plotly

Star 23,446

Flexible Callback Signatures

Flexible callback signatures are new in Dash 2.0! To get the most out of this page, make sure you've read about **Basic Callbacks** in the Dash Fundamentals.

Using flexible callback signatures, introduced in Dash 2.0, you can write app code that is easier to manage by using keyword arguments, argument grouping using dicts or tuples, and by mixing `Input` and `State` dependencies objects.

In Dash 1, the `Input`, `State`, and `Output` dependency objects are always provided to `@app.callback` as positional arguments (either positional arguments directly to `@app.callback`, or as lists to the `inputs`, `state` and `output` keyword arguments). The order in which the dependency objects are provided dictates the order of the positional arguments that are passed to the decorated callback function. This means that the names of the callback function arguments don't matter, only the order they are defined in.

Positional Arguments

```
@callback(
    Output(...), Output(...),
    Input(...), Input(...),
    State(...)
)
def callback(a, b, c):
    return [a + b, b + c]
```

```
@callback(
    [Output(...), Output(...)],
    [Input(...), Input(...)],
    [State(...)]
)
def callback(a, b, c):
    return [a + b, b + c]
```

```
@callback(
    output=[Output(...), Output(...)],
    inputs=[Input(...), Input(...)],
    state=[State(...)]
)
def callback(a, b, c):
    return [a + b, b + c]
```

Keyword Arguments

Callback functions can register to be called with named keyword arguments. You can do this by passing dictionaries to the `inputs` and `state` arguments of `@callback`. In this case, the order of the callback's function arguments doesn't matter. All that matters is that the keys of the dependency dictionary match the function argument names.

Here is an example of using keyword `input` and `state` arguments:

```
@callback(
    output=[Output(...), Output(...)],
    inputs=dict(a=Input(...), b=Input(...)),
    state=dict(c=State(...))
)
def callback(b, c, a):
    return [a + b, b + c]
```

You can also specify the output of a callback function using named arguments. In this case the function is expected to return a dictionary with keys matching the keys in the dictionary passed to the `output` argument of `@callback`. For example:

```
@callback(
    output=dict(x=Output(...), y=Output(...)),
    inputs=dict(a=Input(...), b=Input(...)),
    state=dict(c=State(...))
)
def callback(b, c, a):
    return dict(x=a + b, y=b + c)
```

Interchangeable Input and State

Flexible Callback Signatures mean you can freely mix `Input` and `State` dependencies objects. This means that `State` dependencies can be included in the `inputs` argument, and `Input` dependencies can be included in the `state` argument. We recommended you put both `Input` and `State` dependencies in `inputs` rather than using the `state` keyword argument. For example:

```
@callback(
    output=dict(x=Output(...), y=Output(...)),
    inputs=dict(a=Input(...), b=Input(...), c=State(...)),
)
def callback(b, c, a):
    return dict(x=a + b, y=b + c)
```

Tuple and Dictionary Argument Grouping

You can also combine multiple `Input` and `State` dependency values into a single function argument.

Tuple Grouping

You can group dependency values in a tuple. Here the `ab` keyword function argument is a tuple consisting of the values of two `Input` dependency values.

```
@callback(
    output=[Output(...), Output(...)],
    inputs=dict(
        ab=(Input(...), Input(...)),
        c=Input(...)
    )
)
def callback(ab, c):
    a, b = ab
    return [a + b, b + c]
```

Or with positional indexing

```
@callback(
    output=[Output(...), Output(...)],
    inputs=[(Input(...), Input(...)), Input(...)]
)
```

```
def callback(ab, c):
    a, b = ab
    return [a + b, b + c]
```

Dictionary Grouping

Similarly, you can group multiple `Input` and `State` values together into a dictionary of values when passed to the function. Here, the `ab` argument is passed to the function as a dict containing `"a"` and `"b"` keys with values corresponding to the `Input` dependency values in the `@callback` specification.

```
@callback(
    output=[Output(...), Output(...)],
    inputs=dict(
        ab=dict(a=Input(...), b=Input(...)),
        c=Input(...)
    )
)
def callback(ab, c):
    a, b = ab["a"], ab["b"]
    return [a + b, b + c]
```

You can also nest these groupings arbitrarily deep.

```
@callback(
    output=[Output(...), Output(...)],
    args=dict(
        abc=dict(a=Input(...), b=(Input(...), Input(...)))
    )
)
def param_fn(abc):
    a, (b, c) = abc["a"], abc["b"]
    return [a + b, b + c]
```

Output Grouping

You can use the same tuple and dict groupings for the function output values as well.

Output Tuple Grouping

```
@callback(
    output=[Output(...), (Output(...), Output(...))],
    inputs=dict(
        a=Input(...),
        b=Input(...),
        c=Input(...),
    )
)
def callback(a, b, c):
    return [a, (a + b, b + c)]
```

Output Dict Grouping

```
@callback(
    output=[Output(...), dict(x=Output(...), y=Output(...))],
    inputs=dict(
        a=Input(...),
        b=Input(...),
        c=Input(...),
    )
)
```

```
def callback(a, b, c):
    return [a, dict(x=a+b, y=b+c)]
```

Callback Context with Flexible Callback Signatures

> `dash.callback_context.args_grouping` is new in Dash 2.4

`dash.callback_context.args_grouping` (or `dash.ctx.args_grouping`) is a dict of the inputs used with flexible callback signatures and helps simplify complex callbacks that have many inputs.

When a callback is triggered, we can use `args_grouping` to get the following information about our inputs:

- "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.

In the following example, in the callback, we capture the dictionary of inputs with `c = ctx.args_grouping.all_inputs`. We then check if `btn1` triggered the callback (by checking if `c.btn1.triggered` is True) and if it did we return `btn1`'s value: `c.btn1.value`. We do the same for `btn2` and `btn3`. Note how we only need to pass one argument to the `display` function.

```
@callback(
    Output("container", "children"),
    inputs={
        "all_inputs": {
            "btn1": Input("btn-1", "n_clicks"),
            "btn2": Input("btn-2", "n_clicks"),
            "btn3": Input("btn-3", "n_clicks")
        }
    },
)
def display(all_inputs):
    c = ctx.args_grouping.all_inputs
    if c.btn1.triggered:
        return f"Button 1 clicked {c.btn1.value} times"
    elif c.btn2.triggered:
        return f"Button 2 clicked {c.btn2.value} times"
    elif c.btn3.triggered:
        return f"Button 3 clicked {c.btn3.value} times"
```

Limitations

- Providing the `state` keyword argument to `@callback` is no longer supported unless the `inputs` keyword argument is also provided.
- Clientside callbacks are not supported.

Reference

app.callback API reference

Dash Python > ***Flexible Callback Signatures***