# IPySlides 3.6.1 Documentation

## Creating slides with IPySlides

Abdul Saboor[1], Unknown Author[2]
Jan 16, 2024

[1]My University is somewhere in the middle of nowhere
[2]Their University is somewhere in the middle of nowhere

## Contents



# Main App

**Slides**(citation_mode='global', layout={'center': True, 'scroll': True, 'width': 100, 'aspect': 1.7777777777777777}, short_title='IPySlides | github-link', date='today', logo_src=' ', font_scale=1, text_font='Roboto', code_font='var(--jp-code-font-family)', code_style='default', code_lineno=True, animation={'main': 'slide_h', 'frame': 'appear'}, show_always=True, nav_gui=True, extensions=[])

Interactive Slides in IPython Notebook. Only one instance can exist.

All arguments are passed to corresponding methods in submodules, that you can tweak later as well.

To suppress unwanted print from other libraries/functions, use:

```
with slides.suppress_stdout():
    some_function_that_prints() # This will not be printed
    print('This will not be printed either')
    display('Something') # This will be printed
```

> 💡 Tip
>
> - Use `Slides.instance()` class method to keep older settings. `Slides()` apply default settings every time.
> - Run `slides.demo()` to see a demo of some features.
> - Run `slides.docs()` to see documentation.
> - Instructions in left settings panel are always on your fingertips.
> - Creating slides in a batch using `Slides.create` is much faster than adding them one by one.
> - In JupyterLab, right click on the slides and select `Create New View for Output` for optimized display.

# Jump between slides

Slides.**goto_button**(text, **kwargs)

Initialize a button to jump to given target slide when clicked. `text` is the text to be displayed on button. `kwargs` are passed to `ipywidgets.Button` function.

- Pass to write command or use `.display()` method to display button in a slide.
- Use `.set_target()` method under target slide.

> 💡 Tip
>
> - `goto_button` is converted to a link in exported slides that can be clicked to jump to slide.
> - You can use `.set_target()` on a previous slides and `.display()` on a later slide to create a link that jumps backwards.

# Adding Slides

> 📝 Note
>
> Besides functions below, you can add slides with `%%title`/`%%slide` magics as well.

Slides.**title**()

Use this context manager to write title. It is equivalent to `%%title` magic.

Slides.**slide**()

Slides.**frames**(slide_number, *objs, repeat=False)

Decorator for inserting frames on slide, define a function with two arguments acting on each obj in objs and current frame index. You can also call it as a function, e.g. `.frames(1,*objs)()` becuase it can write by defualt.

```
@slides.frames(1,a,b,c) # slides 1.1, 1.2, 1.3 with content a,b,c
def f(obj, idx):
    do_something(obj)
    if idx == 0: # Main Slide
        print('This is main slide')
    else:
        print('This is frame', idx)

slides.frames(1,a,b,c)() # Auto writes the frames with same content as ab
slides.frames(1,a,b,c, repeat = True)() # content is [a], [a,b], [a,b,c]
slides.frames(1,a,b,c, repeat = [(0,1),(1,2)])() # two frames with conten
```

### Parameters

- slide_number: (int) slide number to insert frames on.
- objs: expanded by * (list, tuple) of objects to write on frames. If repeat is False, only one frame is generated for each obj.
- repeat: (bool, list, tuple) If False, only one frame is generated for each obj. If True, one frame are generated in sequence of ojects linke `[a,b,c]` will generate 3 frames with [a], [a,b], [a,b,c] to given in function and will be written top to bottom. If list or tuple, it will be used as the sequence of frames to generate and number of frames = len(repeat). [(0,1),(1,2)] will generate 2 frames with [a,b] and [b,c] to given in function and will be written top to bottom or the way you write in your function.

No return of defined function required, if any, only should be display/show etc. CSS properties from `prop_dict` are applied to all slides from *objs.

Slides.**from_markdown**(start, file_or_str, trusted=False)

You can create slides from a markdown file or tex block as well. It creates slides `start + (0,1,2,3...)` in order. You should add more slides by higher number than the number of slides in the file/text, or it will overwrite.

- Slides separator should be --- (three dashes) in start of line.
- Frames separator should be -- (two dashes) in start of line. All markdown before first `--` will be written on all frames.
- In case of frames, you can add %++ (percent plus plus) in the content to add frames incrementally.
- You can use frames separator (--) inside `multicol` to make columns span multiple frames with %++.
- Variables defined in jupyter notebook can be passed to markdown file through ~\`var\` syntax.

### Markdown Content

```
# Talk Title
---
# Slide 1
|| Inline - Column A || Inline - Column B ||
~`some_var` that will be replaced by it's html value.
 ```python run source
 myslides = get_slides_instance() # Access slides instance under python c
 # code here will be executed and it's output will be shown in slide.
 ```
 ~`source` from above code block will be replaced by it's html value.
---
# Slide 2
--
## First Frame
 ```multicol 40 60
```

This will create two slides along with title page if start = 0. Second slide will have two frames.

Markdown content of each slide is stored as .markdown attribute to slide. You can append content to it later like this:

```
with slides.slide(2):
    slides.parse(slides[2].markdown) # Instead of write, parse take cares
    plot_something()
```

> 💡 Tip
>
> Find special syntax to be used in markdown by `Slides.xmd_syntax`.
```

> 💡 Tip
>
> Use `Slides.sync_with_file` to auto update slides as markdown content changes.

**Returns**: A tuple of handles to slides created. These handles can be used to access slides and set properties on them.

# Extended Markdown

Extended syntax for markdown is constructed to support almost full presentation from Markdown.

**Following syntax works only under currently building slide:**

- notesTThis is slide notes` to add notes to current slide
- citekey` to add citation to current slide
- citationscitations title` to add citations at end if `citation_mode = 'global'`.
- sectionkey` to add a section that will appear in the table of contents.
- tocTable of content header text` to add a table of contents. Run at last again to collect all.
- proxyplaceholder text` to add a proxy that can be updated later with `Slides.proxies[index].capture` contextmanager. Useful to keep placeholders for plots in markdwon.
- peoxy[Button Text]` to add a proxy that can be replaced by pasting image from clipboard later.
- Triple dashes `---` is used to split markdown text in slides inside `from_markdown(start, file_or_str)` function.
- Double dashes `--` is used to split markdown text in frames.

**Other syntax can be used everywhere in markdown:**

- A syntax func?Markdown?` will be converted to funcParsed HTML` in markdown. Useful to nest special syntax.
- You can escape backtick with backslash: \`→ `
- includemarkdown_file.md` to include a file in markdown format.
- Variables can be replaced with their HTML value (if possible) using ~variable` syntax which gives same result as slides.format_html(variable).
- Two side by side columns can be added inline using || Column A || Column B || sytnax.
- Block multicolumns are made using follwong syntax, column separator is tiple plus +++:

Markdown

```
```multicol widthA widthB
Column A
+++
Column B
```
```

- `multicol` syntax supports frames separator `--` within itself.
- Python code blocks can be exectude by syntax

Markdown

```
```python run source {.CSS_className}
slides = get_slides_instance()
slides.write('Hello, I was written from python code block using slides in
```
```

and source then can be emded with ~source syntax.

- A whole block of markdown can be CSS-classed using syntax

Markdown

```
::: block-yellow
    ### This is Header 3
    <hr/>
    Some **bold text**
```

gives

## This is Header 3

Some **bold text**

> **Note**
>
> You can also look at customblocks extension to make nested blocks with classes. It is added as dependency and can be used to build nested html blocks.

- You can use `Slides.extender` to extend additional syntax using Markdown extensions such as markdown extensions and PyMdown-Extensions
- You can serialize custom python objects to HTML using `Slides.serializer` function. Having a `__format__` method in your class enables to use {obj} syntax in python formatting and ~òbj ìn extended Markdown.

- Other options (that can also take extra args as func[arg1,x=2,y=A]àrg0) include:

color[blue]ṫext,` color[yellow,skyblue]ṫext,` vspaceṅumber in units of em,`alertṫext,`colorṫext,`imagepȁth/src,`rawṫext,`svgpȁth/src,`iframesṙc,`subṫext,`supṫext,`todayfmt like %b-%d-%Y,`textboxṫext,`detailsṫext,`centerṫext or ~ṿariable`

# Adding Content

> 📝 Note
> Besides functions below, you can add content to slides with %%xmd,%xmd as well.

Slides.**write**(*objs, widths=None)

Write `objs` to slides in columns. To create rows in a column, wrap objects in a list or tuple.
You can optionally specify `widths` as a list of percentages for each column.

Write any object that can be displayed in a cell with some additional features:

- Strings will be parsed as as extended markdown that can have citations/python code blocks/Javascript etc.
- Display another function in order by passing it to a lambda function like `lambda: func()`. Only body of the function will be displayed/printed. Return value will be ignored.
- Dispaly IPython widgets such as `ipywidgets` or `ipyvolume` by passing them directly.
- Display Axes/Figure form libraries such as `matplotlib`, `plotly` `altair`, `bokeh`, `ipyvolume` ect. by passing them directly.
- Display source code of functions/classes/modules or other languages by passing them directly or using `Slides.code` API.
- Use `Slides.alt(widget, func)` function to display widget on slides and alternative content in exported slides/report, function should return possible

HTML representation of widget.
- `ipywidgets.HTML` and its subclasses will be displayed as `Slides.alt(widget, html_converter_func)`. The value of exported HTML will be most recent.
- Other options include but not limited to:
  - Output of functions in `ipyslides.utils` module that are also linked to `Slides` object.
  - PIL images, SVGs etc.
  - IPython display objects such as Image, SVG, HTML, Audio, Video, YouTubeVideo, IFrame, Latex, Markdown, JSON, Javascript, etc.
  - Any object that has a `_repr_html_` method, you can create one for your own objects/third party objects by:
    - `Slides.serializer` API. Use its `.get_metadata` or `.display` method to display object as it is and export its HTML representation from metadata when used as `display(obj, metadata = {'text/html': 'html repr by user or by serializer.get_metadata(obj)'})` or `serializer.display(obj)`.
    - `IPython.core.formatters` API for third party libraries.

> **Note**
>
> - `write` is a robust command that can handle most of the cases. If nothing works, `repr(obj)` will be displayed.
> - You can avoid `repr(obj)` by `lambda: func()` e.g. `lambda: plt.show()`.
> - You can use `display(obj, metadata = {'text/html': 'html repr by user'})` for any object to display object as it is and export its HTML representation in metadata.
> - A single string passed to `write` is equivalent to `parse` command.
> - You can add mini columns inside a column by markdown syntax or `Slides.cols`, but content type is limited in that case.

Slides.**parse**(xmd, display_inline=True, rich_outputs=False)

Parse extended markdown and display immediately. If you need output html, use `display_inline = False` but that won't execute python code blocks. Precedence of content return/display is `rich_outputs = True` > `display_inline = True` > `parsed_html_string`.

### Example

```
1 ```python run var_name
```

```
#If no var_name, code will be executed without assigning it to any variab
import numpy as np
```
# Normal Markdown {.report-only}
```multicol 40 60
# First column is 40% width
If 40 60 was not given, all columns will be of equal width, this paragrap
{.info}
+++
# Second column is 60% wide
This ~&#96;var_name&#96; is code from above and will be substituted with
```

```python
```

> **Info**
>
> - Each block can have class names (speparated with space or .) after all other options such as `python .friendly` or `multicol .Sucess.info`.
>   - For example, `python .friendly` will be highlighted with friendly theme from pygments.
>   - Pygments themes, however, are not supported with `multicol`.
>   - You need to write and display CSS for a custom class.
> - The block with `::: class_type` syntax accepts extra classes in quotes, for example `::: multicol "Success" "info"`.
> - There are three special CSS classes `report-only`, `slides-only` and `export-only` that control appearance of content in different modes.

> **Alert**
>
> Nested blocks are not supported.

> **Info**
>
> - Find special syntax to be used in markdown by `Slides.xmd_syntax`.
> - Use `Slides.extender` or `ipyslides.xmd.extender` to add markdown extensions.

Slides.**cite**(key)

Add citation in presentation, key should be a unique string. Citations corresponding to keys used can be created by `Slides.set_citations` method. Citation can be written by Slides.citations decorator.

> 📝 Note
>
> You should set resources in start if using python script or voila, otherwise they will not be updated.

Slides.**clipboard_image**(filename, quality=95, overwrite=False)

Save image from clipboard to file with a given quality. On next run, it loads from saved file under `notebook-dir/ipyslides-assets/screenshots`. Useful to add screenshots from system into IPython. You can use overwite to overwrite existing file.

- Output can be directly used in `write` command.
- Converts to PIL image using `.to_pil()`.
- Convert to HTML representation using `.to_html()`.
- Convert to Numpy array using `.to_numpy()` in RGB format that you can plot later.

# Adding Speaker Notes

> 📝 Note
>
> You can use notes`notes content` in markdown.

> ⚡ Danger
>
> This is experimental feature, and may not work as expected.

Slides.notes.**display**()

Slides.notes.**insert**(content)

Add notes to current slide. Content could be any object except javascript and interactive widgets.

> 💡 Tip
>
> In markdown, you can use notes`notes content`.

# Displaying Source Code

Slides.code.**cast**(obj, language='python', name=None, **kwargs)

Create source code object from file, text or callable. `kwargs` are passed to `ipyslides.formatter.highlight`.

Slides.code.**context**(auto_display=True, **kwargs)

Execute and displays source code in the context manager. `kwargs` are passed to `ipyslides.formatter.highlight` function. Useful when source is written inside context manager itself. If `auto_display` is True (by default), then source is displayed before the output of code. Otherwise you can assign the source to a variable and display it later anywhere.

**Usage**:

```
with source.context(auto_display = False) as s: #if not used as `s`, stil
    do_something()
    write(s) # or s.display(), write(s)

#s.raw, s.value are accesible attributes.
#s.focus_lines, s.show_lines are methods that are used to show selective
```

Slides.code.**from_callable**(callable, **kwargs)

Returns source object from a given callable [class,function,module,method etc.] with `show_lines` and `focus_lines` methods. `kwargs` are passed to `ipyslides.formatter.highlight`

Slides.code.**from_file**(filename, language=None, name=None, **kwargs)

Returns source object with `show_lines` and `focus_lines` methods. `name` is alternate used name for language.
`kwargs` are passed to `ipyslides.formatter.highlight`.

It tries to auto detect lanaguage from filename extension, if `language` is not given.

Slides.code.**from_string**(text, language='python', name=None, **kwargs)

Creates source object from string. `name` is alternate used name for language. `kwargs` are passed to `ipyslides.formatter.highlight`.

# Contents

# Layout and Theme Settings

Slides.settings.**get_footer**(slide, update_widget=False)

Get footer text. `slide` is a slide object.

Slides.settings.**set**(**kwargs)

Add multiple settings at once. keys in kwargs should be name of a function after `Slides.settings.set_` and values should be dictionary or tuple of arguments for that function. See examples below.

```
Slides.settings.set(
    glassmorphic = dict(image_src='image_src'),
    css = ({},), # note trailing comma to make it tuple
    layout = dict(scroll=True),
)
```

Slides.settings.**set_animation**(main='slide_h', frame='appear')

Set animation for slides and frames.

Slides.settings.**set_citation_mode**(mode='global')

Set mode for citations form ['global', 'inline', 'footnote'].

Slides.settings.**set_code_style**(style='default', color=None, background=None, hover_color='var(--hover-bg)', lineno=True)

Set code style CSS. Use background for better view of your choice. This is overwritten by theme change.

Slides.settings.**set_css**(css_dict={})

Set CSS for all slides. This loads on slides navigation, so you can include keyframes animations as well. Individual slide's CSS set by `slides[index].set_css` will override this. This is exported only to html slides, not to report. `css_dict` is a nested dict of css selectors and properties. There are few special rules in `css_dict`:

- All nested selectors are joined with space, so '.A': {'.B': ... } becomes '.A .B {...}' in CSS.
- A '^' in start of a selector joins to parent selector without space, so '.A': {'^:hover': ...} becomes '.A:hover {...}' in CSS. You can also use '.A:hover' directly but it will restrict other nested keys to hover only.
- A '<' in start of a nested selector makes it root selector, so '.A': {'<.B': ...} becomes '.A {} .B {...}' in CSS.
- A list/tuple of values for a key in dict generates CSS fallback, so '.A': {'font-size': ('20px','2em')} becomes '.A {font-size: 20px; font-size: 2em;}' in CSS.

Read about specificity of CSS selectors here.

### Python

```python
{
    '.A': { # .A is repeated nowhere! But in CSS it is a lot
        'z-index': '2',
        '.B': {
            'font-size': ('24px','2em'), # fallbacks given as tuple
            '^:hover': {'opacity': '1'}, # Attach pseudo class to parent l
        },
        '> div': { # Direct nesting by >
            'padding': '0',
            '@media screen and (min-width: 650px)' : { # This will take a
                'padding': '2em',
            },
        },
        '.C p': {'font-size': '14px'},
    },
```

### CSS (output of `...format_css,...set_css` functions)

```css
<style>
.SlideArea .A {
    z-index : 2;
}
.SlideArea .A .B {
    font-size : 24px;
    font-size : 2em; /* This was second item in tuple in source dictionary
}
.SlideArea .A .B:hover {
```

```
    opacity : 1;
}
.SlideArea .A > div {
    padding : 0;
}
@media screen and (min-width: 650px) {
```

Slides.settings.**set_font_family**(text_font=None, code_font=None)

Set main fonts for text and code.

Slides.settings.**set_font_scale**(font_scale=1)

Set font scale to increase or decrease text size. 1 is default.

Slides.settings.**set_footer**(text='', numbering=True, date='today')

Set footer text. `text` should be string. `date` should be 'today' or string of date. To skip date, set it to None or ''

Slides.settings.**set_glassmorphic**(image_src, opacity=0.85, blur_radius=50)

Adds glassmorphic effect to the background. `image_src` can be a url or a local image path. `opacity` and `blur_radius` are optional.

Slides.settings.**set_layout**(center=True, scroll=True, width=100, aspect=1.7777777777777777)

Central aligment of slide by default. If False, left-top aligned.

Slides.settings.**set_logo**(src, width=60, top=0, right=0)

`src` should be PNG/JPEG file name or SVG string. width, top, right can be given as int or in valid CSS units, e.g. '16px'.

Slides.settings.**set_nav_gui**(visible=True)

Show/Hide navigation GUI, keyboard or touch still work. Hover on left-bottom corner to acess settings.

Slides.settings.**set_theme_colors**(colors={})

Set theme colors. Only take effect when using custom theme. colors must be a dictionary with exactly like this:

```
Slides.settings.set_theme_colors({'heading_color': 'navy', 'primary_fg': '
```

Slides.settings.**show_always**(b: bool = True)

If True (default), slides are shown after each cell execution where a slide constructor is present (other view will be closed). Otherwise only when `slides.show()` is called or `slides` is the last line in a cell.

> **Note**
>
> In JupyterLab, right click on the slides and select `Create New View for Output` and follow next step there to optimize display.

# Useful Functions for Rich Content

Slides.**clipboard_image**(filename, quality=95, overwrite=False)

Save image from clipboard to file with a given quality. On next run, it loads from saved file under `notebook-dir/ipyslides-assets/screenshots`. Useful to add screenshots from system into IPython. You can use overwite to overwrite existing file.

- Output can be directly used in `write` command.
- Converts to PIL image using `.to_pil()`.
- Convert to HTML representation using `.to_html()`.
- Convert to Numpy array using `.to_numpy()` in RGB format that you can plot later.

Slides.**alt**(widget, func)

Display `widget` for slides and output of `func(widget)` will be and displayed only in exported formats as HTML. `func` should return possible HTML representation (provided by user) of widget as string.

Python

```python
import ipywidgets as ipw
slides = get_slides_instance()
slides.alt(ipw.IntSlider(),lambda w: f'<input type="range" min="{w.min}"
```

> **Info**

- If you happen to be using alt many times for same type, you can use `Slides.serializer.register` and then pass that type of widget without alt.
- You can also use `display(obj, metadata=Slides.serializer.get_metadata(obj)or)` where obj is widget or any other object, but HTML representation will be oldest as given in metadata.

Slides.**alert**(text)

Alerts text!

Slides.**block**(*objs, widths=None)

Format a block like in LATEX beamer with `objs` in columns and immediately display it. Format rows by given an obj as list of objects.

---

- Block is automatically displayed and returns nothing.
- Available functions include: `block_<red,green,blue,yellow,cyan,magenta,gray>`.
- You can create blocks just by CSS classes in markdown as **{**.block**}**, **{**.block-red**}**, **{**.block-green**}**, etc.
- See documentation of `write` command for details of `objs` and `widths`.

Slides.**bokeh2html**(bokeh_fig, title='')

Write bokeh figure as HTML string to use in `ipyslide.utils.write`. **Parameters**

- bokeh_fig : Bokeh figure instance.
- title : Title for figure.

Slides.**bullets**(iterable, ordered=False, marker=None, className=None)

A powerful bullet list. `iterable` could be list of anything that you can pass to `write` command.
`marker` could be a unicode charcter or string, only effects unordered list.

Slides.**cite**(key)

Add citation in presentation, key should be a unique string. Citations corresponding to keys used can be created by `Slides.set_citations` method. Citation can be written by Slides.citations decorator.

> **Note**
>
> You should set resources in start if using python script or voila, otherwise they will not be updated.

Slides.**classed**(obj, className)

Add a class to a given object, whether a widget or html/IPYthon object and pass to `write` command.

Slides.**color**(text, fg='blue', bg=None)

Colors text, `fg` and `bg` should be valid CSS colors

Slides.**cols**(*objs, widths=None)

Returns HTML containing multiple columns of given widths.

Slides.**details**(str_html, summary='Click to show content')

Show/Hide Content in collapsed html.

Slides.**doc**(obj, prepend_str=None, members=None, itself=True)

Returns documentation of an `obj`. You can prepend a class/module name. members is True/List of attributes to show doc of.

Slides.**sub**(text)

Slides.**sup**(text)

Slides.**today**(fmt='%b %d, %Y', fg='inherit')

Returns today's date in given format.

Slides.**enable_zoom**(obj)

Wraps a given obj in a parent with 'zoom-child' class or add 'zoom-self' to widget, whether a widget or html/IPYthon object

Slides.**format_css**(css_dict)

`css_dict` is a nested dict of css selectors and properties. There are few special rules in `css_dict`:

- All nested selectors are joined with space, so '.A': {'.B': ... } becomes '.A .B {...}' in CSS.
- A '^' in start of a selector joins to parent selector without space, so '.A': {'^:hover': ...} becomes '.A:hover {...}' in CSS. You can also use '.A:hover' directly but it will restrict other nested keys to hover only.
- A '<' in start of a nested selector makes it root selector, so '.A': {'<.B': ...} becomes '.A {} .B {...}' in CSS.
- A list/tuple of values for a key in dict generates CSS fallback, so '.A': {'font-size': ('20px','2em')} becomes '.A {font-size: 20px; font-size: 2em;}' in CSS.

Read about specificity of CSS selectors here.

### Python

```
{
    '.A': { # .A is repeated nowhere! But in CSS it is a lot
        'z-index': '2',
        '.B': {
            'font-size': ('24px','2em'), # fallbacks given as tuple
            '^:hover': {'opacity': '1'}, # Attach pseudo class to parent l
        },
        '> div': { # Direct nesting by >
            'padding': '0',
            '@media screen and (min-width: 650px)' : { # This will take a
                'padding': '2em',
            },
        },
        '.C p': {'font-size': '14px'},
    },
```

### CSS (output of `...format_css,...set_css` functions)

```
<style>
.SlideArea .A {
    z-index : 2;
}
.SlideArea .A .B {
    font-size : 24px;
    font-size : 2em; /* This was second item in tuple in source dictionary
}
.SlideArea .A .B:hover {
```

```
    opacity : 1;
}
.SlideArea .A > div {
    padding : 0;
}
@media screen and (min-width: 650px) {
```

Slides.**highlight**(code, language='python', name=None, className=None, style='default', color=None, background=None, hover_color='var(--hover-bg)', lineno=True)

Highlight code with given language and style. style only works if className is given. If className is given and matches any of `pygments.styles.get_all_styles()`, then style will be applied immediately. color is used for text color as some themes dont provide text color.

Slides.**html**(tag, children=None, className=None, **node_attrs)

Returns html node with given children and node attributes like style, id etc. If an ttribute needs '-' in its name, replace it with '_'.
`tag` can be any valid html tag name. A `tag` that ends with `/` will be self closing e.g. `hr/` will be `<hr/>`.
`children` expects:

- If None, returns node such as 'image' -> Image and 'image/' -> Image
- str: A string to be added as node's text content.
- list/tuple of [objects]: A list of objects that will be parsed and added as child nodes. Widgets are not supported.

Example:

```
html('img',src='ir_uv.jpg') #Returns IPython.display.HTML("<img src='ir_u
```

> 💡 Tip
>
> To keep an image persistently embeded, use `ipyslides.utils.imge` function instead of just an html tag.

Slides.**iframe**(src, width='100%', height='auto', **kwargs)

Display `src` in an iframe. `kwrags` are passed to IPython.display.IFrame

Slides.**image**(data=None, width='95%', caption=None, **kwargs)

Displays PNG/JPEG files or image data etc, `kwrags` are passed to IPython.display.Image. You can provide following to `data` parameter:

- An opened PIL image. Useful for image operations and then direct writing to slides.
- A file path to image file.
- A url to image file.
- A str/bytes object containing image data.

Slides.**keep_format**(plaintext_or_html)

Bypasses from being parsed by markdown parser. Useful for some graphs, e.g. keep_format(obj.to_html()) preserves its actual form.

Slides.**notify**(content, timeout=5)

Send inside notifications for user to know whats happened on some button click. Remain invisible in screenshot.

Slides.**plt2html**(plt_fig=None, transparent=True, caption=None)

Write matplotib figure as HTML string to use in `ipyslide.utils.write`. **Parameters**

- plt_fig : Matplotlib's figure instance, auto picks as well.
- transparent: True of False for fig background.
- caption : Caption for figure.

Slides.**raw**(text, className=None)

Keep shape of text as it is (but apply dedent), preserving whitespaces as well.

Slides.**rows**(*objs)

Returns tuple of objects. Use in `write` for better readiability of writing rows in a column.

Slides.**section**(text)

Add section key to presentation that will appear in table of contents. Sections can be written as table of contents by Slides.toc decorator.

Slides.**set_citations**(citations=None, file=None)

Set citations from dictionary or file. `file` should be a JSON file with citations keys and values.

> **Note**
>
> - You should set citations in start if using python script or voila, otherwise they may not be updated.
> - Citations are replaced with new ones.

Slides.**set_dir**(path)

Context manager to set working directory to given path and return to previous working directory when done.

Slides.**sig**(callable, prepend_str=None)

Returns signature of a callable. You can prepend a class/module name.

Slides.**textbox**(text, **css_props)

Formats text in a box for writing e.g. inline refrences. `css_props` are applied to box and `-` should be `_` like `font-size` -> `font_size`. `text` is not parsed to general markdown i.e. only bold italic etc. applied, so if need markdown, parse it to html before. You can have common CSS for all textboxes using class `text-box`.

Slides.**suppress_output**(keep_stdout=False)

Suppress output of a block of code. If `keep_stdout` is True, only display data is suppressed.

Slides.**suppress_stdout**()

Suppress stdout in a block of code, especially unwanted print from functions in other modules.

Slides.**svg**(data=None, width='95%', caption=None, **kwargs)

Display svg file or svg string/bytes with additional customizations. `kwrags` are passed to IPython.display.SVG. You can provide url/string/bytes/filepath for svg.

Slides.**vspace**(em=1)

Returns html node with given height in `em`.

# Dynamic Content

Slides.**on_refresh**(func)

Decorator for inserting dynamic content on slide, define a function with no arguments. Content updates when `slide.update_display` is called or when `Slides.refresh` is called.

> 💡 Tip
>
> You can use it to dynamically fetch a value from a database or API while presenting, without having to run the cell again.

> 📝 Note
>
> - No return value is required. If any, should be like `display('some value')`, otherwise it will be ignored.
> - A slide with dynamic content enables a refresh button in bottom bar.
> - All slides with dynamic content are updated when refresh button in top bar is clicked.

Python

```python
import time
slides = get_slides_instance() # Get slides instance, this is to make doc
source.display() # Display source code of the block
@slides.on_refresh
def update_time():
    print('Local Time: {3}:{4}:{5}'.format(*time.localtime())) # Print ti
# Updates on update_display or refresh button click
```

Local Time: 14:59:33

> ⚠️ Alert
>
> Do not use this to change global state of slides, because that will affect all slides.

Slides.**on_load**(func)

Decorator for running a function when slide is loaded into view. No return value is required. Use this to e.g. notify during running presentation.

Python

```python
import datetime
slides = get_slides_instance() # Get slides instance, this is to make doc
source.display() # Display source code of the block
@slides.on_load
def push_toast():
    t = datetime.datetime.now()
    time = t.strftime('%H:%M:%S')
    slides.notify(f'Notification at {time}', timeout=5)
```

> ⚠️ Alert
>
> - Do not use this to change global state of slides, because that will affect all slides.
> - This can be used single time per slide, overwriting previous function.

Python

```python
skipper.set_target() # Set target for skip button
self.write('## Dynamic Content')
self.run_doc(self.on_refresh,'Slides')
self.run_doc(self.on_load,'Slides')
s.get_source().display()
```

# Content Styling

You can style or colorize your *content* and *text*. Provide CSS for that using `.format_css` or use some of the available styles. See these styles with `.css_styles` property as below:

```
Use any or combinations of these styles in className argument of writing functions:
--------------------------------------------------------------------------------
 className        | Formatting Style
================================================================================
 'text-[value]'    | [value] should be one of tiny, small, big, large, huge.
 'align-[value]'   | [value] should be one of center, left, right.
 'rtl'             | ------ اردو عربی
 'info'            | Blue text. Icon ℹ for note-info class.
 'tip'             | Blue Text. Icon 💡 for note-tip class.
 'warning'         | Orange Text. Icon ⚠️ for note-warning class.
 'success'         | Green text. Icon ✅ for note-success class.
 'error'           | Red Text. Icon ⚡ for note-error class.
 'note'            | 📝 Text with note icon.
 'slides-only'     | Text will not appear in exported html report.
```

'report-only' | Text will not appear on slides. Use to fill content in report.
'export-only' | Hidden on main slides, but will appear in exported slides/report

Python

```python
self.write(('You can **style**{.error} or **color[teal]`colorize`** your
        'Provide **CSS**{.info} for that using `.format_css` or use some c
        'See these **styles**{.success} with `.css_styles` property as bel
self.css_styles.display()
c.display()
```

# Highlighting Code

pygments is used for syntax highlighting. You can highlight code using `highlight` function or within markdown like this:

Python

```python
import ipyslides as isd
```

Javascript

```javascript
import React, { Component } from "react";
```

Markdown

```markdown
## Highlighting Code
[pygments](https://pygments.org/) is used for syntax highlighting.
You can **highlight**{.error} code using `highlight` function or within
```python
import ipyslides as isd
```
```javascript
import React, { Component } from "react";
```
proxy`source code of slide will be updated here later using slide_hand
```

# Loading from File/Exporting to HTML

> 📝 Note
> You can parse and view a markdown file. The output you can save by exporting notebook in other formats.

Slides.**sync_with_file**(start, path, trusted=False, interval=500)

Auto update slides when content of markdown file changes. You can stop syncing using `Slides.unsync` function. interval is in milliseconds, 500 ms default. Read `Slides.from_markdown` docs about content of file.
Slides.**from_markdown**(start, file_or_str, trusted=False)

You can create slides from a markdown file or tex block as well. It creates slides `start + (0,1,2,3...)` in order. You should add more slides by higher number than the number of slides in the file/text, or it will overwrite.

- Slides separator should be --- (three dashes) in start of line.
- Frames separator should be -- (two dashes) in start of line. All markdown before first `--` will be written on all frames.
- In case of frames, you can add `%++` (percent plus plus) in the content to add frames incrementally.
- You can use frames separator (--) inside `multicol` to make columns span multiple frames with `%++`.
- Variables defined in jupyter notebook can be passed to markdown file through ~v̀ar̀ syntax.

### Markdown Content

````
# Talk Title
---
# Slide 1
|| Inline - Column A || Inline - Column B ||
~`some_var` that will be replaced by it's html value.
 ```python run source
 myslides = get_slides_instance() # Access slides instance under python c
 # code here will be executed and it's output will be shown in slide.
 ```
 ~`source` from above code block will be replaced by it's html value.
---
# Slide 2
--
## First Frame
 ```multicol 40 60
````

This will create two slides along with title page if start = 0. Second slide will have two frames.
````

Markdown content of each slide is stored as .markdown attribute to slide. You can append content to it later like this:

```
with slides.slide(2):
    slides.parse(slides[2].markdown) # Instead of write, parse take cares
    plot_something()
```

> 💡 Tip
>
> Find special syntax to be used in markdown by `Slides.xmd_syntax`.

> 💡 Tip
>
> Use `Slides.sync_with_file` to auto update slides as markdown content changes.

**Returns**: A tuple of handles to slides created. These handles can be used to access slides and set properties on them.

Slides.**demo**()

Demo slides with a variety of content.

Slides.**docs**()

Create presentation from docs of IPySlides.

Slides.export.**slides**(path='slides.html', slide_number=True, overwrite=False)

Slides.export.**report**(path='report.html', page_size='letter', overwrite=False)

Build a beutiful html report from the slides that you can print. Widgets are supported via `Slides.alt(widget,func)`.

- Use 'overrides.css' file in same folder to override CSS styles.
- Use 'report-only' class to generate additional content that only appear in report.
- Use 'slides-only' class to generate content that only appear in slides.
- Use `Save as PDF` option in browser to make links work in output PDF.

# Contents

# Adding User defined Objects/Markdown Extensions

## I will be on exported slides/report

Python

```python
self.write('## Adding User defined Objects/Markdown Extensions')
self.write(
    lambda: display(self.html('h3','I will be on main slides',className='
    metadata = {'text/html': '<h3 class="warning">I will be on exported s
    s.get_source(), widths = [1,3]
)
self.write('If you need to serialize your own or third party objects not
self.doc(self.serializer,'Slides.serializer', members = True, itself = Fa
self.write('**You can also extend markdown syntax** using `markdown exten
self.doc(self.extender,'Slides.extender', members = True, itself = False)
```

> 📝 Note
> If you need to serialize your own or third party objects not serialized by this module, you can use `@Slides.serializer.register` to serialize them to html.

Slides.serializer.**display**(obj)

Display an object with metadata if a serializer available. Same as display(obj, metadata = serializer.get_metadata(obj)))

Slides.serializer.**get_func**(obj_type)

Get serializer function for a type. Returns None if not found.

Slides.serializer.**get_metadata**(obj_type)

Get metadata for a type to use in `display(obj, metadata)` for export purpose. This take precedence over object's own html representation. Returns None if not found.

Slides.serializer.**register**(obj_type, verbose=True)

Decorator to register html serializer for an object type.

- Decoracted function accepts one argument that will take `obj_type` and should return HTML string.
- This definition will take precedence over any other in the module.
- All regeisted serializers only exist for the lifetime of the module in a namespace.
- Only a single serializer can be registered for an object type.

### Usage

```
class MyObject:
    def __repr__(self):
        return 'My object is awesome'

slides = ipyslides.Slides()
@slides.serializer.register(MyObject)
def parse_myobject(obj):
    return f'<h1>{obj!r}</h1>'

my_object = MyObject()
slides.write(my_object) #This will write "My object is awesome" as main h
parse_myobject(my_object) #This will return "<h1>My object is awesome</h1
#This is equivalent to above for custom objects(builtin objects can't be
class MyObject:
```

> 📝 Note
>
> - Serializer function should return html string. It is not validated for correct code on registration time.
> - Serializer is useful for buitin types mostly, for custom objects, you can always define a `_repr_html_` method which works as expected.
> - Serialzers for widgets are equivalent to `Slides.alt(widget, func)` inside `write` command for export purpose. Other commands such as `Slides.format_html` will pick oldest value only.
> - Use `Slides.serializer.get_metadata(obj)` to get metadata of a registerd type and then use `display(obj, metadata = metadata)` to display as it is and export html from metadata. metadata is a dict with `{'text/html': 'html string'}`.

Slides.serializer.**unregister**(obj_type)

Unregister all serializer handlers for a type.

Slides.serializer.**unregisterall**()

Unregister all serializer handlers.

**You can also extend markdown syntax** using `markdown extensions`, (See here and others to install, then use as below):

Slides.extender.**clear**()

Clear all extensions and their configurations added by user.

Slides.extender.**config**(configs_dict)

Add configurations to the Markdown extensions. configs_dict is a dictionary like **{**'extension_name': config_dict**}**

Slides.extender.**extend**(extensions_list)

Add list of extensions to the Markdown parser.

# Keys and Shortcuts

- You can use `Slides.current` to access a slide currently in view.
- You can use `Slides.running` to access the slide currently being built, so you can set CSS, aminations etc.

| Shortcut | Button | Action |
|---|---|---|
| ␣ / ▸ | >, ⌄ | Move to next slide |
| Ctrl + ␣ / ◂ | <, ⌃ | Move to previous slide |
| [1-9] | | Move [1-9] slides right |
| Ctrl + [1-9] | | Move [1-9] slides left |
| Z | , | Toggle objects zoom mode |
| S | | Take screenshot |
| F | , | Toggle fullscreen |
| Esc | | Exit fullscreen |
| V | , | Toggle fit to viewport [voila only] |
| G | , | Toggle settings panel |
| L | , | Toggle LASER pointer |
| K | | Show keyboard shortcuts |

# Focus on what matters

- There is a zoom button on top bar which enables zooming of certain elements. This can be toggled by `Z` key.
- Most of supported elements are zoomable by default like images, matplotlib, bokeh, PIL image, altair plotly, dataframe, etc.
- You can also enable zooming for an object/widget by wrapping it inside `Slide.enable_zoom` function conveniently.
- You can also enable by manully adding `zoom-self`, `zoom-child` classes to an element. To prevent zooming under as `zoom-child` class, use `no-zoom` class.

## Focus on Me 😎

- If zoom button is enabled, you can hover here to zoom in this part!
- You can also zoom in this part by pressing `Z` key while mouse is over this part.

# SVG Icons

Icons that apprear on buttons inslides (and their rotations) available to use in your slides as well.

`chevron:` `pencil:` `bars:` `arrow:` `close:` `dots:` `expand:` `compress:` `camera:` `play:` `pause:` `stop:` `loading:` `circle:` `refresh:` `laser:` `zoom-in:` `zoom-out:` `win-maximize:` `win-restore:` `rows:` `columns:` `settings:`

Python

```python
import ipywidgets as ipw
btn = ipw.Button(description='Chevron-Down',icon='plus').add_class('MyIco
self.write(btn)
self.format_css({'.MyIcon .fa.fa-plus': self.icon('chevron',color='crimso
```

# Auto Slide Numbering in Python Scripts

Slides.**AutoSlides**()

Returns a named tuple `AutoSlides(get_next_number, title,slide,frames, from_markdown)` if run from inside a python script. Functions inside this tuple replace main functions while removing the 'slide_number' paramater. Useful to handle auto numbering of slides inside a sequntially running script. Call at top of script before adding slides.

> ⚠️ Alert
>
> Returns None in Jupyter's context and it is not useful there due to lack of sequence.

```
import ipyslides as isd
slides = isd.Slides()
auto = slides.AutoSlides() # Call at top of script

with auto.slide() as s:
    slides.write(f'This is slide {s.number}')
```

Use `auto.title`, `auto.slide` contextmanagers, `auto.frames` decorator and `auto.from_markdown` function without thinking about what should be slide number.

# Presentation Code

Python

```
def docs(self):
    "Create presentation from docs of IPySlides."
    self.close_view() # Close any previous view to speed up loading 10x f
    self.clear() # Clear previous content
    self.create(*range(22)) # Create slides faster

    from ..core import Slides
    self.settings.set_footer('IPySlides Documentation')

    auto = self.AutoSlides() # Does not work inside notebook (should not

    with auto.title(): # Title
        self.write(f'## IPySlides {self.version} Documentation\n### Creat
        self.center('''
            alert`Abdul Saboor`sup`1`, Unknown Authorsup`2`
```