dict("records"),
            [{"name": i, "id": i} for i in df.columns],
            id="table-example-for-delete",
        ),
    ]
)

# Returning all records from the dataframe to the component when the reload button is clicked
@callback(
    Output("table-example-for-delete", "data"),
    Input("reload-button", "n_clicks")
)
def reload_data(n_clicks):
    return df.to_dict("records")

# Deleting row at index 0 in the data when the delete button is clicked
@callback(
    Output("table-example-for-delete", "data", allow_duplicate=True),
    Input("delete-button", "n_clicks"),
    prevent_initial_call=True
)
def delete_records(n_clicks):
    patched_table = Patch()
    del patched_table[0]
    return patched_table

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

DELETE FIRST ROW RELOAD DATA

State	Number of Solar Plants	Installed Capacity (MW)	Average MW Per Plant	Generation (GWh)
California	289	4395	15.3	10826
Arizona	48	1078	22.5	2550
Nevada	11	238	21.6	557
New Mexico	33	261	7.9	590
Colorado	20	118	5.9	235
Texas	12	187	15.6	354
North Carolina	148	669	4.5	1162
New York	13	53	4.1	84

Exploring the Structure of Properties

To make partial updates to a property, you need to know the structure of the property you want to update. To understand the structure of any component property, check out the reference docs for it. For example, you'll find the reference docs for each Dash Core Component at the end of the component's page. For example, **the Dropdown properties**.

Graph Objects

As we've seen in the examples above, a great use for partial updates is updating individual parts of Plotly.py Graph Objects `Figure` objects, which you can pass to the `figure` parameter of a `dcc.Graph` component.

Often you'll only want to update a small detail on a graph, such as a color, and avoid sending all the graph's data back to the browser.

To do this successfully, you'll need to understand the structure of a `Figure`.

Here is an example of the structure of a simple `Figure` object.

```
Figure({
    'data': [{'hovertemplate': 'x=%{x}<br>y=%{y}<extra></extra>',
              'legendgroup': '',
              'line': {'color': '#636efa', 'dash': 'solid'},
              'marker': {'symbol': 'circle'},
              'mode': 'lines',
              'name': '',
              'orientation': 'v',
              'showlegend': False,
              'type': 'scatter',
              'x': array(['a', 'b', 'c'], dtype=object),
              'xaxis': 'x',
              'y': array([1, 3, 2]),
              'yaxis': 'y'}],
    'layout': {'legend': {'tracegroupgap': 0},
               'template': '...',
               'title': {'font': {'color': 'red'}, 'text': 'sample figure'},
               'xaxis': {'anchor': 'y', 'domain': [0.0, 1.0], 'title': {'text': 'x'}},
               'yaxis': {'anchor': 'x', 'domain': [0.0, 1.0], 'title': {'text': 'y'}}}
})
```

In our first `Patch` example above, we assigned a color to the title.

```
patched_figure['layout']['title']['font']['color'] = new_color
```

Similarly, we could update the title's text, which is currently set to `sample figure`:

```
patched_figure['layout']['title']['text'] = "my new title text"
```

When working with a `Figure`, you can see its structure by printing it.

```
fig = px.line(x=["a","b","c"], y=[1,3,2], title="sample figure")

fig.update_layout(
    title_font_color="red"
)
print(fig)
```

See the **The Figure Data Structure in Python** page in the Graphing Library docs for more details.

Limitations

- A `Patch` object is a representation of the operation to apply to part of an output property. It doesn't give you access to a property's values. For example, the following won't work:

  ```
  patch_output = Patch()
  formatted_property = f"My prop: {patch_output["my_prop"]}"
  ```

  If you need access to the property's value, use **State** on your callback.

- `Patch` is not available on clientside callbacks.

- Each time you `extend` a list, it adds to the existing data. It's not currently possible to limit the maximum number of values in the list.

Dash Python **> *Partial Property Updates***