plotly

Star 23,446

Dash Python > ***Part 4. Sharing Data Between Callbacks***

Sharing Data Between Callbacks

This is the final chapter of the essential **Dash Fundamentals**. The **previous chapter** covered how to use callbacks with the dcc.Graph component. The **rest of the Dash documentation** covers other topics like multi-page apps and component libraries. Just getting started? Make sure to **install the necessary dependencies**.

One of the core Dash principles explained in the **Getting Started Guide on Callbacks** is that **Dash Callbacks must never modify variables outside of their scope**. It is not safe to modify any global variables. This chapter explains why and provides some alternative patterns for sharing state between callbacks.

Why Share State?

In some apps, you may have multiple callbacks that depend on expensive data processing tasks like making database queries, running simulations, or downloading data.

Rather than have each callback run the same expensive task, you can have one callback run the task and then share the results to the other callbacks.

One way to achieve this is by having **multiple outputs** for one callback: the expensive task can be done once and immediately used in all the outputs. For example, if some data needs to be queried from a database and then displayed in both a graph and a table, then you can have one callback that calculates the data and creates both the graph and the table outputs.

But sometimes having multiple outputs in one callback isn't a good solution. For example, suppose your Dash app allows a user to select a date and a temperature unit (Fahrenheit or Celcius), and then displays the temperature for that day. You could have one callback that outputs the temperature by taking both the date and the temperature unit as inputs, but this means that if the user merely changes from Fahrenheit to Celcius then the weather data would have to be re-downloaded, which can be time consuming. Instead, it can be more efficient to have two callbacks: one callback that fetches the weather data, and another callback that outputs the temperature based on the downloaded data. This way, when only the unit is changed, the data does not have to be downloaded again. This is an example of **sharing** a variable, or state, between callbacks.

Dash is Stateless

Dash was designed to be a **stateless** framework.

Stateless frameworks are more scalable and robust than stateful ones. Most websites that you visit are running on stateless servers.

They are more scalable because it's trivial to add more compute power to the application. In order to scale the application to serve more users or run more computations, run more "copies" of the app in separate processes.

In production, this can be done either with gunicorn's worker command:

```
gunicorn app:server --workers 8
```

or by running the app in multiple Docker containers or servers and load balancing between them.

Stateless frameworks are more robust because even if one process fails, other processes can continue serving requests. In Dash Enterprise Kubernetes, these containers can run on separate servers or even separate regions, providing resiliency against server failure.

With a stateless framework, user sessions are not mapped 1-1 with server processes. Each callback request can be executed on *any* of the available processes. `gunicorn` will check which process isn't busy running a callback and send the new callback request to that process. This means that a few processes can balance the requests of 10s or 100s of concurrent users so long as those requests aren't happening at *the exact same time* (they usually don't!).

Why Global Variables Will Break Your App

Dash is designed to work in multi-user environments where multiple people view the application at the same time and have **independent sessions**.

If your app uses and modifies a global variable, then one user's session could set the variable to some value which would affect the next user's session.

Dash is also designed to be able to run with **multiple workers** so that callbacks can be executed in parallel.

This is commonly done with `gunicorn` using syntax like

```
$ gunicorn --workers 4 app:server
```

(`app` refers to a file named `app.py` and `server` refers to a variable in that file named `server`: `server = app.server`).

When Dash apps run across multiple workers, their memory *is not shared*. This means that if you modify a global variable in one callback, that modification will not be applied to the other workers / processes.

Here is a sketch of an app that will *not work reliably* because the callback modifies a global variable, which is outside of its scope.

```
df = pd.DataFrame({
  'student_id' : range(1, 11),
  'score' : [1, 5, 2, 5, 2, 3, 1, 5, 1, 5]
})

app.layout = html.Div([
        dcc.Dropdown(list(range(1, 6)), 1, id='score'),
        'was scored by this many students:',
        html.Div(id='output'),
])

@callback(Output('output', 'children'), Input('score', 'value'))
def update_output(value):
        global df
        df = df[df['score'] == value]
        return len(df)
```

The callback returns the correct output the very first time it is called, but once the global `df` variable is modified, any subsequent callback that uses that dataframe is not using the original data anymore.

To improve this app, reassign the filtered dataframe to a new variable inside the callback as shown below, or follow one of the strategies outlined in the next parts of this guide.

```
df = pd.DataFrame({
  'student_id' : range(1, 11),
  'score' : [1, 5, 2, 5, 2, 3, 1, 5, 1, 5]
})

app.layout = html.Div([
    dcc.Dropdown(list(range(1, 6)), 1, id='score'),
        'was scored by this many students:',
        html.Div(id='output'),
])
```

```
@callback(Output('output', 'children'), Input('score', 'value'))
def update_output(value):
        filtered_df = df[df['score'] == value]
        return len(filtered_df)
```

Storing Shared Data

To share data safely across multiple processes or servers, we need to store the data somewhere that is accessible to each of the processes.

There are three places you can store this data:

- In the user's browser session, using **dcc.Store**
- On the disk (e.g. in a file or database)
- In server-side memory (RAM) shared across processes and servers such as a Redis database. Dash Enterprise includes **onboard, one-click Redis databases** for this purpose.

The following examples illustrate some of these approaches.

Example 1 - Storing Data in the Browser with `dcc.Store`

To save data in the user's browser's session:

- The data has to be converted to a string like JSON or base64 encoded binary data for storage
- Data that is cached in this way will *only be available in the user's current session*.
    - If you open up a new browser window, the app's callbacks will always re-compute the data. The data is only cached between callbacks within the same session.
    - This method doesn't increase the memory footprint of the app.
    - There could be a cost in network traffic. If you're sharing 10MB of data between callbacks, then that data will be transported over the network between each callback.
    - If the network cost is too high, then compute the aggregations upfront and transport those. Your app likely won't be displaying 10MB of data, it will just be displaying a subset or an aggregation of it.

The example below shows one of the common ways you can leverage `dcc.Store`: if processing a dataset takes a long time and different outputs use this dataset, `dcc.Store` can be used to store the processed data as an *intermediate value* that can then be used as an input in multiple callbacks to generate different outputs. This way, the expensive data processing step is only performed once in one callback instead of repeating the same expensive computation multiple times in each callback.

```
app.layout = html.Div([
    dcc.Graph(id='graph'),
    html.Table(id='table'),
    dcc.Dropdown(id='dropdown'),

    # dcc.Store stores the intermediate value
    dcc.Store(id='intermediate-value')
])

@callback(Output('intermediate-value', 'data'), Input('dropdown', 'value'))
def clean_data(value):
     # some expensive data processing step
     cleaned_df = slow_processing_step(value)

     # more generally, this line would be
     # json.dumps(cleaned_df)
     return cleaned_df.to_json(date_format='iso', orient='split')

@callback(Output('graph', 'figure'), Input('intermediate-value', 'data'))
def update_graph(jsonified_cleaned_data):

    # more generally, this line would be
```

```
    # json.loads(jsonified_cleaned_data)
    dff = pd.read_json(jsonified_cleaned_data, orient='split')

    figure = create_figure(dff)
    return figure

@callback(Output('table', 'children'), Input('intermediate-value', 'data'))
def update_table(jsonified_cleaned_data):
    dff = pd.read_json(jsonified_cleaned_data, orient='split')
    table = create_table(dff)
    return table
```

Notice that the data needs to be serialized into a JSON string before being placed in storage. Also note how the processed data gets stored in `dcc.Store` by assigning the data as its output, and then the same data gets used by multiple callbacks by using the same `dcc.Store` as an input.

▶ Note about a previous version of this example

Example 2 - Computing Aggregations Upfront

Sending the computed data over the network can be expensive if the data is large. In some cases, serializing this data to JSON can also be expensive.

In many cases, your app will only display a subset or an aggregation of the processed data. In these cases, you could precompute the aggregations in your data processing callback and transport these aggregations to the remaining callbacks.

Here's a simple example of how you might transport filtered or aggregated data to multiple callbacks, again using the same `dcc.Store`.

```
@callback(
    Output('intermediate-value', 'data'),
    Input('dropdown', 'value'))
def clean_data(value):
     cleaned_df = slow_processing_step(value)

     # a few filter steps that compute the data
     # as it's needed in the future callbacks
     df_1 = cleaned_df[cleaned_df['fruit'] == 'apples']
     df_2 = cleaned_df[cleaned_df['fruit'] == 'oranges']
     df_3 = cleaned_df[cleaned_df['fruit'] == 'figs']

     datasets = {
         'df_1': df_1.to_json(orient='split', date_format='iso'),
         'df_2': df_2.to_json(orient='split', date_format='iso'),
         'df_3': df_3.to_json(orient='split', date_format='iso'),
     }

     return json.dumps(datasets)

@callback(
    Output('graph1', 'figure'),
    Input('intermediate-value', 'data'))
def update_graph_1(jsonified_cleaned_data):
    datasets = json.loads(jsonified_cleaned_data)
    dff = pd.read_json(datasets['df_1'], orient='split')
    figure = create_figure_1(dff)
    return figure

@callback(
    Output('graph2', 'figure'),
    Input('intermediate-value', 'data'))
def update_graph_2(jsonified_cleaned_data):
    datasets = json.loads(jsonified_cleaned_data)
    dff = pd.read_json(datasets['df_2'], orient='split')
    figure = create_figure_2(dff)
    return figure

@callback(
    Output('graph3', 'figure'),
    Input('intermediate-value', 'data'))
```

```
def update_graph_3(jsonified_cleaned_data):
    datasets = json.loads(jsonified_cleaned_data)
    dff = pd.read_json(datasets['df_3'], orient='split')
    figure = create_figure_3(dff)
    return figure
```

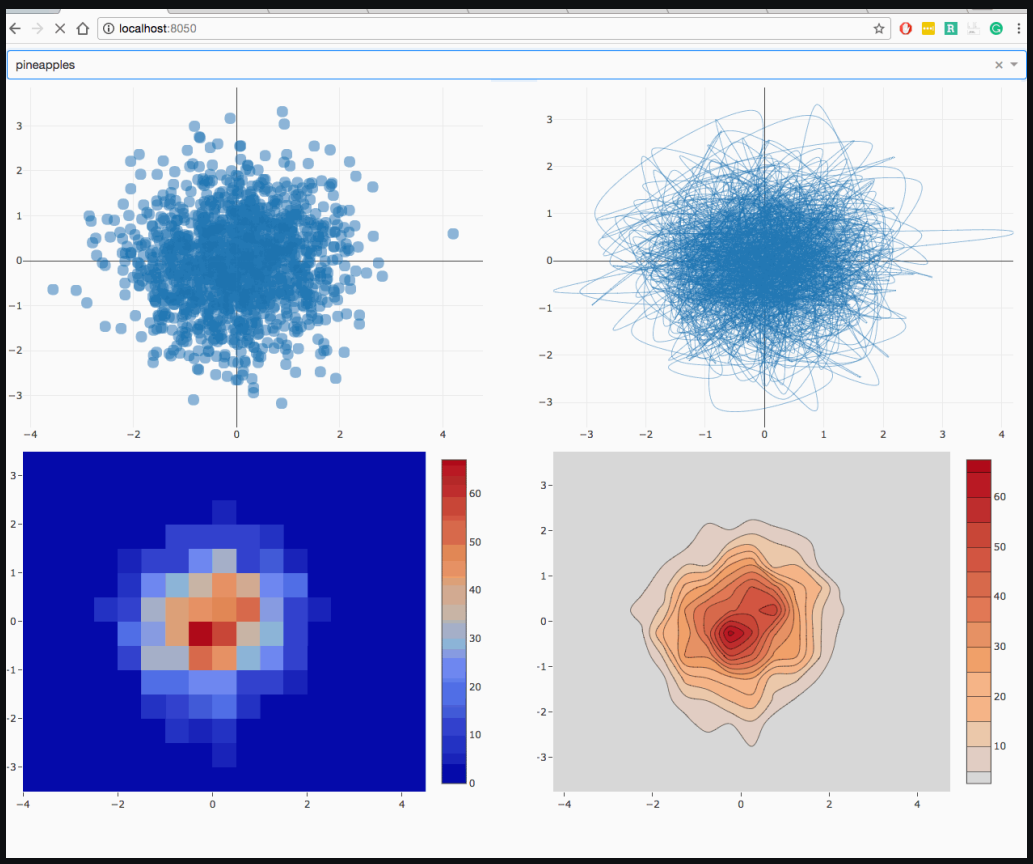
Example 3 - Caching and Signaling

This example:

- Uses Redis via Flask-Cache for storing "global variables" on the server-side in a database. This data is accessed through a function (`global_store()`), the output of which is cached and keyed by its input arguments.
- Uses the `dcc.Store` solution to send a signal to the other callbacks when the expensive computation is complete.
- Note that instead of Redis, you could also save this to the file system. See **https://flask-caching.readthedocs.io/en/latest/** for more details.
- This "signaling" is performant because it allows the expensive computation to only take up one process and be performed once. Without this type of signaling, each callback could end up computing the expensive computation in parallel, locking four processes instead of one.

Another benefit of this approach is that future sessions can use the pre-computed value. This will work well for apps that have a small number of inputs.

Here's what this example looks like.



▶ Here's what this example looks like in code:

Some things to note:

- We've simulated an expensive process by using a system sleep of 3 seconds.
- When the app loads, it takes three seconds to render all four graphs.
- The initial computation only blocks one process.
- Once the computation is complete, the signal is sent and four callbacks are executed in parallel to render the graphs. Each of these callbacks retrieves the data from the "global server-side store": the Redis or filesystem

cache.

- We've set `processes=6` in `app.run` so that multiple callbacks can be executed in parallel. In production, this is done with something like `$ gunicorn --workers 6 app:server`. If you don't run with multiple processes, then you won't see the graphs update in parallel as callbacks will be updated serially.
- As we are running the server with multiple processes, we set `threaded` to `False`. A Flask server can't be be both multi-process and multi-threaded.
- Selecting a value in the dropdown will take less than three seconds if it has already been selected in the past. This is because the value is being pulled from the cache.
- Similarly, reloading the page or opening the app in a new window is also fast because the initial state and the initial expensive computation has already been computed.

Example 4 - User-Based Session Data on the Server

The previous example cached computations in a way that was accessible for all users.

Sometimes you may want to keep the data isolated to user sessions: one user's derived data shouldn't update the next user's derived data. One way to do this is to save the data in a `dcc.Store`, as demonstrated in the first example.

Another way to do this is to save the data in a cache along with a session ID and then reference the data using that session ID. Because data is saved on the server instead of transported over the network, this method is generally faster than the `dcc.Store` method.

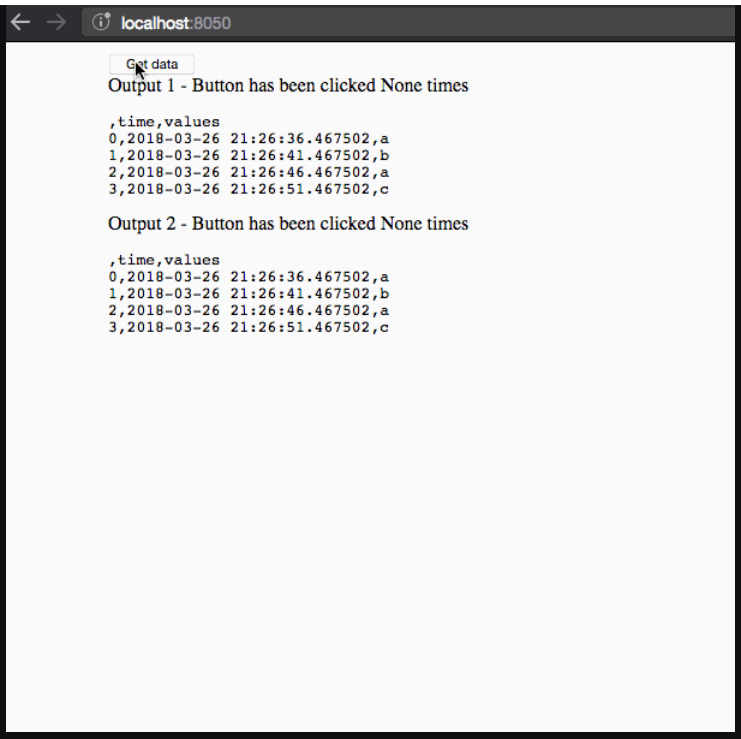
*This method was originally discussed in a **Dash Community Forum thread**.*

This example:

- Caches data using the `flask_caching` filesystem cache. You can also save to an in-memory cache or database such as Redis instead.
- Serializes the data as JSON.
    - If you are using Pandas, consider serializing with Apache Arrow for faster serialization or Plasma for smaller dataframe size. **Community thread**
- Saves session data up to the number of expected concurrent users. This prevents the cache from being overfilled with data.
- Creates unique session IDs for each session and stores it as the data of `dcc.Store` on every page load. This means that every user session has unique data in the `dcc.Store` on their page.

> Note: As with all examples that send data to the client, be aware that these sessions aren't necessarily secure or encrypted. These session IDs may be vulnerable to **Session Fixation** style attacks.

▶ Here's what this example looks like in code:



There are three things to notice in this example:

- The timestamps of the dataframe don't update when we retrieve the data. This data is cached as part of the user's session.
- Retrieving the data initially takes three seconds but successive queries are instant, as the data has been cached.
- The second session displays different data than the first session: the data that is shared between callbacks is isolated to individual user sessions.

Questions? Discuss these examples on the **Dash Community Forum**.

Dash Python ***> Part 4. Sharing Data Between Callbacks***