plotly

Star 23,447

Dash Python > ***API Reference***

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

API Reference

This page displays the docstrings for the public methods of the `dash` module including the `app` object.

Curious about the implementation details? **Browse the Dash source code**.

The dash module

```
import dash
```

dash.ALL

```
dash.ALL
```

Used in the IDs of pattern-matching callback definitions, `ALL` matches every component with the corresponding key in its ID, and invokes the callback once with all items together in a list.

dash.ALLSMALLER

```
dash.ALLSMALLER
```

Used in the IDs of `Input` and `State` items in pattern-matching callback definitions. You must use `MATCH` on the same key of an `Output`, then `ALLSMALLER` matches every component with a value smaller than that of the `MATCH`.

dash.CeleryManager

```
dash.CeleryManager(
    celery_app,
    cache_by=None,
    expire=None
)
```

Manage background execution of callbacks with a celery queue.

dash.ClientsideFunction

```
dash.ClientsideFunction(
    namespace: str,
    function_name: str
)
```

(No docstring available)

dash.Dash

```
dash.Dash(
    name: Optional[str] = None,
    server: Union[bool,
    flask.app.Flask] = True,
    assets_folder: str = 'assets',
    pages_folder: str = 'pages',
    use_pages: Optional[bool] = None,
    assets_url_path: str = 'assets',
    assets_ignore: str = '',
    assets_path_ignore: List[str] = None,
    assets_external_path: Optional[str] = None,
    eager_loading: bool = False,
    include_assets_files: bool = True,
    include_pages_meta: bool = True,
    url_base_pathname: Optional[str] = None,
    requests_pathname_prefix: Optional[str] = None,
    routes_pathname_prefix: Optional[str] = None,
    serve_locally: bool = True,
    compress: Optional[bool] = None,
    meta_tags: Optional[Sequence[Dict[str,
    Any]]] = None,
    index_string: str = '<!DOCTYPE html>\n<html>\n    <head>\n        {%me
    "Dash v2.7+ does not support Internet Explorer. Please use a newer bro
);\n        </script><![endif]-->\n        {%app_entry%}\n        <footer>
    external_scripts: Optional[Sequence[Union[str,
    Dict[str,
    Any]]]] = None,
    external_stylesheets: Optional[Sequence[Union[str,
    Dict[str,
    Any]]]] = None,
    suppress_callback_exceptions: Optional[bool] = None,
    prevent_initial_callbacks: bool = False,
    show_undo_redo: bool = False,
    extra_hot_reload_paths: Optional[Sequence[str]] = None,
    plugins: Optional[list] = None,
    title: str = 'Dash',
    update_title: str = 'Updating...',
    background_callback_manager: Optional[Any] = None,
    add_log_handler: bool = True,
    hooks: Optional[dash.types.RendererHooks] = None,
    routing_callback_inputs: Optional[Dict[str,
    Union[dash.dependencies.Input,
    dash.dependencies.State]]] = None,
    description: Optional[str] = None,
    on_error: Optional[Callable[[Exception],
    Any]] = None,
    use_async: Optional[bool] = None,
    **obsolete
)
```

Dash is a framework for building analytical web applications. No JavaScript required.

If a parameter can be set by an environment variable, that is listed as: env: `DASH_****` Values provided here take precedence over environment variables.

`name`

The name Flask should use for your app. Even if you provide your own `server`, `name` will be used to help find assets. Typically `__name__` (the magic global var, not a string) is the best value to use. Default `'__main__'`, env: `DASH_APP_NAME`

type: string

`server`

Sets the Flask server for your app. There are three options: `True` (default): Dash will create a new server `False`: The server will be added later via `app.init_app(server)` where `server` is a `flask.Flask` instance. `flask.Flask`: use this pre-existing Flask server.

type: boolean or flask.Flask

`assets_folder`

a path, relative to the current working directory, for extra files to be used in the browser. Default `'assets'`. All .js and .css files will be loaded immediately unless excluded by `assets_ignore`, and other files such as images will be served if requested.

type: string

`pages_folder`

a relative or absolute path for pages of a multi-page app. Default `'pages'`.

type: string or pathlib.Path

`use_pages`

When True, the `pages` feature for multi-page apps is enabled. If you set a non-default `pages_folder` this will be inferred to be True. Default `None`.

type: boolean

`include_pages_meta`

Include the page meta tags for twitter cards.

type: bool

`assets_url_path`

The local urls for assets will be: `requests_pathname_prefix + assets_url_path + '/' + asset_path` where `asset_path` is the path to a file inside `assets_folder`. Default `'assets'`.

type: string

`assets_ignore`

A regex, as a string to pass to `re.compile`, for assets to omit from immediate loading. Ignored files will still be served if specifically requested. You cannot use this to prevent access to sensitive files.

type: string

`assets_path_ignore`

A list of regex, each regex as a string to pass to `re.compile`, for assets path to omit from immediate loading. The files in these ignored paths will still be served if specifically requested. You cannot use this to prevent access to sensitive files.

type: list of strings

`assets_external_path`

an absolute URL from which to load assets. Use with `serve_locally=False`. assets_external_path is joined with assets_url_path to determine the absolute url to the asset folder. Dash can still find js and css to automatically load if you also keep local copies in your assets folder that Dash can index, but external serving can improve performance and reduce load on the Dash server. env: `DASH_ASSETS_EXTERNAL_PATH`

type: string

`include_assets_files`

Default `True`, set to `False` to prevent immediate loading of any assets. Assets will still be served if specifically requested. You cannot use this to prevent access to sensitive files. env: `DASH_INCLUDE_ASSETS_FILES`

type: boolean

`url_base_pathname`

A local URL prefix to use app-wide. Default `'/'`. Both `requests_pathname_prefix` and `routes_pathname_prefix` default to `url_base_pathname`. env: `DASH_URL_BASE_PATHNAME`

type: string

`requests_pathname_prefix`

A local URL prefix for file requests. Defaults to `url_base_pathname`, and must end with `routes_pathname_prefix`. env: `DASH_REQUESTS_PATHNAME_PREFIX`

type: string

`routes_pathname_prefix`

A local URL prefix for JSON requests. Defaults to `url_base_pathname`, and must start and end with `'/'`. env: `DASH_ROUTES_PATHNAME_PREFIX`

type: string

`serve_locally`

If `True` (default), assets and dependencies (Dash and Component js and css) will be served from local URLs. If `False` we will use CDN links where available.

type: boolean

`compress`

Use gzip to compress files and data served by Flask. To use this option, you need to install dash**compress**
Default `False`

type: boolean

`meta_tags`

html <meta> tags to be added to the index page. Each dict should have the attributes and values for one tag, eg: `{'name': 'description', 'content': 'My App'}`

type: list of dicts

`index_string`

Override the standard Dash index page. Must contain the correct insertion markers to interpolate various content into it depending on the app config and components used. See **https://dash.plotly.com/external-resources** for details.

type: string

`external_scripts`

Additional JS files to load with the page. Each entry can be a string (the URL) or a dict with `src` (the URL) and optionally other `<script>` tag attributes such as `integrity` and `crossorigin`.

type: list of strings or dicts

`external_stylesheets`

Additional CSS files to load with the page. Each entry can be a string (the URL) or a dict with `href` (the URL) and optionally other `<link>` tag attributes such as `rel`, `integrity` and `crossorigin`.

type: list of strings or dicts

`suppress_callback_exceptions`

Default `False`: check callbacks to ensure referenced IDs exist and props are valid. Set to `True` if your layout is dynamic, to bypass these checks. env: `DASH_SUPPRESS_CALLBACK_EXCEPTIONS`

type: boolean

`prevent_initial_callbacks`

Default `False`: Sets the default value of `prevent_initial_call` for all callbacks added to the app. Normally all callbacks are fired when the associated outputs are first added to the page. You can disable this for individual callbacks by setting `prevent_initial_call` in their definitions, or set it `True` here in which case you must explicitly set it `False` for those callbacks you wish to have an initial call. This setting has no effect on triggering callbacks when their inputs change later on.

`show_undo_redo`

Default `False`, set to `True` to enable undo and redo buttons for stepping through the history of the app state.

type: boolean

`extra_hot_reload_paths`

A list of paths to watch for changes, in addition to assets and known Python and JS code, if hot reloading is enabled.

type: list of strings

`plugins`

Extend Dash functionality by passing a list of objects with a `plug` method, taking a single argument: this app, which will be called after the Flask server is attached.

type: list of objects

`title`

Default `Dash`. Configures the document.title (the text that appears in a browser tab).

`update_title`

Default `Updating...`. Configures the document.title (the text that appears in a browser tab) text when a callback is being run. Set to None or '' if you don't want the document.title to change or if you want to control the document.title through a separate component or clientside callback.

`background_callback_manager`

Background callback manager instance to support the `@callback(..., background=True)` decorator. One of `DiskcacheManager` or `CeleryManager` currently supported.

`add_log_handler`

Automatically add a StreamHandler to the app logger if not added previously.

`hooks`

Extend Dash renderer functionality by passing a dictionary of javascript functions. To hook into the layout, use dict keys "layout_pre" and "layout_post". To hook into the callbacks, use keys "request_pre" and "request_post"

`routing_callback_inputs`

When using Dash pages (use*pages=True), allows to add new States to the routing callback, to pass additional data to the layout functions. The syntax for this parameter is a dict of State objects:* `routing_callback_inputs={"language": Input("language", "value")}` *NOTE: the keys "pathname*" and "search_" are reserved for internal use.

`description`

Sets a default description for meta tags on Dash pages (use_pages=True).

`on_error`

Global callback error handler to call when an exception is raised. Receives the exception object as first argument. The callback_context can be used to access the original callback inputs, states and output.

`use_async`

When True, the app will create async endpoints, as a dev, they will be responsible for installing the `flask[async]` dependency.

type: boolean

dash.DiskcacheManager

```
dash.DiskcacheManager(
    cache=None,
    cache_by=None,
    expire=None
)
```

Manage the background execution of callbacks with subprocesses and a diskcache result backend.

dash.Input

```
dash.Input(
    component_id: Union[str,
    dash.development.base_component.Component,
    dict],
    component_property: str,
    allow_optional: bool = False
)
```

Input of callback: trigger an update when it is updated.

dash.MATCH

```
dash.MATCH
```

Used in the IDs of pattern-matching callback definitions, `MATCH` matches every component with the corresponding key in its ID, and invokes the callback once for each item it finds.

dash.Output

```
dash.Output(
    component_id: Union[str,
    dash.development.base_component.Component,
    dict],
    component_property: str,
    allow_duplicate: bool = False
)
```

Output of a callback.

dash.Patch

```
dash.Patch(
    location: Optional[List[Union[str,
    int]]] = None,
    parent: Optional[ForwardRef(
    'Patch'
)] = None
)
```

Patch a callback output value

Act like a proxy of the output prop value on the frontend.

Supported prop types: Dictionaries and lists.

dash.State

```
dash.State(
    component_id: Union[str,
    dash.development.base_component.Component,
    dict],
    component_property: str,
    allow_optional: bool = False
)
```

Use the value of a State in a callback but don't trigger updates.

dash.background_callback

```
dash.background_callback
```

(No docstring available)

dash.callback

```
dash.callback(
    *_args,
    background: bool = False,
    interval: int = 1000,
    progress: Union[List[dash.dependencies.Output],
    dash.dependencies.Output,
    NoneType] = None,
    progress_default: Any = None,
```

```
    running: Optional[List[Tuple[dash.dependencies.Output,
    Any,
    Any]]] = None,
    cancel: Union[List[dash.dependencies.Input],
    dash.dependencies.Input,
    NoneType] = None,
    manager: Optional[dash.background_callback.managers.BaseBackgroundCall
    cache_args_to_ignore: Optional[list] = None,
    cache_ignore_triggered=True,
    on_error: Optional[Callable[[Exception],
    Any]] = None,
    **_kwargs
) -> Callable[...,
    Any]
```

Normally used as a decorator, `@dash.callback` provides a server-side callback relating the values of one or more `Output` items to one or more `Input` items which will trigger the callback when they change, and optionally `State` items which provide additional information but do not trigger the callback directly.

`@dash.callback` is an alternative to `@app.callback` (where `app = dash.Dash()`) introduced in Dash 2.0. It allows you to register callbacks without defining or importing the `app` object. The call signature is identical and it can be used instead of `app.callback` in all cases.

The last, optional argument `prevent_initial_call` causes the callback not to fire when its outputs are first added to the page. Defaults to `False` and unlike `app.callback` is not configurable at the app level.

Keyword arguments:

background

Mark the callback as a background callback to execute in a manager for callbacks that take a long time without locking up the Dash app or timing out.

manager

A background callback manager instance. Currently, an instance of one of `DiskcacheManager` or `CeleryManager`. Defaults to the `background_callback_manager` instance provided to the `dash.Dash constructor`.

- A diskcache manager (`DiskcacheManager`) that runs callback logic in a separate process and stores the results to disk using the diskcache library. This is the easiest backend to use for local development.
- A Celery manager (`CeleryManager`) that runs callback logic in a celery worker and returns results to the Dash app through a Celery broker like RabbitMQ or Redis.

running

A list of 3-element tuples. The first element of each tuple should be an `Output` dependency object referencing a property of a component in the app layout. The second element is the value that the property should be set to while the callback is running, and the third element is the value the property should be set to when the callback completes.

cancel

A list of `Input` dependency objects that reference a property of a component in the app's layout. When the value of this property changes while a callback is running, the callback is canceled. Note that the value of the property is not significant, any change in value will result in the cancellation of the running job (if any). This parameter only applies to background callbacks (`background=True`).

progress

An `Output` dependency grouping that references properties of components in the app's layout. When provided, the decorated function will be called with an extra argument as the first argument to the function. This argument, is a function handle that the decorated function should call in order to provide updates to the app on its current progress. This function accepts a single argument, which correspond to the grouping of properties specified in the provided `Output` dependency grouping. This parameter only applies to background callbacks (`background=True`).

progress_default

A grouping of values that should be assigned to the components specified by the `progress` argument when the callback is not in progress. If `progress_default` is not provided, all the dependency properties specified in `progress` will be set to `None` when the callback is not running. This parameter only applies to background callbacks (`background=True`).

cache_args_to_ignore

Arguments to ignore when caching is enabled. If callback is configured with keyword arguments (Input/State provided in a dict), this should be a list of argument names as strings. Otherwise, this should be a list of argument indices as integers. This parameter only applies to background callbacks (`background=True`).

cache_ignore_triggered

Whether to ignore which inputs triggered the callback when creating the cache. This parameter only applies to background callbacks (`background=True`).

interval

Time to wait between the background callback update requests.

on_error

Function to call when the callback raises an exception. Receives the exception object as first argument. The callback_context can be used to access the original callback inputs, states and output.

dash.callback_context

```
dash.callback_context
```

(No docstring available)

dash.clientside_callback

```
dash.clientside_callback(
    clientside_function: Union[str,
    dash.dependencies.ClientsideFunction],
    *args,
    **kwargs
)
```

(No docstring available)

dash.ctx

```
dash.ctx
```

(No docstring available)

dash.dash_table

```
dash.dash_table
```

An interactive table component designed for viewing, editing, and exploring large datasets.

dash.dcc

```
dash.dcc
```

A core set of supercharged components for interactive user interfaces.

dash.get_app

```
dash.get_app(

)
```

(No docstring available)

dash.get_asset_url

```
dash.get_asset_url(
    path
)
```

Return the URL for the provided `path` in the assets directory.

`dash.get_asset_url` is not compatible with Dash Snapshots. Use `get_asset_url` on the app instance instead: `app.get_asset_url`. See `app.get_asset_url` for more information.

dash.get_relative_path

```
dash.get_relative_path(
    path
)
```

Return a path with `requests_pathname_prefix` prefixed before it. Use this function when specifying local URL paths that will work in environments regardless of what `requests_pathname_prefix` is. In some deployment environments, like Dash Enterprise, `requests_pathname_prefix` is set to the application name, e.g. `my-dash-app`. When working locally, `requests_pathname_prefix` might be unset and so a relative URL like `/page-2` can just be `/page-2`. However, when the app is deployed to a URL like `/my-dash-app`, then `dash.get_relative_path('/page-2')` will return `/my-dash-app/page-2`. This can be used as an alternative to `get_asset_url` as well with `dash.get_relative_path('/assets/logo.png')`

Use this function with `dash.strip_relative_path` in callbacks that deal with `dcc.Location` `pathname` routing. That is, your usage may look like:

```
app.layout = html.Div([
    dcc.Location(id='url'),
    html.Div(id='content')
])
@dash.callback(Output('content', 'children'), [Input('url', 'pathname')])
def display_content(path):
    page_name = dash.strip_relative_path(path)
    if not page_name:  # None or ''
        return html.Div([
            dcc.Link(href=dash.get_relative_path('/page-1')),
            dcc.Link(href=dash.get_relative_path('/page-2')),
        ])
    elif page_name == 'page-1':
        return chapters.page_1
    if page_name == "page-2":
        return chapters.page_2
```

`dash.get_relative_path` is not compatible with Dash Snapshots. Use `get_relative_path` on the app instance instead: `app.get_relative_path`.

dash.hooks

```
dash.hooks
```

(No docstring available)

dash.html

```
dash.html
```

Vanilla HTML components for Dash

dash.jupyter_dash

```
dash.jupyter_dash
```

Interact with dash apps inside jupyter notebooks.

dash.no_update

```
dash.no_update
```

Return this from a callback to stop an output from updating. See also `dash.exceptions.PreventUpdate` which you can raise to stop all outputs from updating.

dash.page_container

```
dash.page_container
```

A Div component. Div is a wrapper for the HTML5 element. For detailed attribute info see: **https://developer.mozilla.org/en-US/docs/Web/HTML/Element/div**

Keyword arguments:

- children (a list of or a singular dash component, string or number; optional): The children of this component.
- id (string; optional): The ID of this component, used to identify dash components in callbacks. The ID needs to be unique across all of the components in an app.
- accessKey (string; optional): Keyboard shortcut to activate or add focus to the element.
- aria-* (string; optional): A wildcard aria attribute.
- className (string; optional): Often used with CSS to style elements with common properties.
- contentEditable (string; optional): Indicates whether the element's content is editable.
- data-* (string; optional): A wildcard data attribute.
- dir (string; optional): Defines the text direction. Allowed values are ltr (Left-To-Right) or rtl (Right-To-Left).
- disable_n_clicks (boolean; optional): When True, this will disable the n_clicks prop. Use this to remove event listeners that may interfere with screen readers.
- draggable (string; optional): Defines whether the element can be dragged.
- hidden (a value equal to: 'hidden', 'HIDDEN' | boolean; optional): Prevents rendering of given element, while keeping child elements, e.g. script elements, active.
- key (string; optional): A unique identifier for the component, used to improve performance by React.js while rendering components See **https://reactjs.org/docs/lists-and-keys.html** for more info.
- lang (string; optional): Defines the language used in the element.
- n_clicks (number; default 0): An integer that represents the number of times that this element has been clicked on.
- n_clicks_timestamp (number; default -1): An integer that represents the time (in ms since 1970) at which n_clicks changed. This can be used to tell which button was changed most recently.
- role (string; optional): Defines an explicit role for an element for use by assistive technologies.
- spellCheck (string; optional): Indicates whether spell checking is allowed for the element.
- tabIndex (string | number; optional): Overrides the browser's default tab order and follows the one specified instead.
- title (string; optional): Text to be displayed in a tooltip when hovering over the element.

dash.page_registry

```
dash.page_registry
```

Dictionary that remembers insertion order

dash.register_page

```
dash.register_page(
    module,
    path=None,
    path_template=None,
    name=None,
    order=None,
    title=None,
    description=None,
    image=None,
    image_url=None,
    redirect_from=None,
    layout=None,
    **kwargs
)
```

Assigns the variables to `dash.page_registry` as an `OrderedDict` (ordered by `order`).

`dash.page_registry` is used by `pages_plugin` to set up the layouts as a multi-page Dash app. This includes the URL routing callbacks (using `dcc.Location`) and the HTML templates to include title, meta description, and the meta description image.

`dash.page_registry` can also be used by Dash developers to create the page navigation links or by template authors.

- `module`: The module path where this page's `layout` is defined. Often `__name__`.
- `path`: URL Path, e.g. `/` or `/home-page`. If not supplied, will be inferred from the `path_template` or `module`, e.g. based on path_template: `/asset/<asset_id` to `/asset/none` e.g. based on module: `pages.weekly_analytics` to `/weekly-analytics`
- `relative_path`: The path with `requests_pathname_prefix` prefixed before it. Use this path when specifying local URL paths that will work in environments regardless of what `requests_pathname_prefix` is. In some deployment environments, like Dash Enterprise, `requests_pathname_prefix` is set to the application name, e.g. `my-dash-app`. When working locally, `requests_pathname_prefix` might be unset and so a relative URL like `/page-2` can just be `/page-2`. However, when the app is deployed to a URL like `/my-dash-app`, then `relative_path` will be `/my-dash-app/page-2`.
- `path_template`: Add variables to a URL by marking sections with <variable_name>. The layout function then receives the <variable_name> as a keyword argument. e.g. path_template= "/asset/<asset_id>" then if pathname in browser is "/assets/a100" then layout will receive **{"asset_id":"a100"}
- `name`: The name of the link. If not supplied, will be inferred from `module`, e.g. `pages.weekly_analytics` to `Weekly analytics`
- `order`: The order of the pages in `page_registry`. If not supplied, then the filename is used and the page with path `/` has order `0`
- `title`: (string or function) Specifies the page title displayed in the browser tab. If not supplied, the app's title is used if different from the default "Dash". Otherwise, the title is the given `name` or inferred from the module name. For example, `pages.weekly_analytics` is inferred as "Weekly Analytics".
- `description`: (string or function) The <meta type="description"></meta>. If not defined, the application description will be used if available.
- `image`: The meta description image used by social media platforms. If not supplied, then it looks for the following images in `assets/`:
    - A page specific image: `assets/<module>.<extension>` is used, e.g. `assets/weekly_analytics.png`
    - A generic app image at `assets/app.<extension>`
    - A logo at `assets/logo.<extension>` When inferring the image file, it will look for the following extensions: APNG, AVIF, GIF, JPEG, JPG, PNG, SVG, WebP.
- `image_url`: Overrides the image property and sets the `<image>` meta tag to the provided image URL.

- `redirect_from`: A list of paths that should redirect to this page. For example: `redirect_from=['/v2', '/v3']`
- `layout`: The layout function or component for this page. If not supplied, then looks for `layout` from within the supplied `module`.
- `**kwargs`: Arbitrary keyword arguments that can be stored

`page_registry` stores the original property that was passed in under `supplied_<property>` and the coerced property under `<property>`. For example, if this was called:

```
register_page(
    'pages.historical_outlook',
    name='Our historical view',
    custom_key='custom value'
)
```

Then this will appear in `page_registry`:

```
OrderedDict([
    (
        'pages.historical_outlook',
        dict(
            module='pages.historical_outlook',

            supplied_path=None,
            path='/historical-outlook',

            supplied_name='Our historical view',
            name='Our historical view',

            supplied_title=None,
            title='Our historical view'

            supplied_layout=None,
            layout=<function pages.historical_outlook.layout>,

            custom_key='custom value'
        )
    ),
])
```

dash.set_props

```
dash.set_props(
    component_id: Union[str,
    dict],
    props: dict
)
```

Set the props for a component not included in the callback outputs.

dash.strip_relative_path

```
dash.strip_relative_path(
    path
)
```

Return a path with `requests_pathname_prefix` and leading and trailing slashes stripped from it. Also, if None is passed in, None is returned. Use this function with `get_relative_path` in callbacks that deal with `dcc.Location` `pathname` routing. That is, your usage may look like:

```
app.layout = html.Div([
    dcc.Location(id='url'),
    html.Div(id='content')
```

```
])
@dash.callback(Output('content', 'children'), [Input('url', 'pathname')])
def display_content(path):
    page_name = dash.strip_relative_path(path)
    if not page_name:  # None or ''
        return html.Div([
            dcc.Link(href=dash.get_relative_path('/page-1')),
            dcc.Link(href=dash.get_relative_path('/page-2')),
        ])
    elif page_name == 'page-1':
        return chapters.page_1
    if page_name == "page-2":
        return chapters.page_2
```

Note that `chapters.page_1` will be served if the user visits `/page-1` *or* `/page-1/` since `strip_relative_path` removes the trailing slash.

Also note that `strip_relative_path` is compatible with `get_relative_path` in environments where `requests_pathname_prefix` set. In some deployment environments, like Dash Enterprise, `requests_pathname_prefix` is set to the application name, e.g. `my-dash-app`. When working locally, `requests_pathname_prefix` might be unset and so a relative URL like `/page-2` can just be `/page-2`. However, when the app is deployed to a URL like `/my-dash-app`, then `dash.get_relative_path('/page-2')` will return `/my-dash-app/page-2`

The `pathname` property of `dcc.Location` will return '`/my-dash-app/page-2`' to the callback. In this case, `dash.strip_relative_path('/my-dash-app/page-2')` will return `'page-2'`

For nested URLs, slashes are still included: `dash.strip_relative_path('/page-1/sub-page-1/')` will return `page-1/sub-page-1`

dash.types

```
dash.types
```

(No docstring available)

The app Object

```
from dash import Dash
app = Dash()
```

app.STARTUP_ROUTES

```
app.STARTUP_ROUTES
```

Built-in mutable sequence.

If no argument is given, the constructor creates a new empty list. The argument must be an iterable if specified.

app.add_startup_route

```
app.add_startup_route(
    name: str,
    view_func: Callable[...,
    Any],
    methods: Sequence[Literal['POST',
    'GET']]
) -> None
```

Add a route to the app to be initialized at the end of Dash initialization. Use this if the package requires a route to be added to the app, and you will not need to worry about at what point to add it.

`name`

The name of the route. eg "my-new-url/path".

`view_func`

The function to call when the route is requested. The function should return a JSON serializable object.

`methods`

The HTTP methods that the route should respond to. eg **"GET", "POST"** or either one.

app.async_dispatch

```
app.async_dispatch(

)
```

(No docstring available)

app.callback

```
app.callback(
    *_args,
    **_kwargs
) -> Callable[...,
    Any]
```

Normally used as a decorator, `@app.callback` provides a server-side callback relating the values of one or more `Output` items to one or more `Input` items which will trigger the callback when they change, and optionally `State` items which provide additional information but do not trigger the callback directly.

The last, optional argument `prevent_initial_call` causes the callback not to fire when its outputs are first added to the page. Defaults to `False` unless `prevent_initial_callbacks=True` at the app level.

app.clientside_callback

```
app.clientside_callback(
    clientside_function,
    *args,
    **kwargs
)
```

Create a callback that updates the output by calling a clientside (JavaScript) function instead of a Python function.

Unlike `@app.callback`, `clientside_callback` is not a decorator: it takes either a `dash.dependencies.ClientsideFunction(namespace, function_name)` argument that describes which JavaScript function to call (Dash will look for the JavaScript function at `window.dash_clientside[namespace][function_name]`), or it may take a string argument that contains the clientside function source.

For example, when using a `dash.dependencies.ClientsideFunction`:

```
app.clientside_callback(
    ClientsideFunction('my_clientside_library', 'my_function'),
    Output('my-div' 'children'),
    [Input('my-input', 'value'),
     Input('another-input', 'value')]
)
```

With this signature, Dash's front-end will call `window.dash_clientside.my_clientside_library.my_function` with the current values of the `value` properties of the components `my-input` and `another-input` whenever those values change.

Include a JavaScript file by including it your `assets/` folder. The file can be named anything but you'll need to assign the function's namespace to the `window.dash_clientside` namespace. For example, this file might look:

```
window.dash_clientside = window.dash_clientside || {};
window.dash_clientside.my_clientside_library = {
    my_function: function(input_value_1, input_value_2) {
        return (
            parseFloat(input_value_1, 10) +
            parseFloat(input_value_2, 10)
        );
    }
}
```

Alternatively, you can pass the JavaScript source directly to `clientside_callback`. In this case, the same example would look like:

```
app.clientside_callback(
    '''
    function(input_value_1, input_value_2) {
        return (
            parseFloat(input_value_1, 10) +
            parseFloat(input_value_2, 10)
        );
    }
    ''',
    Output('my-div' 'children'),
    [Input('my-input', 'value'),
     Input('another-input', 'value')]
)
```

The last, optional argument `prevent_initial_call` causes the callback not to fire when its outputs are first added to the page. Defaults to `False` unless `prevent_initial_callbacks=True` at the app level.

app.config

```
app.config
```

Most of the app-wide settings are collected into `app.config`. In general it's preferable to set these using `Dash()` constructor arguments, but many of these settings can also be altered later, for example:

`app.config.suppress_callback_exceptions=True`

app.csp_hashes

```
app.csp_hashes(
    hash_algorithm='sha256'
) -> Sequence[str]
```

Calculates CSP hashes (sha + base64) of all inline scripts, such that one of the biggest benefits of CSP (disallowing general inline scripts) can be utilized together with Dash clientside callbacks (inline scripts).

Calculate these hashes after all inline callbacks are defined, and add them to your CSP headers before starting the server, for example with the flask-talisman package from PyPI:

flask_talisman.Talisman(app.server, content_security_policy={ "default-src": "'self'", "script-src": **"'self'"** + app.csp_hashes() })

hash_algorithm

One of the recognized CSP hash algorithms ('sha256', 'sha384', 'sha512').

returns: List of CSP hash strings of all inline scripts.

app.enable_dev_tools

```
app.enable_dev_tools(
    debug: Optional[bool] = None,
    dev_tools_ui: Optional[bool] = None,
    dev_tools_props_check: Optional[bool] = None,
    dev_tools_serve_dev_bundles: Optional[bool] = None,
    dev_tools_hot_reload: Optional[bool] = None,
    dev_tools_hot_reload_interval: Optional[int] = None,
    dev_tools_hot_reload_watch_interval: Optional[int] = None,
    dev_tools_hot_reload_max_retry: Optional[int] = None,
    dev_tools_silence_routes_logging: Optional[bool] = None,
    dev_tools_disable_version_check: Optional[bool] = None,
    dev_tools_prune_errors: Optional[bool] = None
) -> bool
```

Activate the dev tools, called by `run`. If your application is served by wsgi and you want to activate the dev tools, you can call this method out of `__main__`.

All parameters can be set by environment variables as listed. Values provided here take precedence over environment variables.

Available dev_tools environment variables:

- DASH_DEBUG
- DASH_UI
- DASH_PROPS_CHECK
- DASH_SERVE_DEV_BUNDLES
- DASH_HOT_RELOAD
- DASH_HOT_RELOAD_INTERVAL
- DASH_HOT_RELOAD_WATCH_INTERVAL
- DASH_HOT_RELOAD_MAX_RETRY
- DASH_SILENCE_ROUTES_LOGGING
- DASH_DISABLE_VERSION_CHECK
- DASH_PRUNE_ERRORS

debug

Enable/disable all the dev tools unless overridden by the arguments or environment variables. Default is `True` when `enable_dev_tools` is called directly, and `False` when called via `run`. env: `DASH_DEBUG`

type: bool

dev_tools_ui

Show the dev tools UI. env: `DASH_UI`

type: bool

dev_tools_props_check

Validate the types and values of Dash component props. env: `DASH_PROPS_CHECK`

type: bool

dev_tools_serve_dev_bundles

Serve the dev bundles. Production bundles do not necessarily include all the dev tools code. env: `DASH_SERVE_DEV_BUNDLES`

type: bool

dev_tools_hot_reload

Activate hot reloading when app, assets, and component files change. env: `DASH_HOT_RELOAD`

type: bool

dev_tools_hot_reload_interval

Interval in seconds for the client to request the reload hash. Default 3. env: `DASH_HOT_RELOAD_INTERVAL`

type: float

dev_tools_hot_reload_watch_interval

Interval in seconds for the server to check asset and component folders for changes. Default 0.5. env: `DASH_HOT_RELOAD_WATCH_INTERVAL`

type: float

dev_tools_hot_reload_max_retry

Maximum number of failed reload hash requests before failing and displaying a pop up. Default 8. env: `DASH_HOT_RELOAD_MAX_RETRY`

type: int

dev_tools_silence_routes_logging

Silence the `werkzeug` logger, will remove all routes logging. Enabled with debugging by default because hot reload hash checks generate a lot of requests. env: `DASH_SILENCE_ROUTES_LOGGING`

type: bool

dev_tools_disable_version_check

Silence the upgrade notification to prevent making requests to the Dash server. env: `DASH_DISABLE_VERSION_CHECK`

type: bool

dev_tools_prune_errors

Reduce tracebacks to just user code, stripping out Flask and Dash pieces. Only available with debugging. `True` by default, set to `False` to see the complete traceback. env: `DASH_PRUNE_ERRORS`

type: bool

returns: debug

app.enable_pages

```
app.enable_pages(

) -> None
```

(No docstring available)

app.get_asset_url

```
app.get_asset_url(
    path: str
) -> str
```

Return the URL for the provided `path` in the assets directory.

If `assets_external_path` is set, `get_asset_url` returns `assets_external_path` + `assets_url_path` + `path`, where `path` is the path passed to `get_asset_url`.

Otherwise, `get_asset_url` returns `requests_pathname_prefix` + `assets_url_path` + `path`, where `path` is the path passed to `get_asset_url`.

Use `get_asset_url` in an app to access assets at the correct location in different environments. In a deployed app on Dash Enterprise, `requests_pathname_prefix` is the app name. For an app called "my-app", `app.get_asset_url("image.png")` would return:

```
/my-app/assets/image.png
```

While the same app running locally, without `requests_pathname_prefix` set, would return:

```
/assets/image.png
```

app.get_dist

```
app.get_dist(
    libraries: Sequence[str]
) -> list
```

(No docstring available)

app.get_relative_path

```
app.get_relative_path(
    path
)
```

Return a path with `requests_pathname_prefix` prefixed before it. Use this function when specifying local URL paths that will work in environments regardless of what `requests_pathname_prefix` is. In some deployment environments, like Dash Enterprise, `requests_pathname_prefix` is set to the application name, e.g. `my-dash-app`. When working locally, `requests_pathname_prefix` might be unset and so a relative URL like `/page-2` can just be `/page-2`. However, when the app is deployed to a URL like `/my-dash-app`, then `app.get_relative_path('/page-2')` will return `/my-dash-app/page-2`. This can be used as an alternative to `get_asset_url` as well with `app.get_relative_path('/assets/logo.png')`

Use this function with `app.strip_relative_path` in callbacks that deal with `dcc.Location` `pathname` routing. That is, your usage may look like:

```
app.layout = html.Div([
    dcc.Location(id='url'),
    html.Div(id='content')
])
@app.callback(Output('content', 'children'), [Input('url', 'pathname')])
def display_content(path):
    page_name = app.strip_relative_path(path)
    if not page_name:  # None or ''
        return html.Div([
            dcc.Link(href=app.get_relative_path('/page-1')),
            dcc.Link(href=app.get_relative_path('/page-2')),
        ])
    elif page_name == 'page-1':
        return chapters.page_1
    if page_name == "page-2":
        return chapters.page_2
```

app.index_string

```
app.index_string
```

Set this to override the HTML skeleton into which Dash inserts the app.

app.init_app

```
app.init_app(
    app: Optional[flask.app.Flask] = None,
    **kwargs
) -> None
```

Initialize the parts of Dash that require a flask app.

app.interpolate_index

```
app.interpolate_index(
    metas='',
    title='',
    css='',
```

```
    config='',
    scripts='',
    app_entry='',
    favicon='',
    renderer=''
)
```

Called to create the initial HTML string that is loaded on page. Override this method to provide you own custom HTML.

Example:

```
class MyDash(dash.Dash):
    def interpolate_index(self, **kwargs):
        return '''<!DOCTYPE html>
        <html>
            <head>
                <title>My App</title>
            </head>
            <body>
                <div id="custom-header">My custom header</div>
                {app_entry}
                {config}
                {scripts}
                {renderer}
                <div id="custom-footer">My custom footer</div>
            </body>
        </html>'''.format(app_entry=kwargs.get('app_entry'),
                          config=kwargs.get('config'),
                          scripts=kwargs.get('scripts'),
                          renderer=kwargs.get('renderer'))
```

metas

Collected & formatted meta tags.

title

The title of the app.

css

Collected & formatted css dependencies as <link> tags.

config

Configs needed by dash-renderer.

scripts

Collected & formatted scripts tags.

renderer

A script tag that instantiates the DashRenderer.

app_entry

Where the app will render.

favicon

A favicon <link> tag if found in assets folder.

returns: The interpolated HTML string for the index.

app.layout

```
app.layout
```

Set this to the initial layout the app should have on page load. Can be a Dash component or a function that returns a Dash component.

app.pages_folder

```
app.pages_folder
```

pages

app.routing_callback_inputs

```
app.routing_callback_inputs
```

dict() -> new empty dictionary dict(mapping) -> new dictionary initialized from a mapping object's (key, value) pairs dict(iterable) -> new dictionary initialized as if via: d = {} for k, v in iterable: d**k** = v dict(**kwargs) -> new dictionary initialized with the name=value pairs in the keyword argument list. For example: dict(one=1, two=2)

app.run

```
app.run(
    host: Optional[str] = None,
    port: Union[str,
    int,
    NoneType] = None,
    proxy: Optional[str] = None,
    debug: Optional[bool] = None,
    jupyter_mode: Optional[typing_extensions.Literal['inline',
    'external',
    'jupyterlab',
    'tab',
    '_none']] = None,
    jupyter_width: str = '100%',
    jupyter_height: int = 650,
    jupyter_server_url: Optional[str] = None,
    dev_tools_ui: Optional[bool] = None,
    dev_tools_props_check: Optional[bool] = None,
    dev_tools_serve_dev_bundles: Optional[bool] = None,
    dev_tools_hot_reload: Optional[bool] = None,
    dev_tools_hot_reload_interval: Optional[int] = None,
    dev_tools_hot_reload_watch_interval: Optional[int] = None,
    dev_tools_hot_reload_max_retry: Optional[int] = None,
    dev_tools_silence_routes_logging: Optional[bool] = None,
    dev_tools_disable_version_check: Optional[bool] = None,
    dev_tools_prune_errors: Optional[bool] = None,
    **flask_run_options
)
```

Start the flask server in local mode, you should not run this on a production server, use gunicorn/waitress instead.

If a parameter can be set by an environment variable, that is listed too. Values provided here take precedence over environment variables.

host

Host IP used to serve the application, default to "127.0.0.1" env: `HOST`

type: string

port

Port used to serve the application, default to "8050" env: `PORT`

type: int

proxy

If this application will be served to a different URL via a proxy configured outside of Python, you can list it here as a string of the form `"{input}::{output}"`, for example: `"http://0.0.0.0:8050::https://my.domain.com"` so that the startup message will display an accurate URL. env: `DASH_PROXY`

type: string

debug

Set Flask debug mode and enable dev tools. env: `DASH_DEBUG`

type: bool

`debug`

Enable/disable all the dev tools unless overridden by the arguments or environment variables. Default is `True` when `enable_dev_tools` is called directly, and `False` when called via `run`. env: `DASH_DEBUG`

type: bool

`dev_tools_ui`

Show the dev tools UI. env: `DASH_UI`

type: bool

`dev_tools_props_check`

Validate the types and values of Dash component props. env: `DASH_PROPS_CHECK`

type: bool

`dev_tools_serve_dev_bundles`

Serve the dev bundles. Production bundles do not necessarily include all the dev tools code. env: `DASH_SERVE_DEV_BUNDLES`

type: bool

`dev_tools_hot_reload`

Activate hot reloading when app, assets, and component files change. env: `DASH_HOT_RELOAD`

type: bool

`dev_tools_hot_reload_interval`

Interval in seconds for the client to request the reload hash. Default 3. env: `DASH_HOT_RELOAD_INTERVAL`

type: float

`dev_tools_hot_reload_watch_interval`

Interval in seconds for the server to check asset and component folders for changes. Default 0.5. env: `DASH_HOT_RELOAD_WATCH_INTERVAL`

type: float

`dev_tools_hot_reload_max_retry`

Maximum number of failed reload hash requests before failing and displaying a pop up. Default 8. env: `DASH_HOT_RELOAD_MAX_RETRY`

type: int

`dev_tools_silence_routes_logging`

Silence the `werkzeug` logger, will remove all routes logging. Enabled with debugging by default because hot reload hash checks generate a lot of requests. env: `DASH_SILENCE_ROUTES_LOGGING`

type: bool

`dev_tools_disable_version_check`

Silence the upgrade notification to prevent making requests to the Dash server. env: `DASH_DISABLE_VERSION_CHECK`

type: bool

`dev_tools_prune_errors`

Reduce tracebacks to just user code, stripping out Flask and Dash pieces. Only available with debugging. `True` by default, set to `False` to see the complete traceback. env: `DASH_PRUNE_ERRORS`

type: bool

`jupyter_mode`

How to display the application when running inside a jupyter notebook.

`jupyter_width`

Determine the width of the output cell when displaying inline in jupyter notebooks.

type: str

`jupyter_height`

Height of app when displayed using jupyter_mode="inline"

type: int

`jupyter_server_url`

Custom server url to display the app in jupyter notebook.

`flask_run_options`

Given to `Flask.run`

returns:

app.server

```
app.server(
    environ: dict,
    start_response: Callable
) -> Any
```

The Flask server associated with this app. Often used in conjunction with `gunicorn` when running the app in production with multiple workers:

`app.py`

```
app = Dash()

# expose the flask variable in the file
server = app.server
```

`Procfile`

```
gunicorn app:server
```

app.setup_startup_routes

```
app.setup_startup_routes(

) -> None
```

Initialize the startup routes stored in STARTUP_ROUTES.

app.strip_relative_path

```
app.strip_relative_path(
    path: str
) -> Optional[str]
```

Return a path with `requests_pathname_prefix` and leading and trailing slashes stripped from it. Also, if None is passed in, None is returned. Use this function with `get_relative_path` in callbacks that deal with `dcc.Location` `pathname` routing. That is, your usage may look like:

```
app.layout = html.Div([
    dcc.Location(id='url'),
    html.Div(id='content')
])
@app.callback(Output('content', 'children'), [Input('url', 'pathname')])
def display_content(path):
```

```
    page_name = app.strip_relative_path(path)
    if not page_name:  # None or ''
        return html.Div([
            dcc.Link(href=app.get_relative_path('/page-1')),
            dcc.Link(href=app.get_relative_path('/page-2')),
        ])
    elif page_name == 'page-1':
        return chapters.page_1
    if page_name == "page-2":
        return chapters.page_2
```

Note that `chapters.page_1` will be served if the user visits `/page-1` *or* `/page-1/` since `strip_relative_path` removes the trailing slash.

Also note that `strip_relative_path` is compatible with `get_relative_path` in environments where `requests_pathname_prefix` set. In some deployment environments, like Dash Enterprise, `requests_pathname_prefix` is set to the application name, e.g. `my-dash-app`. When working locally, `requests_pathname_prefix` might be unset and so a relative URL like `/page-2` can just be `/page-2`. However, when the app is deployed to a URL like `/my-dash-app`, then `app.get_relative_path('/page-2')` will return `/my-dash-app/page-2`

The `pathname` property of `dcc.Location` will return '`/my-dash-app/page-2`' to the callback. In this case, `app.strip_relative_path('/my-dash-app/page-2')` will return `'page-2'`

For nested URLs, slashes are still included: `app.strip_relative_path('/page-1/sub-page-1/')` will return `page-1/sub-page-1`

```

```

app.title

```
app.title
```

Configures the document.title (the text that appears in a browser tab).

Default is "Dash".

This is now configurable in the `Dash(title='...')` constructor instead of as a property of `app`. We have kept this property in the `app` object for backwards compatibility.

app.use_pages

```
app.use_pages
```

bool(x) -> bool

Returns True when the argument x is true, False otherwise. The builtins True and False are the only two instances of the class bool. The class bool is a subclass of the class int, and cannot be subclassed.

The dash.dependencies module

The classes in `dash.dependencies` are all used in callback definitions. Starting in Dash v2.0 these are all available directly from the main `dash` module.

dash.dependencies.ALL

dash.dependencies.ALLSMALLER

dash.dependencies.ClientsideFunction

dash.dependencies.Component

dash.dependencies.ComponentIdType

dash.dependencies.Input

dash.dependencies.MATCH

dash.dependencies.Output

dash.dependencies.Sequence

dash.dependencies.State

dash.dependencies.Union

dash.dependencies.stringify_id

The dash.exceptions module

Dash will raise exceptions under certain scenarios. Dash will always use a special exception class that can be caught to handle this particular scenario. These exception classes are in this module.

dash.exceptions.BackgroundCallbackError

dash.exceptions.CallbackException

dash.exceptions.DashException

dash.exceptions.DependencyException

dash.exceptions.DuplicateCallback

dash.exceptions.DuplicateIdError

dash.exceptions.HookError

dash.exceptions.IDsCantContainPeriods

dash.exceptions.ImportedInsideCallbackError

```
dash.exceptions.IncorrectTypeException

dash.exceptions.InvalidCallbackReturnValu
e

dash.exceptions.InvalidComponentIdError

dash.exceptions.InvalidConfig

dash.exceptions.InvalidIndexException

dash.exceptions.InvalidResourceError

dash.exceptions.MissingCallbackContextExce
ption

dash.exceptions.MissingLongCallbackManager
Error

dash.exceptions.NoLayoutException

dash.exceptions.NonExistentEventException

dash.exceptions.ObsoleteAttributeExceptio
n

dash.exceptions.ObsoleteKwargException

dash.exceptions.PageError

dash.exceptions.PreventUpdate

dash.exceptions.ProxyError

dash.exceptions.ResourceException

dash.exceptions.UnsupportedRelativePath

dash.exceptions.WildcardInLongCallback
```