plotly

Star 18,978

Python

Dash Python > Dash DataTable > **Reference**

dash_table.DataTable

DataTable Properties

Access this documentation in your Python terminal with:

```
>>> help(dash.dash_table.DataTable)
```

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

`data` (*list of dicts with strings as keys and values of type string | number | boolean*; optional): The contents of the table. The keys of each item in data should match the column IDs. Each item can also have an 'id' key, whose value is its row ID. If there is a column with ID='id' this will display the row ID, otherwise it is just used to reference the row for selections, filtering, etc. Example: [{'column-1': 4.5, 'column-2': 'montreal', 'column-3': 'canada'}, {'column-1': 8, 'column-2': 'boston', 'column-3': 'america'}].

`columns` (*list of dicts*; optional): Columns describes various aspects about each individual column. `name` and `id` are the only required parameters.

`columns` is a list of dicts with keys:

- **`clearable`** (*a value equal to: 'first' or 'last' | boolean | list of booleans; optional*): If True, the user can clear the column by clicking on the `clear` action button on the column. If there are multiple header rows, True will display the action button on each row. If `last`, the `clear` action button will only appear on the last header row. If `first` it will only appear on the first header row. These are respectively shortcut equivalents to `[False, ..., False, True]` and `[True, False, ..., False]`. If there are merged, multi-header columns then you can choose which column header row to display the `clear` action button in by supplying an array of booleans. For example, `[True, False]` will display the `clear` action button on the first row, but not the second row. If the `clear` action button appears on a merged column, then clicking on that button will clear *all* of the merged columns associated with it. Unlike `column.deletable`, this action does not remove the column(s) from the table. It only removed the associated entries from `data`.

- **`deletable`** (*a value equal to: 'first' or 'last' | boolean | list of booleans*; optional): If True, the user can remove the column by clicking on the `delete` action button on the column. If there are multiple header rows, True will display the action button on each row. If `last`, the `delete` action button will only appear on the last header row. If `first` it will only appear on the first header row. These are respectively shortcut equivalents to `[False, ..., False, True]` and `[True, False, ..., False]`. If there are merged, multi-header columns then you can choose which column header row to display the `delete` action button in by supplying an array of booleans. For example, `[True, False]` will display the `delete` action button on the first row, but not the second row. If the `delete` action button appears on a merged column, then clicking on that button will remove *all* of the merged columns associated with it.

- **`editable`** (*boolean*; optional): There are two `editable` flags in the table. This is the column-level editable flag and there is also the table-level `editable` flag. These flags determine whether the contents of the table are editable or not. If the column-level `editable` flag is set it overrides the table-level `editable` flag for that column.

- **`filter_options`** (*dict*; optional): There are two `filter_options` props in the table. This is the column-level filter_options prop and there is also the table-level `filter_options` prop. If the column-level `filter_options` prop is set it overrides the table-level `filter_options` prop for that column.

`filter_options` is a dict with keys:

- **`case`** (*a value equal to: 'sensitive' or 'insensitive'*; optional): (default: 'sensitive') Determine whether the applicable filter relational operators will default to `sensitive` or `insensitive` comparison.

- **`placeholder_text`** (*string*; optional): (default: 'filter data...') The filter cell placeholder text.

- **`format`** (*dict*; optional): The formatting applied to the column's data. This prop is derived from the **d3-format** library specification. Apart from being structured slightly differently (under a single prop), the usage is the same. See also dash_table.FormatTemplate. It contains helper functions for typical number formats.

  `format` is a dict with keys:

  - **`locale`** (*dict*; optional): Represents localization specific formatting information. When left unspecified, will use the default value provided by d3-format.

    `locale` is a dict with keys:

    - **`decimal`** (*string*; optional): (default: '.'). The string used for the decimal separator.

    - **`group`** (*string*; optional): (default: ','). The string used for the groups separator.

    - **`grouping`** (*list of numbers*; optional): (default: [3]). A list of integers representing the grouping pattern. The default is 3 for thousands.

    - **`numerals`** (*list of strings*; optional): A list of ten strings used as replacements for numbers 0-9.

    - **`percent`** (*string*; optional): (default: '%'). The string used for the percentage symbol.

    - **`separate_4digits`** (*boolean*; optional): (default: True). Separates integers with 4-digits or less.

        - `symbol` (*list of strings*; optional): (default: ['$', '']). A list of two strings representing the prefix and suffix symbols. Typically used for currency, and implemented using d3's currency format, but you can use this for other symbols such as measurement units.

    - `nully` (*boolean* | *number* | *string* | *list* | *dict*; optional): A value that will be used in place of the Noney value during formatting. If the value type matches the column type, it will be formatted normally.

    - `prefix` (*number*; optional): A number representing the SI unit to use during formatting. See `dash_table.Format.Prefix` enumeration for the list of valid values.

    - `specifier` (*string*; optional): (default: ''). Represents the d3 rules to apply when formatting the number.

- `hideable` (*a value equal to: 'first' or 'last'* | *boolean* | *list of booleans*; optional): If True, the user can hide the column by clicking on the `hide` action button on the column. If there are multiple header rows, True will display the action button on each row. If `last`, the `hide` action button will only appear on the last header row. If `first` it will only appear on the first header row. These are respectively shortcut equivalents to `[False, ..., False, True]` and `[True, False, ..., False]`. If there are merged, multi-header columns then you can choose which column header row to display the `hide` action button in by supplying an array of booleans. For example, `[True, False]` will display the `hide` action button on the first row, but not the second row. If the `hide` action button appears on a merged column, then clicking on that button will hide *all* of the merged columns associated with it.

- `id` (*string*; required): The `id` of the column. The column `id` is used to match cells in data with particular columns. The `id` is not visible in the table.

- `name` (*string* | *list of strings*; required): The `name` of the column, as it appears in the column header. If `name` is a list of strings, then the columns will render with multiple headers rows.

- **`on_change`** (*dict*; optional): The `on_change` behavior of the column for user-initiated modifications.

  `on_change` is a dict with keys:

  - **`action`** (*a value equal to: 'coerce', 'none' or 'validate'*; optional): (default 'coerce'): 'none': do not validate data; 'coerce': check if the data corresponds to the destination type and attempts to coerce it into the destination type if not; 'validate': check if the data corresponds to the destination type (no coercion).

  - **`failure`** (*a value equal to: 'accept', 'default' or 'reject'*; optional): (default 'reject'): What to do with the value if the action fails. 'accept': use the invalid value; 'default': replace the provided value with `validation.default`; 'reject': do not modify the existing value.

- **`presentation`** (*a value equal to: 'input', 'dropdown' or 'markdown'*; optional): The `presentation` to use to display data. Markdown can be used on columns with type 'text'. See 'dropdown' for more info. Defaults to 'input' for ['datetime', 'numeric', 'text', 'any'].

- **`renamable`** (*a value equal to: 'first' or 'last' | boolean | list of booleans*; optional): If True, the user can rename the column by clicking on the `rename` action button on the column. If there are multiple header rows, True will display the action button on each row. If `last`, the `rename` action button will only appear on the last header row. If `first` it will only appear on the first header row. These are respectively shortcut equivalents to `[False, ..., False, True]` and `[True, False, ..., False]`. If there are merged, multi-header columns then you can choose which column header row to display the `rename` action button in by supplying an array of booleans. For example, `[True, False]` will display the `rename` action button on the first row, but not the second row. If the `rename` action button appears on a merged column, then clicking on that button will rename *all* of the merged columns associated with it.

- **`selectable`** (*a value equal to: 'first' or 'last' | boolean | list of booleans*;

optional): If True, the user can select the column by clicking on the checkbox or radio button in the column. If there are multiple header rows, True will display the input on each row. If `last`, the input will only appear on the last header row. If `first` it will only appear on the first header row. These are respectively shortcut equivalents to `[False, ..., False, True]` and `[True, False, ..., False]`. If there are merged, multi-header columns then you can choose which column header row to display the input in by supplying an array of booleans. For example, `[True, False]` will display the `selectable` input on the first row, but now on the second row. If the `selectable` input appears on a merged columns, then clicking on that input will select *all* of the merged columns associated with it. The table-level prop `column_selectable` is used to determine the type of column selection to use.

- **`sort_as_null`** (*list of strings | numbers | booleans*; optional): An array of string, number and boolean values that are treated as `None` (i.e. ignored and always displayed last) when sorting. This value overrides the table-level `sort_as_None`.

- **`type`** (*a value equal to: 'any', 'numeric', 'text' or 'datetime'*; optional): The data-type provides support for per column typing and allows for data validation and coercion. 'numeric': represents both floats and ints. 'text': represents a string. 'datetime': a string representing a date or date-time, in the form: 'YYYY-MM-DD HH:MM:SS.ssssss' or some truncation thereof. Years must have 4 digits, unless you use `validation.allow_YY: True`. Also accepts 'T' or 't' between date and time, and allows timezone info at the end. To convert these strings to Python `datetime` objects, use `dateutil.parser.isoparse`. In R use `parse_iso_8601` from the `parsedate` library. WARNING: these parsers do not work with 2-digit years, if you use `validation.allow_YY: True` and do not coerce to 4-digit years. And parsers that do work with 2-digit years may make a different guess about the century than we make on the front end. 'any': represents any type of data. Defaults to 'any' if undefined.

- **`validation`** (*dict*; optional): The `validation` options for user input processing that can accept, reject or apply a default value.

  `validation` is a dict with keys:

- **`allow_YY`** (*boolean*; optional): This is for `datetime` columns only. Allow 2-digit years (default: False). If True, we

  interpret years as ranging from now-70 to now+29 - in 2019 this is 1949 to 2048 but in 2020 it will be different. If used with `action: 'coerce'`, will convert user input to a 4-digit year.
- **`allow_null`** (*boolean*; optional): Allow the use of Noney values. (undefined, None, NaN) (default: False).
- **`default`** (*boolean* | *number* | *string* | *list* | *dict*; optional): The default value to apply with on_change.failure = 'default'. (default: None).

`editable` (*boolean*; default `False`): If True, then the data in all of the cells is editable. When `editable` is True, particular columns can be made uneditable by setting `editable` to `False` inside the `columns` property. If False, then the data in all of the cells is uneditable. When `editable` is False, particular columns can be made editable by setting `editable` to `True` inside the `columns` property.

`fixed_columns` (*dict*; default `{ headers: False, data: 0}`): `fixed_columns` will "fix" the set of columns so that they remain visible when scrolling horizontally across the unfixed columns. `fixed_columns` fixes columns from left-to-right. If `headers` is False, no columns are fixed. If `headers` is True, all operation columns (see `row_deletable` and `row_selectable`) are fixed. Additional data columns can be fixed by assigning a number to `data`. Note that fixing columns introduces some changes to the underlying markup of the table and may impact the way that your columns are rendered or sized. View the documentation examples to learn more.

`fixed_columns` is a dict with keys:

- **`data`** (*a value equal to: 0*; optional): Example `{'headers':False, 'data':0}` No columns are fixed (the default).
- **`headers`** (*a value equal to: false*; optional) | dict with keys:
- **`data`** (*number*; optional): Example `{'headers':True, 'data':1}` one column is fixed.

- **`headers`** (*a value equal to: true*; required)

`fixed_rows` (*dict*; default `{ headers: False, data: 0}`): `fixed_rows` will "fix" the set of rows so that they remain visible when scrolling vertically down the table. `fixed_rows` fixes rows from top-to-bottom, starting from the headers. If `headers` is False, no rows are fixed. If `headers` is True, all header and filter rows (see `filter_action`) are fixed. Additional data rows can be fixed by assigning a number to `data`. Note that fixing rows introduces some changes to the underlying markup of the table and may impact the way that your columns are rendered or sized. View the documentation examples to learn more.

`fixed_rows` is a dict with keys:

- **`data`** (*a value equal to: 0*; optional): Example `{'headers':False, 'data':0}` No rows are fixed (the default).
- **`headers`** (*a value equal to: false*; optional) | dict with keys:
- **`data`** (*number*; optional): Example `{'headers':True, 'data':1}` one row is fixed.
- **`headers`** (*a value equal to: true*; required)

`column_selectable` (*a value equal to: 'single', 'multi' or false*; default `False`): If `single`, then the user can select a single column or group of merged columns via the radio button that will appear in the header rows. If `multi`, then the user can select multiple columns or groups of merged columns via the checkbox that will appear in the header rows. If False, then the user will not be able to select columns and no input will appear in the header rows. When a column is selected, its id will be contained in `selected_columns` and `derived_viewport_selected_columns`.

`cell_selectable` (*boolean*; default `True`): If True (default), then it is possible to click and navigate table cells.

`row_selectable` (*a value equal to: 'single', 'multi' or false*; default `False`): If `single`, then the user can select a single row via a radio button that will appear next to each row. If `multi`, then the user can select multiple rows via a checkbox that will appear next to each row. If False, then the user will not be able to select rows and no additional UI elements will appear. When a row is selected, its index will be contained in `selected_rows`.

is selected, its index will be contained in `selected_rows`.

`row_deletable` (*boolean*; optional): If True, then a `x` will appear next to each `row` and the user can delete the row.

`active_cell` (*dict*; optional): The row and column indices and IDs of the currently active cell. `row_id` is only returned if the data rows have an `id` key.

`active_cell` is a dict with keys:

- **`column`** (*number*; optional)
- **`column_id`** (*string*; optional)
- **`row`** (*number*; optional)
- **`row_id`** (*string* | *number*; optional)

`selected_cells` (*list of dicts*; optional): `selected_cells` represents the set of cells that are selected, as an array of objects, each item similar to `active_cell`. Multiple cells can be selected by holding down shift and clicking on a different cell or holding down shift and navigating with the arrow keys.

`selected_cells` is a list of dicts with keys:

- **`column`** (*number*; optional)
- **`column_id`** (*string*; optional)
- **`row`** (*number*; optional)
- **`row_id`** (*string* | *number*; optional)

`selected_rows` (*list of numbers*; optional): `selected_rows` contains the indices of rows that are selected via the UI elements that appear when `row_selectable` is `'single'` or `'multi'`.

`selected_columns` (*list of strings*; optional): `selected_columns` contains the ids of columns that are selected via the UI elements that appear when `column_selectable` is `'single' or 'multi'`.

`selected_row_ids` (*list of strings* | *numbers*; optional): `selected_row_ids` contains the ids of rows that are selected via the UI elements that appear when `row_selectable` is `'single'` or `'multi'`.

`start_cell` (*dict*; optional): When selecting multiple cells (via clicking on a cell and then shift-clicking on another cell), `start_cell` represents the [row, column] coordinates of the cell in one of the corners of the region. `end_cell` represents the coordinates of the other corner.

`start_cell` is a dict with keys:

- **`column`** (*number*; optional)
- **`column_id`** (*string*; optional)
- **`row`** (*number*; optional)
- **`row_id`** (*string* | *number*; optional)

`end_cell` (*dict*; optional): When selecting multiple cells (via clicking on a cell and then shift-clicking on another cell), `end_cell` represents the row / column coordinates and IDs of the cell in one of the corners of the region. `start_cell` represents the coordinates of the other corner.

`end_cell` is a dict with keys:

- **`column`** (*number*; optional)
- **`column_id`** (*string*; optional)
- **`row`** (*number*; optional)
- **`row_id`** (*string* | *number*; optional)

`data_previous` (*list of dicts*; optional): The previous state of `data`. `data_previous` has the same structure as `data` and it will be updated whenever `data` changes, either through a callback or by editing the table. This is a read-only property: setting this property will not have any impact on the table.

`hidden_columns` (*list of strings*; optional): List of columns ids of the columns that are currently hidden. See the associated nested prop `columns.hideable`.

`is_focused` (*boolean*; optional): If True, then the `active_cell` is in a focused state.

focused state.

`merge_duplicate_headers` (*boolean*; optional): If True, then column headers that have neighbors with duplicate names will be merged into a single cell. This will be applied for single column headers and multi-column headers.

`data_timestamp` (*number*; optional): The unix timestamp when the data was last edited. Use this property with other timestamp properties (such as `n_clicks_timestamp` in `dash_html_components`) to determine which property has changed within a callback.

`include_headers_on_copy_paste` (*boolean*; default `False`): If True, headers are included when copying from the table to different tabs and elsewhere. Note that headers are ignored when copying from the table onto itself and between two tables within the same tab.

`export_columns` (*a value equal to: 'all' or 'visible'*; default `'visible'`): Denotes the columns that will be used in the export data file. If `all`, all columns will be used (visible + hidden). If `visible`, only the visible columns will be used. Defaults to `visible`.

`export_format` (*a value equal to: 'csv', 'xlsx' or 'none'*; default `'none'`): Denotes the type of the export data file, Defaults to `'none'`.

`export_headers` (*a value equal to: 'none', 'ids', 'names' or 'display'*; optional): Denotes the format of the headers in the export data file. If `'none'`, there will be no header. If `'display'`, then the header of the data file will be be how it is currently displayed. Note that `'display'` is only supported for `'xlsx'` export_format and will behave like `'names'` for `'csv'` export format. If `'ids'` or `'names'`, then the headers of data file will be the column id or the column names, respectively.

`page_action` (*a value equal to: 'custom', 'native' or 'none'*; default `'native'`): `page_action` refers to a mode of the table where not all of the rows are displayed at once: only a subset are displayed (a "page") and the next subset of rows can viewed by clicking "Next" or "Previous" buttons at the bottom of the page. Pagination is used to improve performance: instead of rendering all of the rows at once (which can be expensive), we only display a subset of them. With pagination, we can either page through data that exists in the table (e.g. page through `10,000` rows in `data` `100` rows at a time) or we can update the data on-the-fly with callbacks when the user clicks on the "Previous" or "Next" buttons. These modes can be toggled with this `page_action` parameter: `'native'`: all data is passed to the table up-front, paging logic is handled by the table; `'custom'`: data is passed to the table one page at a time, paging logic is handled via callbacks; `'none'`: disables paging, render all of the data at once.

`page_current` (*number*; default `0`): `page_current` represents which page the user is on. Use this property to index through data in your callbacks with backend paging.

`page_count` (*number*; optional): `page_count` represents the number of the pages in the paginated table. This is really only useful when performing backend pagination, since the front end is able to use the full size of the table to calculate the number of pages.

`page_size` (*number*; default `250`): `page_size` represents the number of rows that will be displayed on a particular page when `page_action` is `'custom'` or `'native'`.

`filter_query` (*string*; default `''`): If `filter_action` is enabled, then the current filtering string is represented in this `filter_query` property.

`filter_action` (*dict*; default `'none'`): The `filter_action` property controls the behavior of the `filtering` UI. If `'none'`, then the filtering UI is not displayed. If `'native'`, then the filtering UI is displayed and the filtering logic is handled by the table. That is, it is performed on the data that exists in the `data` property. If `'custom'`, then the filtering UI is displayed but it is the responsibility of the developer to program the filtering through a callback (where `filter_query` or `derived_filter_query_structure` would be the input and `data` would be the output).

`filter_action` is an a value equal to: 'custom', 'native' or 'none' | dict with keys:

- **`operator`** (*a value equal to: 'and' or 'or'*; optional)
- **`type`** (*a value equal to: 'custom' or 'native'*; required)

`filter_options` (*dict*; optional): There are two `filter_options` props in the table. This is the table-level filter_options prop and there is also the column-level `filter_options` prop. If the column-level `filter_options` prop is set it overrides the table-level `filter_options` prop for that column.

`filter_options` is a dict with keys:

- **`case`** (*a value equal to: 'sensitive' or 'insensitive'*; optional): (default: 'sensitive') Determine whether the applicable filter relational operators will default to `sensitive` or `insensitive` comparison.
- **`placeholder_text`** (*string*; optional): (default: 'filter data...') The filter cell placeholder text.

`sort_action` (*a value equal to: 'custom', 'native' or 'none'*; default `'none'`): The `sort_action` property enables data to be sorted on a per-column basis. If `'none'`, then the sorting UI is not displayed. If `'native'`, then the sorting UI is displayed and the sorting logic is handled by the table. That is, it is performed on the data that exists in the `data` property. If `'custom'`, the the sorting UI is displayed but it is the responsibility of the developer to program the sorting through a callback (where `sort_by` would be the input and `data` would be the output). Clicking on the sort arrows will update the `sort_by` property.

`sort_mode` (*a value equal to: 'single' or 'multi'*; default `'single'`): Sorting can be performed across multiple columns (e.g. sort by country, sort within each country, sort by year) or by a single column. NOTE - With multi-column sort, it's currently not possible to determine the order in which the columns were sorted through the UI. See https://github.com/plotly/dash-table/issues/170.

`sort_by` (*list of dicts*; optional): `sort_by` describes the current state of the sorting UI. That is, if the user clicked on the sort arrow of a column, then this property will be updated with the column ID and the direction (`asc` or `desc`) of the sort. For multi-column sorting, this will be a list of sorting parameters, in the order in which they were clicked.

`sort_by` is a list of dicts with keys:

- **`column_id`** (*string*; required)
- **`direction`** (*a value equal to: 'asc' or 'desc'*; required)

`sort_as_null` (*list of strings | numbers | booleans*; optional): An array of string, number and boolean values that are treated as `None` (i.e. ignored and always displayed last) when sorting. This value will be used by columns without `sort_as_None`. Defaults to `[]`.

`dropdown` (*dict*; optional): `dropdown` specifies dropdown options for different columns. Each entry refers to the column ID. The `clearable` property defines whether the value can be deleted. The `options` property refers to the `options` of the dropdown.

`dropdown` is a dict with strings as keys and values of type dict with keys:

- **`clearable`** (*boolean*; optional)
- **`options`** (*list of dicts*; required)

  `options` is a list of dicts with keys:

  - **`label`** (*string*; required)
  - **`value`** (*number* | *string* | *boolean*; required)

`dropdown_conditional` (*list of dicts*; optional): `dropdown_conditional` specifies dropdown options in various columns and cells. This property allows you to specify different dropdowns depending on certain conditions. For example, you may render different "city" dropdowns in a row depending on the current value in the "state" column.

`dropdown_conditional` is a list of dicts with keys:

- **`clearable`** (*boolean*; optional)
- **`if`** (*dict*; optional)

  `if` is a dict with keys:

  - **`column_id`** (*string*; optional)
  - **`filter_query`** (*string*; optional)
- **`options`** (*list of dicts*; required)

  `options` is a list of dicts with keys:

  - **`label`** (*string*; required)

- **value** (*number* | *string* | *boolean*; required)

dropdown_data (*list of dicts*; optional): `dropdown_data` specifies dropdown options on a row-by-row, column-by-column basis. Each item in the array corresponds to the corresponding dropdowns for the `data` item at the same index. Each entry in the item refers to the Column ID.

`dropdown_data` is a list of dicts with strings as keys and values of type dict with keys:

- **clearable** (*boolean*; optional)
- **options** (*list of dicts*; required)

  `options` is a list of dicts with keys:

  - **label** (*string*; required)
  - **value** (*number* | *string* | *boolean*; required)

tooltip (*dict*; optional): `tooltip` is the column based tooltip configuration applied to all rows. The key is the column id and the value is a tooltip configuration. Example: {i: {'value': i, 'use_with: 'both'} for i in df.columns}.

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

- **delay** (*number*; optional): Represents the delay in milliseconds before the tooltip is shown when hovering a cell. This overrides the table's `tooltip_delay` property. If set to `None`, the tooltip will be shown immediately.
- **duration** (*number*; optional): represents the duration in milliseconds during which the tooltip is shown when hovering a cell. This overrides the table's `tooltip_duration` property. If set to `None`, the tooltip will not disappear.
- **type** (*a value equal to: 'text' or 'markdown'*; optional): refers to the type of tooltip syntax used for the tooltip generation. Can either be `markdown` or `text`. Defaults to `text`.
- **use_with** (*a value equal to: 'both', 'data' or 'header'*; optional): Refers to whether the tooltip will be shown only on data or headers. Can be

`both`, `data`, `header`. Defaults to `both`.

- **`value`** (*string*; required): refers to the syntax-based content of the tooltip. This value is required. Alternatively, the value of the property can also be a plain string. The `text` syntax will be used in that case.

`tooltip_conditional` (*list of dicts*; optional): `tooltip_conditional` represents the tooltip shown for different columns and cells. This property allows you to specify different tooltips depending on certain conditions. For example, you may have different tooltips in the same column based on the value of a certain data property. Priority is from first to last defined conditional tooltip in the list. Higher priority (more specific) conditional tooltips should be put at the beginning of the list.

`tooltip_conditional` is a list of dicts with keys:

- **`delay`** (*number*; optional): The `delay` represents the delay in milliseconds before the tooltip is shown when hovering a cell. This overrides the table's `tooltip_delay` property. If set to `None`, the tooltip will be shown immediately.

- **`duration`** (*number*; optional): The `duration` represents the duration in milliseconds during which the tooltip is shown when hovering a cell. This overrides the table's `tooltip_duration` property. If set to `None`, the tooltip will not disappear.

- **`if`** (*dict*; required): The `if` refers to the condition that needs to be fulfilled in order for the associated tooltip configuration to be used. If multiple conditions are defined, all conditions must be met for the tooltip to be used by a cell.

  `if` is a dict with keys:

  - **`column_id`** (*string*; optional): `column_id` refers to the column ID that must be matched.

  - **`filter_query`** (*string*; optional): `filter_query` refers to the query that must evaluate to True.

  - **`row_index`** (*number* | *a value equal to: 'odd' or 'even'*; optional): `row_index` refers to the index of the row in the source `data`.

- **`type`** (*a value equal to: 'text' or 'markdown'*; optional): The `type` refers to the type of tooltip syntax used for the tooltip generation. Can either be `markdown` or `text`. Defaults to `text`.
- **`value`** (*string*; required): The `value` refers to the syntax-based content of the tooltip. This value is required.

`tooltip_data` (*list of dicts*; optional): `tooltip_data` represents the tooltip shown for different columns and cells. A list of dicts for which each key is a column id and the value is a tooltip configuration.

`tooltip_data` is a list of dicts with strings as keys and values of type string | dict with keys:

- **`delay`** (*number*; optional): The `delay` represents the delay in milliseconds before the tooltip is shown when hovering a cell. This overrides the table's `tooltip_delay` property. If set to `None`, the tooltip will be shown immediately.
- **`duration`** (*number*; optional): The `duration` represents the duration in milliseconds during which the tooltip is shown when hovering a cell. This overrides the table's `tooltip_duration` property. If set to `None`, the tooltip will not disappear. Alternatively, the value of the property can also be a plain string. The `text` syntax will be used in that case.
- **`type`** (*a value equal to: 'text' or 'markdown'*; optional): For each tooltip configuration, The `type` refers to the type of tooltip syntax used for the tooltip generation. Can either be `markdown` or `text`. Defaults to `text`.
- **`value`** (*string*; required): The `value` refers to the syntax-based content of the tooltip. This value is required.

`tooltip_header` (*dict*; optional): `tooltip_header` represents the tooltip shown for each header column and optionally each header row. Example to show long column names in a tooltip: {i: i for i in df.columns}. Example to show different column names in a tooltip: {'Rep': 'Republican', 'Dem': 'Democrat'}. If the table has multiple rows of headers, then use a list as the value of the `tooltip_header` items.

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

- **`delay`** (*number*; optional): The `delay` represents the delay in milliseconds before the tooltip is shown when hovering a cell. This overrides the table's `tooltip_delay` property. If set to `None`, the tooltip will be shown immediately.
- **`duration`** (*number*; optional): The `duration` represents the duration in milliseconds during which the tooltip is shown when hovering a cell. This overrides the table's `tooltip_duration` property. If set to `None`, the tooltip will not disappear. Alternatively, the value of the property can also be a plain string. The `text` syntax will be used in that case.
- **`type`** (*a value equal to: 'text' or 'markdown'*; optional): For each tooltip configuration, The `type` refers to the type of tooltip syntax used for the tooltip generation. Can either be `markdown` or `text`. Defaults to `text`.
- **`value`** (*string*; required): The `value` refers to the syntax-based content of the tooltip. This value is required. | list of values equal to: null | string | dict with keys:
- **`delay`** (*number*; optional)
- **`duration`** (*number*; optional)
- **`type`** (*a value equal to: 'text' or 'markdown'*; optional)
- **`value`** (*string*; required)

`tooltip_delay` (*number*; default `350`): `tooltip_delay` represents the table-wide delay in milliseconds before the tooltip is shown when hovering a cell. If set to `None`, the tooltip will be shown immediately. Defaults to 350.

`tooltip_duration` (*number*; default `2000`): `tooltip_duration` represents the table-wide duration in milliseconds during which the tooltip will be displayed when hovering a cell. If set to `None`, the tooltip will not disappear. Defaults to 2000.

`locale_format` (*dict*; optional): The localization specific formatting information applied to all columns in the table. This prop is derived from the d3.formatLocale data structure specification. When left unspecified, each individual nested prop will default to a pre-determined value.

`locale_format` is a dict with keys:

- `decimal` (*string*; optional): (default: '.'). The string used for the decimal separator.
- `group` (*string*; optional): (default: ','). The string used for the groups separator.
- `grouping` (*list of numbers*; optional): (default: [3]). A list of integers representing the grouping pattern.
- `numerals` (*list of strings*; optional): A list of ten strings used as replacements for numbers 0-9.
- `percent` (*string*; optional): (default: '%'). The string used for the percentage symbol.
- `separate_4digits` (*boolean*; optional): (default: True). Separate integers with 4-digits or less.
- `symbol` (*list of strings*; optional): (default: ['$', '']). A list of two strings representing the prefix and suffix symbols. Typically used for currency, and implemented using d3's currency format, but you can use this for other symbols such as measurement units.

`style_as_list_view` (*boolean*; default `False`): If True, then the table will be styled like a list view and not have borders between the columns.

`fill_width` (*boolean*; default `True`): `fill_width` toggles between a set of CSS for two common behaviors: True: The table container's width will grow to fill the available space; False: The table container's width will equal the width of its content.

`markdown_options` (*dict*; default `{ link_target: '_blank', html: False}`): The `markdown_options` property allows customization of the markdown cells behavior.

`markdown_options` is a dict with keys:

- **`html`** (*boolean*; optional): (default: False) If True, html may be used in markdown cells Be careful enabling html if the content being rendered

  can come from an untrusted user, as this may create an XSS vulnerability.

- **`link_target`** (*string | a value equal to: '_blank', '_parent', '_self' or '_top'*; optional): (default: '_blank'). The link's behavior (_blank opens the link in a new tab, _parent opens the link in the parent frame, _self opens the link in the current tab, and _top opens the link in the top frame) or a string.

`css` (*list of dicts*; optional): The `css` property is a way to embed CSS selectors and rules onto the page. We recommend starting with the `style_*` properties before using this `css` property. Example: [{"selector": ".dash-spreadsheet", "rule": 'font-family: "monospace"'}].

`css` is a list of dicts with keys:

- **`rule`** (*string*; required)

- **`selector`** (*string*; required)

`style_table` (*dict*; optional): CSS styles to be applied to the outer `table` container. This is commonly used for setting properties like the width or the height of the table.

`style_cell` (*dict*; optional): CSS styles to be applied to each individual cell of the table. This includes the header cells, the `data` cells, and the filter cells.

`style_data` (*dict*; optional): CSS styles to be applied to each individual data cell. That is, unlike `style_cell`, it excludes the header and filter cells.

`style_filter` (*dict*; optional): CSS styles to be applied to the filter cells. Note that this may change in the future as we build out a more complex filtering UI.

`style_header` (*dict*; optional): CSS styles to be applied to each individual header cell. That is, unlike `style_cell`, it excludes the `data` and filter cells.

`style_cell_conditional` (*list of dicts*; optional): Conditional CSS styles for the cells. This can be used to apply styles to cells on a per-column basis.

`style_cell_conditional` is a list of dicts with keys:

`style_cell_conditional` is a list of dicts with keys:

- **`if`** (*dict*; optional)

  `if` is a dict with keys:

  - **`column_id`** (*string* | *list of strings*; optional)
  - **`column_type`** (*a value equal to: 'any', 'numeric', 'text' or 'datetime'*; optional)

`style_data_conditional` (*list of dicts*; optional): Conditional CSS styles for the data cells. This can be used to apply styles to data cells on a per-column basis.

`style_data_conditional` is a list of dicts with keys:

- **`if`** (*dict*; optional)

  `if` is a dict with keys:

  - **`column_editable`** (*boolean*; optional)
  - **`column_id`** (*string* | *list of strings*; optional)
  - **`column_type`** (*a value equal to: 'any', 'numeric', 'text' or 'datetime'*; optional)
  - **`filter_query`** (*string*; optional)
  - **`row_index`** (*number* | *a value equal to: 'odd' or 'even'* | *list of numbers*; optional)
  - **`state`** (*a value equal to: 'active' or 'selected'*; optional)

`style_filter_conditional` (*list of dicts*; optional): Conditional CSS styles for the filter cells. This can be used to apply styles to filter cells on a per-column basis.

`style_filter_conditional` is a list of dicts with keys:

- **`if`** (*dict*; optional)

  `if` is a dict with keys:

  - **`column_editable`** (*boolean*; optional)

- **column_id** (*string* | *list of strings*; optional)
- **column_type** (*a value equal to: 'any', 'numeric', 'text' or 'datetime'*; optional)

style_header_conditional (*list of dicts*; optional): Conditional CSS styles for the header cells. This can be used to apply styles to header cells on a per-column basis.

`style_header_conditional` is a list of dicts with keys:

- **if** (*dict*; optional)

  `if` is a dict with keys:

  - **column_editable** (*boolean*; optional)
  - **column_id** (*string* | *list of strings*; optional)
  - **column_type** (*a value equal to: 'any', 'numeric', 'text' or 'datetime'*; optional)
  - **header_index** (*number* | *list of numbers* | *a value equal to: 'odd' or 'even'*; optional)

virtualization (*boolean*; default `False`): This property tells the table to use virtualization when rendering. Assumptions are that: the width of the columns is fixed; the height of the rows is always the same; and runtime styling changes will not affect width and height vs. first rendering.

derived_filter_query_structure (*dict*; optional): This property represents the current structure of `filter_query` as a tree structure. Each node of the query structure has: type (string; required): 'open-block', 'logical-operator', 'relational-operator', 'unary-operator', or 'expression'; subType (string; optional): 'open-block': '()', 'logical-operator': '&&', '||', 'relational-operator': '=', '>=', '>', '<=', '<', '!=', 'contains', 'unary-operator': '!', 'is bool', 'is even', 'is nil', 'is num', 'is object', 'is odd', 'is prime', 'is str', 'expression': 'value', 'field'; value (any): 'expression, value': passed value, 'expression, field': the field/prop name. block (nested query structure; optional). left (nested query structure; optional). right (nested query structure; optional). If the query is invalid or empty, the `derived_filter_query_structure` will be `None`.

derived_viewport_data (*list of dicts*; optional): This property represents the current state of `data` on the current page. This property will be updated

on paging, sorting, and filtering.

derived_viewport_indices (*list of numbers*; optional): `derived_viewport_indices` indicates the order in which the original rows appear after being filtered, sorted, and/or paged. `derived_viewport_indices` contains indices for the current page, while `derived_virtual_indices` contains indices across all pages.

derived_viewport_row_ids (*list of strings* | *numbers*; optional): `derived_viewport_row_ids` lists row IDs in the order they appear after being filtered, sorted, and/or paged. `derived_viewport_row_ids` contains IDs for the current page, while `derived_virtual_row_ids` contains IDs across all pages.

derived_viewport_selected_columns (*list of strings*; optional): `derived_viewport_selected_columns` contains the ids of the `selected_columns` that are not currently hidden.

derived_viewport_selected_rows (*list of numbers*; optional): `derived_viewport_selected_rows` represents the indices of the `selected_rows` from the perspective of the `derived_viewport_indices`.

derived_viewport_selected_row_ids (*list of strings* | *numbers*; optional): `derived_viewport_selected_row_ids` represents the IDs of the `selected_rows` on the currently visible page.

derived_virtual_data (*list of dicts*; optional): This property represents the visible state of `data` across all pages after the front-end sorting and filtering as been applied.

derived_virtual_indices (*list of numbers*; optional): `derived_virtual_indices` indicates the order in which the original rows appear after being filtered and sorted. `derived_viewport_indices` contains indices for the current page, while `derived_virtual_indices` contains indices across all pages.

derived_virtual_row_ids (*list of strings* | *numbers*; optional): `derived_virtual_row_ids` indicates the row IDs in the order in which they appear after being filtered and sorted. `derived_viewport_row_ids`

contains IDs for the current page, while `derived_virtual_row_ids` contains IDs across all pages.

`derived_virtual_selected_rows` (*list of numbers*; optional): `derived_virtual_selected_rows` represents the indices of the `selected_rows` from the perspective of the `derived_virtual_indices`.

`derived_virtual_selected_row_ids` (*list of strings* | *numbers*; optional): `derived_virtual_selected_row_ids` represents the IDs of the `selected_rows` as they appear after filtering and sorting, across all pages.

`id` (*string*; optional): The ID of the table.

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

`loading_state` is a dict with keys:

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

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

`persisted_props` (*list of values equal to: 'columns.name', 'data', 'filter_query', 'hidden_columns', 'page_current', 'selected_columns', 'selected_rows' or 'sort_by'*; default `[ 'columns.name', 'filter_query', 'hidden_columns', 'page_current', 'selected_columns', 'selected_rows', 'sort_by']`): Properties whose user interactions will persist after refreshing the component or the page.

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

Dash Python **>** *Dash DataTable* **>** ***Reference***