PyWebIO

Release 1.8.0

Weimin Wang

Apr 10, 2023

MANUAL

PyWebIO provides a diverse set of imperative functions to obtain user input and output content on the browser, turning the browser into a "rich text terminal", and can be used to build simple web applications or browser-based GUI applications. Using PyWebIO, developers can write applications just like writing terminal scripts (interaction based on input and print function), without the need to have knowledge of HTML and JS. PyWebIO is ideal for quickly building interactive applications that don't require a complicated user interface.

CHAPTER

ONE

FEATURES

- Use synchronization instead of callback-based method to get input
- Non-declarative layout, simple and efficient
- Less intrusive: old script code can be transformed into a Web service only by modifying the input and output operation
- Support integration into existing web services, currently supports Flask, Django, Tornado, aiohttp and FastAPI(Starlette) framework
- Support for `asyncio` and coroutine
- Support data visualization with third-party libraries

CHAPTER

TWO

INSTALLATION

Stable version:

```
pip3 install -U pywebio
```

Development version:

```
pip3 install -U https://github.com/pywebio/PyWebIO/archive/dev-release.zip
```

Prerequisites: PyWebIO requires Python 3.5.2 or newer

CHAPTER

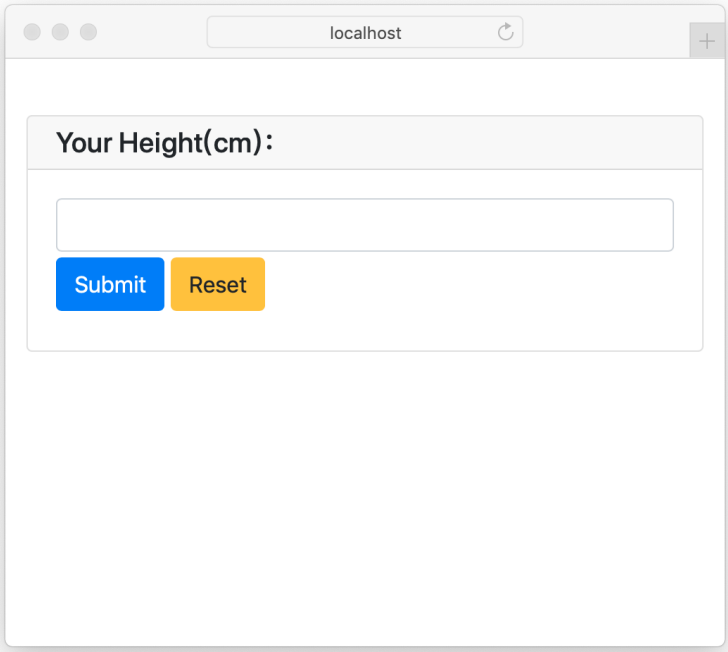
THREE

HELLO, WORLD

Here is a simple PyWebIO script to calculate the BMI

```
# A simple script to calculate BMI
from pywebio.input import input, FLOAT
from pywebio.output import put_text

def bmi():
    height = input("Input your height(cm)", type=FLOAT)
    weight = input("Input your weight(kg)", type=FLOAT)

    BMI = weight / (height / 100) ** 2

    top_status = [(16, 'Severely underweight'), (18.5, 'Underweight'),
                  (25, 'Normal'), (30, 'Overweight'),
                  (35, 'Moderately obese'), (float('inf'), 'Severely obese')]

    for top, status in top_status:
        if BMI <= top:
            put_text('Your BMI: %.1f. Category: %s' % (BMI, status))
            break

if __name__ == '__main__':
    bmi()
```

This is just a very simple script if you ignore PyWebIO, but after using the input and output functions provided by PyWebIO, you can interact with the code in the browser:



In the last line of the above code, changing the function call `bmi()` to *`pywebio.start_server(bmi, port=80)`* will start a bmi web service on port 80 (online Demo).

If you want to integrate the `bmi()` service into an existing web framework, you can visit *Integration with a web framework* section of this document.

CHAPTER

FOUR

DOCUMENTATION

This documentation is also available in PDF and Epub formats.

4.1 User's guide

If you are familiar with web development, you may not be accustomed to the usage of PyWebIO described below, which is different from the traditional web development pattern that backend implement api and frontend display content. In PyWebIO, you only need to write code in Python.

In fact, the way of writing PyWebIO applications is more like writing a console program, except that the terminal here becomes a browser. Using the imperative API provided by PyWebIO, you can simply call `put_text()`, `put_image()`, `put_table()` and other functions to output text, pictures, tables and other content to the browser, or you can call some functions such as `input()`, `select()`, `file_upload()` to display different forms on the browser to get user input. In addition, PyWebIO also provides support for click events, layout, etc. PyWebIO aims to allow you to use the least code to interact with the user and provide a good user experience as much as possible.

This user guide introduces you the most of the features of PyWebIO. There is a demo link at the top right of the example codes in this document, where you can run the example code online and see what happens. Also, the PyWebIO Playground is a good place to write, run and share your PyWebIO code online.

4.1.1 Input

The input functions are defined in the *pywebio.input* module and can be imported using `from pywebio.input import *`.

When calling the input function, an input form will be popped up on the browser. PyWebIO's input functions is blocking (same as Python's built-in `input()` function) and will not return until the form is successfully submitted.

Basic input

Here are some basic types of input.

Text input:

```
age = input("How old are you?", type=NUMBER)
```

After running the above code, the browser will pop up a text input field to get the input. After the user completes the input and submits the form, the function returns the value entered by the user.

Here are some other types of input functions:

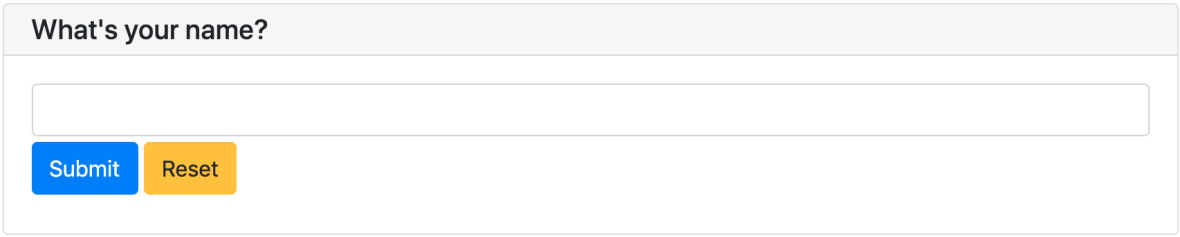
```
# Password input
password = input("Input password", type=PASSWORD)

# Drop-down selection
gift = select('Which gift you want?', ['keyboard', 'ipad'])

# Checkbox
agree = checkbox("User Term", options=['I agree to terms and conditions'])

# Single choice
answer = radio("Choose one", options=['A', 'B', 'C', 'D'])

# Multi-line text input
text = textarea('Text Area', rows=3, placeholder='Some text')

# File Upload
img = file_upload("Select a image:", accept="image/*")
```

Parameter of input functions

There are many parameters that can be passed to the input function(for complete parameters, please refer to the *function document*):

```
input('This is label', type=TEXT, placeholder='This is placeholder',
        help_text='This is help text', required=True)
```

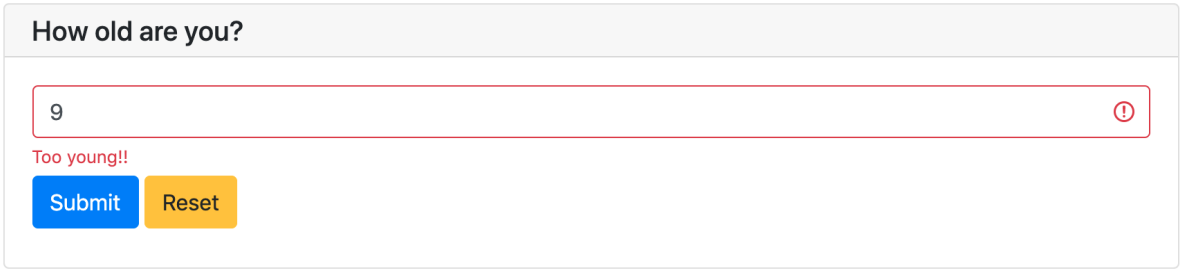
The results of the above example are as follows:



You can specify a validation function for the input by using `validate` parameter. The validation function should return `None` when the check passes, otherwise an error message will be returned:

```
def check_age(p):  # return None when the check passes, otherwise return the error␣
↪message
    if p < 10:
        return 'Too young!!'
    if p > 60:
        return 'Too old!!'

age = input("How old are you?", type=NUMBER, validate=check_age)
```

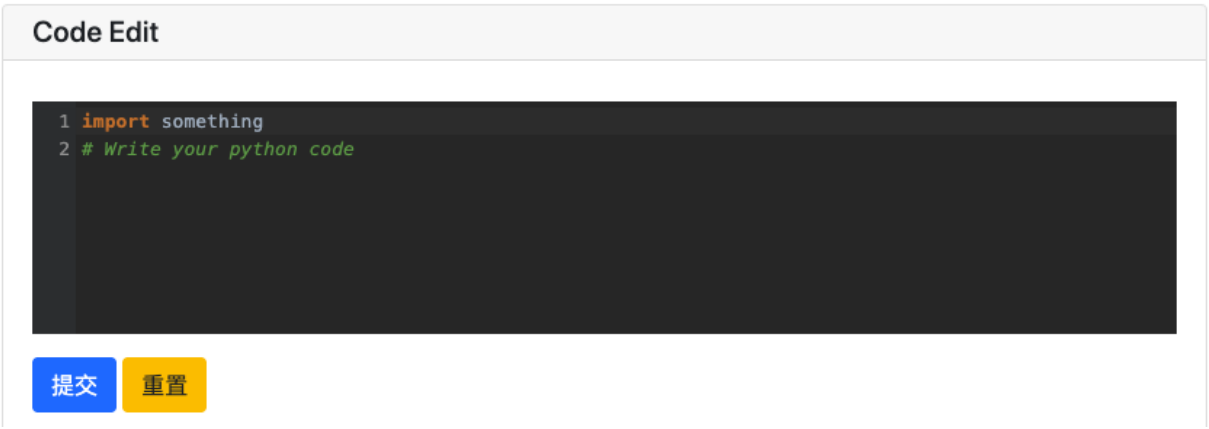
When the user input an illegal value, the input field is displayed as follows:



You can use `code` parameter in *`pywebio.input.textarea()`* to make a code editing textarea.

```
code = textarea('Code Edit', code={
    'mode': "python",
    'theme': 'darcula',
}, value='import something\n# Write your python code')
```

The results of the above example are as follows:



Input Group

PyWebIO uses input group to get multiple inputs in a single form. *`pywebio.input.input_group()`* accepts a list of single input function call as parameter, and returns a dictionary with the `name` of the single input as its key and the input data as its value:

```
data = input_group("Basic info",[
  input('Input your name', name='name'),
  input('Input your age', name='age', type=NUMBER, validate=check_age)
])
put_text(data['name'], data['age'])
```

The input group also supports using `validate` parameter to set the validation function, which accepts the entire form data as parameter:

```
def check_form(data):  # return (input name, error msg) when validation fail
    if len(data['name']) > 6:
        return ('name', 'Name too long!')
    if data['age'] <= 0:
        return ('age', 'Age can not be negative!')
```

Attention: PyWebIO determines whether the input function is in `input_group()` or is called alone according to whether the `name` parameter is passed. So when calling an input function alone, **do not** set the `name` parameter; when calling the input function in `input_group()`, you **must** provide the `name` parameter.

4.1.2 Output

The output functions are all defined in the *pywebio.output* module and can be imported using `from pywebio.output import *`.

When output functions is called, the content will be output to the browser in real time. The output functions can be called at any time during the application lifetime.

Basic Output

Using output functions, you can output a variety of content, such as text, tables, images and so on:

```
# Text Output
put_text("Hello world!")

# Table Output
put_table([
    ['Commodity', 'Price'],
    ['Apple', '5.5'],
    ['Banana', '7'],
])

# Image Output
put_image(open('/path/to/some/image.png', 'rb').read())  # local image
put_image('http://example.com/some-image.png')  # internet image

# Markdown Output
put_markdown('~~Strikethrough~~')

# File Output
put_file('hello_word.txt', b'hello word!')

# Show a PopUp
popup('popup title', 'popup text content')

# Show a notification message
toast('New message ')
```

For all output functions provided by PyWebIO, please refer to the *pywebio.output* module. In addition, PyWebIO also supports data visualization with some third-party libraries, see *Third-party library ecology*.

Note: If you use PyWebIO in interactive execution environment of Python shell, IPython or jupyter notebook, you need call `show()` method explicitly to show output:

```
>>> put_text("Hello world!").show()
>>> put_table([
...     ['A', 'B'],
...     [put_markdown(...), put_text('C')]
... ]).show()
```

Combined Output

The output functions whose name starts with `put_` can be combined with some output functions as part of the final output:

You can pass `put_xxx()` calls to *put_table()* as cell content:

```
put_table([
    ['Type', 'Content'],
    ['html', put_html('X<sup>2</sup>')],
    ['text', '<hr/>'],  # equal to ['text', put_text('<hr/>')]
    ['buttons', put_buttons(['A', 'B'], onclick=...)],
    ['markdown', put_markdown('`Awesome PyWebIO!`')],
    ['file', put_file('hello.text', b'hello world')],
    ['table', put_table([['A', 'B'], ['C', 'D']])]
])
```

The results of the above example are as follows:

Type	Content
html	X^2
text	
buttons	A B
markdown	`Awesome PyWebIO!`
file	hello.text
table	A | B C | D

Similarly, you can pass `put_xxx()` calls to *popup()* as the popup content:

```
popup('Popup title', [
    put_html('<h3>Popup Content</h3>'),
    'plain html: <br/>',  # Equivalent to: put_text('plain html: <br/>')
    put_table([['A', 'B'], ['C', 'D']]),
    put_button('close_popup()', onclick=close_popup)
])
```

In addition, you can use *put_widget()* to make your own output widgets that can accept `put_xxx()` calls.

For a full list of functions that accept `put_xxx()` calls as content, see *Output functions list*

Context Manager

Some output functions that accept `put_xxx()` calls as content can be used as context manager:

```
with put_collapse('This is title'):
    for i in range(4):
        put_text(i)

    put_table([
```

(continues on next page)

(continued from previous page)

```
        ['Commodity', 'Price'],
        ['Apple', '5.5'],
        ['Banana', '7'],
    ])
```

For a full list of functions that support context manager, see *Output functions list*

Click Callback

As we can see from the above, the interaction of PyWebIO has two parts: input and output. The input function of PyWebIO is blocking, a form will be displayed on the user's web browser when calling input function, the input function will not return until the user submits the form. The output function is used to output content to the browser in real time. The input and output behavior of PyWebIO is consistent with the console program. That's why we say PyWebIO turning the browser into a "rich text terminal". So you can write PyWebIO applications in script programming way.

In addition, PyWebIO also supports event callbacks: PyWebIO allows you to output some buttons and bind callbacks to them. The provided callback function will be executed when the button is clicked.

This is an example:

```
from functools import partial

def edit_row(choice, row):
    put_text("You click %s button ar row %s" % (choice, row))

put_table([
    ['Idx', 'Actions'],
    [1, put_buttons(['edit', 'delete'], onclick=partial(edit_row, row=1))],
    [2, put_buttons(['edit', 'delete'], onclick=partial(edit_row, row=2))],
    [3, put_buttons(['edit', 'delete'], onclick=partial(edit_row, row=3))],
])
```

The call to *put_table()* will not block. When user clicks a button, the corresponding callback function will be invoked:

Idx	Actions
1	edit delete
2	edit delete
3	edit delete

Of course, PyWebIO also supports outputting individual button:

```
def btn_click(btn_val):
    put_text("You click %s button" % btn_val)

put_buttons(['A', 'B', 'C'], onclick=btn_click)  # a group of buttons
```

(continues on next page)

(continued from previous page)

```
put_button("Click me", onclick=lambda: toast("Clicked"))  # single button
```

In fact, all output can be bound to click events, not just buttons. You can call `onclick()` method after the output function (function name like `put_xxx()`) call:

```
put_image('some-image.png').onclick(lambda: toast('You click an image'))

# set onclick in combined output
put_table([
    ['Commodity', 'Price'],
    ['Apple', put_text('5.5').onclick(lambda: toast('You click the text'))],
])
```

The return value of `onclick()` method is the object itself so it can be used in combined output.

Output Scope

PyWebIO uses the scope model to give more control to the location of content output. The output scope is a container of output content. You can create a scope in somewhere and append content to it.

Each output function (function name like `put_xxx()`) will output its content to a scope, the default is "current scope". The "current scope" is set by *use_scope()*.

use_scope()

You can use *use_scope()* to open and enter a new output scope, or enter an existing output scope:

```
with use_scope('scope1'):  # open and enter a new output: 'scope1'
    put_text('text1 in scope1')  # output text to scope1

put_text('text in parent scope of scope1')  # output text to ROOT scope

with use_scope('scope1'):  # enter an existing scope: 'scope1'
    put_text('text2 in scope1')  # output text to scope1
```

The results of the above code are as follows:

```
text1 in scope1
text2 in scope1
text in parent scope of scope1
```

You can use `clear` parameter in *use_scope()* to clear the existing content before entering the scope:

```
with use_scope('scope2'):
    put_text('create scope2')

put_text('text in parent scope of scope2')

with use_scope('scope2', clear=True):  # enter the existing scope and clear the␣
↪previous content
    put_text('text in scope2')
```

The results of the above code are as follows:

```
text in scope2
text in parent scope of scope2
```

use_scope() can also be used as decorator:

```
from datetime import datetime

@use_scope('time', clear=True)
def show_time():
    put_text(datetime.now())
```

When calling `show_time()` for the first time, a `time` scope will be created, and the current time will be output to it. And then every time the `show_time()` is called, the new content will replace the previous content.

Scopes can be nested. At the beginning, PyWebIO applications have only one `ROOT` scope. You can create new scope in a scope. For example, the following code will create 3 scopes:

```
with use_scope('A'):
    put_text('Text in scope A')

    with use_scope('B'):
        put_text('Text in scope B')

with use_scope('C'):
    put_text('Text in scope C')
```

The above code will generate the following scope layout:

put_scope()

We already know that the scope is a container of output content. So can we use this container as a sub-item of a output (like, set a cell in table as a container)? Yes, you can use *put_scope()* to create a scope explicitly. The function name starts with `put_`, which means it can be pass to the functions that accept `put_xxx()` calls.

```
put_table([
    ['Name', 'Hobbies'],
    ['Tom', put_scope('hobby', content=put_text('Coding'))]  # hobby is initialized␣
↪to coding
])

with use_scope('hobby', clear=True):
    put_text('Movie')  # hobby is reset to Movie

# append Music, Drama to hobby
with use_scope('hobby'):
    put_text('Music')
    put_text('Drama')
```

(continues on next page)

(continued from previous page)

```
# insert the Coding into the top of the hobby
put_markdown('**Coding**', scope='hobby', position=0)
```

Caution: It is not allowed to have two scopes with the same name in the application.

Scope control

In addition to *use_scope()* and *put_scope()*, PyWebIO also provides the following scope control functions:

- *clear(scope)* : Clear the contents of the scope
- *remove(scope)* : Remove scope
- *scroll_to(scope)* : Scroll the page to the scope

Also, all output functions (function name like `put_xxx()`) support a `scope` parameter to specify the destination scope to output, and support a `position` parameter to specify the insert position in target scope. Refer *output module* for more information.

Layout

In general, using the output functions introduced above is enough to output what you want, but these outputs are arranged vertically. If you want to create a more complex layout (such as displaying a code block on the left side of the page and an image on the right), you need to use layout functions.

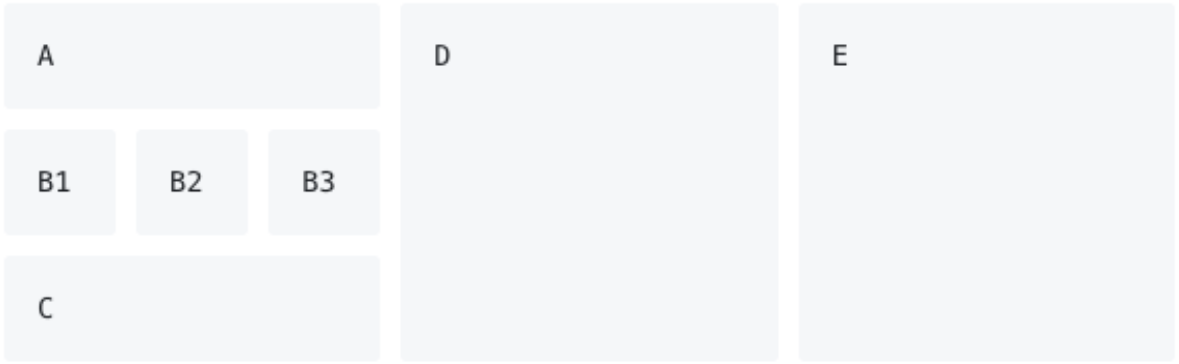
The `pywebio.output` module provides 3 layout functions, and you can create complex layouts by combining them:

- *put_row()* : Use row layout to output content. The content is arranged horizontally
- *put_column()* : Use column layout to output content. The content is arranged vertically
- *put_grid()* : Output content using grid layout

Here is an example by combining `put_row()` and `put_column()`:

```
put_row([
    put_column([
        put_code('A'),
        put_row([
            put_code('B1'), None,  # None represents the space between the output
            put_code('B2'), None,
            put_code('B3'),
        ]),
        put_code('C'),
    ]), None,
    put_code('D'), None,
    put_code('E')
])
```

The results of the above example are as follows:



The layout function also supports customizing the size of each part:

```
put_row([put_image(...), put_image(...)], size='40% 60%')  # The ratio of the width␣
↪of two images is 2:3
```

For more information, please refer to the *layout functions documentation*.

Style

If you are familiar with CSS styles, you can use the `style()` method of output return to set a custom style for the output.

You can set the CSS style for a single `put_xxx()` output:

```
put_text('hello').style('color: red; font-size: 20px')

# in combined output
put_row([
    put_text('hello').style('color: red'),
    put_markdown('markdown')
]).style('margin-top: 20px')
```

The return value of `style()` method is the object itself so it can be used in combined output.

4.1.3 Run application

In PyWebIO, there are two modes to run PyWebIO applications: running as a script and using *pywebio.start_server()* or *pywebio.platform.path_deploy()* to run as a web service.

Overview

Server mode

In server mode, PyWebIO will start a web server to continuously provide services. When the user accesses the service address, PyWebIO will open a new session and run PyWebIO application in it.

start_server() is the most common way to start a web server to serve given PyWebIO applications:

```python
from pywebio import *

def main():  # PyWebIO application function
    name = input.input("what's your name")
    output.put_text("hello", name)

start_server(main, port=8080, debug=True)
```

Now head over to http://127.0.0.1:8080/, and you should see your hello greeting.

By using `debug=True` to enable debug mode, the server will automatically reload if code changes.

The *`start_server()`* provide a remote access support, when enabled (by passing `remote_access=True` to `start_server()`), you will get a public, shareable address for the current application, others can access your application in their browser via this address. Because the processing happens on your device (as long as your device stays on!), you don't have to worry about any dependencies. Using remote access makes it easy to temporarily share the application with others.

Another way to deploy PyWebIO application as web service is using *`path_deploy()`*. *`path_deploy()`* is used to deploy the PyWebIO applications from a directory. Just define PyWebIO applications in python files under this directory, and you can access them via the path in the URL. Refer to *platform module* for more information.

Attention: Note that in Server mode, all functions from `pywebio.input`, `pywebio.output` and `pywebio.session` modules can only be called in the context of PyWebIO application functions. For example, the following code is **not allowed**:

```python
import pywebio
from pywebio.input import input

port = input('Input port number:')    #  error
pywebio.start_server(my_task_func, port=int(port))
```

Script mode

If you never call `start_server()` or `path_deploy()` in your code, then you are running PyWebIO application as script mode.

In script mode, a web browser page will be open automatically when running to the first call to PyWebIO interactive functions, and all subsequent PyWebIO interactions will take place on this page. When the script exit, the page will be inactive.

If the user closes the browser before the script exiting, then subsequent calls to PyWebIO's interactive functions will cause a `SessionException` exception.

Concurrent

PyWebIO can be used in a multi-threading environment.

Script mode

In script mode, you can freely start new thread and call PyWebIO interactive functions in it. When all non-daemonic threads finish running, the script exits.

Server mode

In server mode, if you need to use PyWebIO interactive functions in new thread, you need to use *`pywebio.session.register_thread(thread)`* to register the new thread (so that PyWebIO can know which session

the thread belongs to). If the PyWebIO interactive function is not used in the new thread, no registration is required. Threads that are not registered with *register_thread(thread)* calling PyWebIO's interactive functions will cause `SessionNotFoundException`.

Example of using multi-threading in Server mode:

```
def show_time():
    while True:
        with use_scope(name='time', clear=True):
            put_text(datetime.datetime.now())
            time.sleep(1)

def app():
    t = threading.Thread(target=show_time)
    register_thread(t)
    put_markdown('## Clock')
    t.start()  # run `show_time()` in background

    #  this thread will cause `SessionNotFoundException`
    threading.Thread(target=show_time).start()

    put_text('Background task started.')


start_server(app, port=8080, debug=True)
```

Close of session

When user close the browser page, the session will be closed. After the browser page is closed, PyWebIO input function calls that have not yet returned in the current session will cause `SessionClosedException`, and subsequent calls to PyWebIO interactive functions will cause `SessionNotFoundException` or `SessionClosedException`.

In most cases, you don't need to catch those exceptions, because let those exceptions to abort the running is the right way to exit.

You can use *pywebio.session.defer_call(func)* to set the function to be called when the session closes. *defer_call(func)* can be used for resource cleaning. You can call *defer_call(func)* multiple times in the session, and the set functions will be executed sequentially after the session closes.

4.1.4 More about PyWebIO

By now, you already get the most important features of PyWebIO and can start to write awesome PyWebIO applications. However, there are some other useful features we don't cover in the above. Here we just make a briefly explain about them. When you need them in your application, you can refer to their document.

Also, *here* is a cookbook where you can find some useful code snippets for your PyWebIO application.

`session` module

The *pywebio.session* module give you more control to session.

- Use `set_env()` to configure the title, page appearance, input panel and so on for current session.
- The `info` object provides a lot information about the current session, such as the user IP address, user language and user browser information.
- `local` is a session-local storage, it used to save data whose values are session specific.
- `run_js()` let you execute JavaScript code in user's browser, and `eval_js()` let you execute JavaScript expression and get the value of it.

`pin` module

As you already know, the input function of PyWebIO is blocking and the input form will be destroyed after successful submission. In some cases, you may want to make the input form not disappear after submission, and can continue to receive input. So PyWebIO provides the *pywebio.pin* module to achieve persistent input by pinning input widgets to the page.

`platform` module

The *pywebio.platform* module provides support for deploying PyWebIO applications in different ways.

There are two protocols (WebSocket and HTTP) can be used in server to communicates with the browser. The WebSocket is used by default. If you want to use HTTP protocol, you can choose other `start_server()` functions in this module.

You might want to set some web page related configuration (such as SEO information, js and css injection) for your PyWebIO application, `pywebio.config()` can be helpful.

Advanced features

The PyWebIO application can be integrated into an existing Python web project, the PyWebIO application and the web project share a web framework. Refer to *Advanced Topic: Integration with Web Framework* for more information.

PyWebIO also provides support for coroutine-based sessions. Refer to *Advanced Topic: Coroutine-based session* for more information.

If you try to bundles your PyWebIO application into a stand-alone executable file, to make users can run the application without installing a Python interpreter or any modules, you might want to refer to *Libraries support: Build stand-alone App*

If you want to make some data visualization in your PyWebIO application, you can't miss *Libraries support: Data visualization*

4.1.5 Last but not least

This is basically all features of PyWebIO, you can continue to read the rest of the documents, or start writing your PyWebIO applications now.

Finally, please allow me to provide one more suggestion. When you encounter a design problem when using PyWebIO, you can ask yourself a question: What would I do if it is in a terminal program? If you already have the answer, it can be done in the same way with PyWebIO. If the problem persists or the solution is not good enough, you can consider the *callback mechanism* or *pin* module.

OK, Have fun with PyWebIO!

4.2 `pywebio.input` — Get input from web browser

This module provides functions to get all kinds of input of user from the browser

There are two ways to use the input functions, one is to call the input function alone to get a single input:

```
name = input("What's your name")
print("Your name is %s" % name)
```

The other is to use *input_group* to get multiple inputs at once:

```
info = input_group("User info",[
  input('Input your name', name='name'),
  input('Input your age', name='age', type=NUMBER)
])
print(info['name'], info['age'])
```

When use *input_group*, you needs to provide the `name` parameter in each input function to identify the input items in the result.

Note: PyWebIO determines whether the input function is in *input_group* or is called alone according to whether the `name` parameter is passed. So when calling an input function alone, **do not** set the `name` parameter; when calling the input function in *input_group*, you **must** provide the `name` parameter.

By default, the user can submit empty input value. If the user must provide a non-empty input value, you need to pass `required=True` to the input function (some input functions do not support the `required` parameter)

The input functions in this module is blocking, and the input form will be destroyed after successful submission. If you want the form to always be displayed on the page and receive input continuously, you can consider the *pin* module.

4.2.1 Functions list

Function name	Description
input	Text input
textarea	Multi-line text input
select	Drop-down selection
checkbox	Checkbox
radio	Radio
slider	Slider
actions	Actions selection
file_upload	File uploading
input_group	Input group
input_update	Update input item

4.2.2 Functions doc

pywebio.input.**input**(*label: str = ''*, *type: str = 'text'*, ***, *validate: Optional[Callable[[Any], Optional[str]]] = None*, *name: Optional[str] = None*, *value: Optional[Union[str, int]] = None*, *action: Optional[Tuple[str, Callable[[Callable], None]]] = None*, *onchange: Optional[Callable[[Any], None]] = None*, *placeholder: Optional[str] = None*, *required: Optional[bool] = None*, *readonly: Optional[bool] = None*, *datalist: Optional[List[str]] = None*, *help_text: Optional[str] = None*, ***other_html_attrs*)

Text input

Parameters

- **label** (*str*) – Label of input field.
- **type** (*str*) – Input type. Currently, supported types are`TEXT` , `NUMBER` , `FLOAT` , `PASSWORD` , `URL` , `DATE` , `TIME`, `DATETIME`, `COLOR`

  The value of `DATE` , `TIME`, `DATETIME` type is a string in the format of `YYYY-MM-DD` , `HH:MM:SS` , `YYYY-MM-DDTHH:MM` respectively (`%Y-%m-%d`, `%H:%M:%S`, `%Y-%m-%dT%H:%M` in python strptime() format).
- **validate** (*callable*) – Input value validation function. If provided, the validation function will be called when user completes the input field or submits the form.

  `validate` receives the input value as a parameter. When the input value is valid, it returns `None`. When the input value is invalid, it returns an error message string.

  For example:

```
def check_age(age):
    if age>30:
        return 'Too old'
    elif age<10:
        return 'Too young'
input('Input your age', type=NUMBER, validate=check_age)
```

- **name** (*str*) – A string specifying a name for the input. Used with *input_group()* to identify different input items in the results of the input group. If call the input function alone, this parameter can **not** be set!
- **value** (*str*) – The initial value of the input

- **action** (*tuple(label:str, callback:callable)*) – Put a button on the right side of the input field, and user can click the button to set the value for the input.

  `label` is the label of the button, and `callback` is the callback function to set the input value when clicked.

  The callback is invoked with one argument, the `set_value`. `set_value` is a callable object, which is invoked with one or two arguments. You can use `set_value` to set the value for the input.

  `set_value` can be invoked with one argument: `set_value(value:str)`. The `value` parameter is the value to be set for the input.

  `set_value` can be invoked with two arguments: `set_value(value:any, label:str)`. Each arguments are described as follows:

  – `value` : The real value of the input, can be any object. it will not be passed to the user browser.

  – `label` : The text displayed to the user

  When calling `set_value` with two arguments, the input item in web page will become read-only.

  The usage scenario of `set_value(value:any, label:str)` is: You need to dynamically generate the value of the input in the callback, and hope that the result displayed to the user is different from the actual submitted data (for example, result displayed to the user can be some user-friendly texts, and the value of the input can be objects that are easier to process)

  Usage example:

```
import time
def set_now_ts(set_value):
    set_value(int(time.time()))

ts = input('Timestamp', type=NUMBER, action=('Now', set_now_ts))
from datetime import date,timedelta
def select_date(set_value):
    with popup('Select Date'):
        put_buttons(['Today'], onclick=[lambda: set_value(date.
↪today(), 'Today')])
        put_buttons(['Yesterday'], onclick=[lambda: set_value(date.
↪today() - timedelta(days=1), 'Yesterday')])

d = input('Date', action=('Select', select_date), readonly=True)
put_text(type(d), d)
```

  Note: When using *Coroutine-based session* implementation, the `callback` function can be a coroutine function.

- **onchange** (*callable*) – A callback function which will be called when user change the value of this input field.

  The `onchange` callback is invoked with one argument, the current value of input field. A typical usage scenario of `onchange` is to update other input item by using *input_update()*

- **placeholder** (*str*) – A hint to the user of what can be entered in the input. It will appear in the input field when it has no value set.

- **required** (*bool*) – Whether a value is required for the input to be submittable, default is `False`
- **readonly** (*bool*) – Whether the value is readonly(not editable)
- **datalist** (*list*) – A list of predefined values to suggest to the user for this input. Can only be used when `type=TEXT`
- **help_text** (*str*) – Help text for the input. The text will be displayed below the input field with small font
- **other_html_attrs** – Additional html attributes added to the input element. reference: https://developer.mozilla.org/zh-CN/docs/Web/HTML/Element/input#%E5%B1%9E%E6%80%A7

Returns The value that user input.

pywebio.input.**textarea**(*label: str = ''*, ***, *rows: int = 6*, *code: Optional[Union[bool, Dict]] = None*, *maxlength: Optional[int] = None*, *minlength: Optional[int] = None*, *validate: Optional[Callable[[Any], Optional[str]]] = None*, *name: Optional[str] = None*, *value: Optional[str] = None*, *onchange: Optional[Callable[[Any], None]] = None*, *placeholder: Optional[str] = None*, *required: Optional[bool] = None*, *readonly: Optional[bool] = None*, *help_text: Optional[str] = None*, ***other_html_attrs*)

Text input area (multi-line text input)

Parameters

- **rows** (*int*) – The number of visible text lines for the input area. Scroll bar will be used when content exceeds.
- **maxlength** (*int*) – The maximum number of characters (UTF-16 code units) that the user can enter. If this value isn't specified, the user can enter an unlimited number of characters.
- **minlength** (*int*) – The minimum number of characters (UTF-16 code units) required that the user should enter.
- **code** (*dict/bool*) – Enable a code style editor by providing the Codemirror options:

```
res = textarea('Text area', code={
    'mode': "python",
    'theme': 'darcula'
})
```

You can simply use `code={}` or `code=True` to enable code style editor. You can use `Esc` or `F11` to toggle fullscreen of code style textarea.

Some commonly used Codemirror options are listed here.

- **label, validate, name, value, onchange, placeholder, required, readonly, help_text, other_html_attrs** (–) – Those arguments have the same meaning as for *input()*

Returns The string value that user input.

pywebio.input.**select**(*label: str = ''*, *options: Optional[List[Union[Dict[str, Any], Tuple, List, str]]] = None*, ***, *multiple: Optional[bool] = None*, *validate: Optional[Callable[[Any], Optional[str]]] = None*, *name: Optional[str] = None*, *value: Optional[Union[List, str]] = None*, *onchange: Optional[Callable[[Any], None]] = None*, *native: bool = True*, *required: Optional[bool] = None*, *help_text: Optional[str] = None*, ***other_html_attrs*)

Drop-down selection

By default, only one option can be selected at a time, you can set `multiple` parameter to enable multiple selection.

Parameters

- **options** (*list*) – list of options. The available formats of the list items are:
  - dict:

```
{
    "label":(str) option label,
    "value":(object) option value,
    "selected":(bool, optional) whether the option is initially␣
↪selected,
    "disabled":(bool, optional) whether the option is initially␣
↪disabled
}
```

  - tuple or list: `(label, value, [selected,] [disabled])`
  - single value: label and value of option use the same value

  Attention

  1. The `value` of option can be any JSON serializable object
  2. If the `multiple` is not `True`, the list of options can only have one `selected` item at most.
- **multiple** (*bool*) – whether multiple options can be selected
- **value** (*list or str*) – The value of the initial selected item. When `multiple=True`, `value` must be a list. You can also set the initial selected option by setting the `selected` field in the `options` list item.
- **required** (*bool*) – Whether to select at least one item, only available when `multiple=True`
- **native** (*bool*) – Using browser's native select component rather than bootstrap-select. This is the default behavior.
- **label, validate, name, onchange, help_text, other_html_attrs** (–) – Those arguments have the same meaning as for *input()*

Returns If `multiple=True`, return a list of the values in the `options` selected by the user; otherwise, return the single value selected by the user.

pywebio.input.**checkbox**(*label: str = ''*, *options: Optional[List[Union[Dict[str, Any], Tuple, List, str]]] = None*, ***, *inline: Optional[bool] = None*, *validate: Optional[Callable[[Any], Optional[str]]] = None*, *name: Optional[str] = None*, *value: Optional[List] = None*, *onchange: Optional[Callable[[Any], None]] = None*, *help_text: Optional[str] = None*, ***other_html_attrs*)

A group of check box that allowing single values to be selected/deselected.

Parameters

- **options** (*list*) – List of options. The format is the same as the `options` parameter of the *select()* function
- **inline** (*bool*) – Whether to display the options on one line. Default is `False`
- **value** (*list*) – The value list of the initial selected items. You can also set the initial selected option by setting the `selected` field in the `options` list item.

- **label, validate, name, onchange, help_text, other_html_attrs** (–) – Those arguments have the same meaning as for *input()*

Returns A list of the values in the `options` selected by the user

pywebio.input.**radio**(*label: str = ''*, *options: Optional[List[Union[Dict[str, Any], Tuple, List, str]]] = None*, ***, *inline: Optional[bool] = None*, *validate: Optional[Callable[[Any], Optional[str]]] = None*, *name: Optional[str] = None*, *value: Optional[str] = None*, *onchange: Optional[Callable[[Any], None]] = None*, *required: Optional[bool] = None*, *help_text: Optional[str] = None*, ***other_html_attrs*)

A group of radio button. Only a single button can be selected.

Parameters

- **options** (*list*) – List of options. The format is the same as the `options` parameter of the *select()* function
- **inline** (*bool*) – Whether to display the options on one line. Default is `False`
- **value** (*str*) – The value of the initial selected items. You can also set the initial selected option by setting the `selected` field in the `options` list item.
- **required** (*bool*) – whether to must select one option. (the user can select nothing option by default)
- **label, validate, name, onchange, help_text, other_html_attrs** (–) – Those arguments have the same meaning as for *input()*

Returns The value of the option selected by the user, if the user does not select any value, return `None`

pywebio.input.**actions**(*label: str = ''*, *buttons: Optional[List[Union[Dict[str, Any], Tuple, List, str]]] = None*, *name: Optional[str] = None*, *help_text: Optional[str] = None*)

Actions selection

It is displayed as a group of buttons on the page. After the user clicks the button of it, it will behave differently depending on the type of the button.

Parameters

- **buttons** (*list*) – list of buttons. The available formats of the list items are:
  - dict:

```
{
   "label":(str) button label,
   "value":(object) button value,
   "type":(str, optional) button type,
   "disabled":(bool, optional) whether the button is disabled,
   "color":(str, optional) button color
}
```

    When `type='reset'/'cancel'` or `disabled=True`, `value` can be omitted
  - tuple or list: `(label, value, [type], [disabled])`
  - single value: label and value of button use the same value

  The `value` of button can be any JSON serializable object.

  `type` can be:

  - `'submit'` : After clicking the button, the entire form is submitted immediately, and the value of this input item in the final form is the `value` of the button that was clicked. `'submit'` is the default value of `type`

- `'cancel'` : Cancel form. After clicking the button, the entire form will be submitted immediately, and the form value will return `None`
- `'reset'` : Reset form. After clicking the button, the entire form will be reset, and the input items will become the initial state. Note: After clicking the `type=reset` button, the form will not be submitted, and the `actions()` call will not return

The `color` of button can be one of: `primary`, `secondary`, `success`, `danger`, `warning`, `info`, `light`, `dark`.

- **label, name, help_text** (–) – Those arguments have the same meaning as for *input()*

Returns If the user clicks the `type=submit` button to submit the form, return the value of the button clicked by the user. If the user clicks the `type=cancel` button or submits the form by other means, `None` is returned.

When `actions()` is used as the last input item in *input_group()* and contains a button with `type='submit'`, the default submit button of the *input_group()* form will be replace with the current `actions()`

**usage scenes of actions() **

- Perform simple selection operations:

```
confirm = actions('Confirm to delete file?', ['confirm', 'cancel'],
                  help_text='Unrecoverable after file deletion')
if confirm=='confirm':
    ...
```

Compared with other input items, when using *actions()*, the user only needs to click once to complete the submission.

- Replace the default submit button:

```
info = input_group('Add user', [
    input('username', type=TEXT, name='username', required=True),
    input('password', type=PASSWORD, name='password', required=True),
    actions('actions', [
        {'label': 'Save', 'value': 'save'},
        {'label': 'Save and add next', 'value': 'save_and_continue'},
        {'label': 'Reset', 'type': 'reset', 'color': 'warning'},
        {'label': 'Cancel', 'type': 'cancel', 'color': 'danger'},
    ], name='action', help_text='actions'),
])
put_code('info = ' + json.dumps(info, indent=4))
if info is not None:
    save_user(info['username'], info['password'])
    if info['action'] == 'save_and_continue':
        add_next()
```

pywebio.input.**file_upload**(*label: str = ''*, *accept: Optional[Union[List, str]] = None*, *name: Optional[str] = None*, *placeholder: str = 'Choose file'*, *multiple: bool = False*, *max_size: Union[int, str] = 0*, *max_total_size: Union[int, str] = 0*, *required: Optional[bool] = None*, *help_text: Optional[str] = None*, ***other_html_attrs*)

File uploading

Parameters

- **accept** (*str or list*) – Single value or list, indicating acceptable file types. The available formats of file types are:
  - A valid case-insensitive filename extension, starting with a period (".") character. For example: `.jpg`, `.pdf`, or `.doc`.
  - A valid MIME type string, with no extensions. For examples: `application/pdf`, `audio/*`, `video/*`, `image/*`. For more information, please visit: https://developer.mozilla.org/en-US/docs/Web/HTTP/Basics_of_HTTP/MIME_types
- **placeholder** (*str*) – A hint to the user of what to be uploaded. It will appear in the input field when there is no file selected.
- **multiple** (*bool*) – Whether to allow upload multiple files. Default is `False`.
- **max_size** (*int/str*) –

  The maximum size of a single file, exceeding the limit will prohibit uploading. The default is 0, which means there is no limit to the size.

  `max_size` can be a integer indicating the number of bytes, or a case-insensitive string ending with `K` / `M` / `G` (representing kilobytes, megabytes, and gigabytes, respectively). E.g: `max_size=500`, `max_size='40K'`, `max_size='3M'`
- **max_total_size** (*int/str*) – The maximum size of all files. Only available when `multiple=True`. The default is 0, which means there is no limit to the size. The format is the same as the `max_size` parameter
- **required** (*bool*) – Indicates whether the user must specify a file for the input. Default is `False`.
- **label, name, help_text, other_html_attrs** (–) – Those arguments have the same meaning as for *input()*

Returns

When `multiple=False`, a dict is returned:

```
{
    'filename': file name
    'content'content of the file (in bytes),
    'mime_type': MIME type of the file,
    'last_modified': Last modified time (timestamp) of the file
}
```

If there is no file uploaded, return `None`.

When `multiple=True`, a list is returned. The format of the list item is the same as the return value when `multiple=False` above. If the user does not upload a file, an empty list is returned.

Note: If uploading large files, please pay attention to the file upload size limit setting of the web framework. When using *start_server()* or *path_deploy()* to start the PyWebIO application, the maximum file size to be uploaded allowed by the web framework can be set through the `max_payload_size` parameter.

```
# Upload a file and save to server
f = input.file_upload("Upload a file")
open('asset/'+f['filename'], 'wb').write(f['content'])
```

(continues on next page)

(continued from previous page)

```
imgs = file_upload("Select some pictures:", accept="image/*", multiple=True)
for img in imgs:
    put_image(img['content'])
```

pywebio.input.**slider**(*label: str = ''*, ***, *name: Optional[str] = None*, *value: Union[int, float] = 0*, *min_value: Union[int, float] = 0*, *max_value: Union[int, float] = 100*, *step: int = 1*, *validate: Optional[Callable[[Any], Optional[str]]] = None*, *onchange: Optional[Callable[[Any], None]] = None*, *required: Optional[bool] = None*, *help_text: Optional[str] = None*, ***other_html_attrs*)

Range input.

Parameters

- **value** (*int/float*) – The initial value of the slider.
- **min_value** (*int/float*) – The minimum permitted value.
- **max_value** (*int/float*) – The maximum permitted value.
- **step** (*int*) – The stepping interval. Only available when `value`, `min_value` and `max_value` are all integer.
- **label, name, validate, onchange, required, help_text, other_html_attrs** (–) – Those arguments have the same meaning as for *input()*

Return int/float If one of `value`, `min_value` and `max_value` is float, the return value is a float, otherwise an int is returned.

pywebio.input.**input_group**(*label: str = ''*, *inputs: Optional[List] = None*, *validate: Optional[Callable[[Dict], Optional[Tuple[str, str]]]] = None*, *cancelable: bool = False*)

Input group. Request a set of inputs from the user at once.

Parameters

- **label** (*str*) – Label of input group.
- **inputs** (*list*) – Input items. The item of the list is the call to the single input function, and the `name` parameter need to be passed in the single input function.
- **validate** (*callable*) – validation function for the group. If provided, the validation function will be called when the user submits the form.

  Function signature: `callback(data) -> (name, error_msg)`. `validate` receives the value of the entire group as a parameter. When the form value is valid, it returns `None`. When an input item's value is invalid, it returns the `name` value of the item and an error message. For example:

```
def check_form(data):
    if len(data['name']) > 6:
        return ('name', 'Name to long!')
    if data['age'] <= 0:
        return ('age', 'Age cannot be negative!')

data = input_group("Basic info",[
    input('Input your name', name='name'),
    input('Repeat your age', name='age', type=NUMBER)
], validate=check_form)

put_text(data['name'], data['age'])
```

Parameters **cancelable** (*bool*) – Whether the form can be cancelled. Default is `False`. If `cancelable=True`, a "Cancel" button will be displayed at the bottom of the form.

Note: If the last input item in the group is *actions()*, `cancelable` will be ignored.

Returns If the user cancels the form, return `None`, otherwise a `dict` is returned, whose key is the `name` of the input item, and whose value is the value of the input item.

`pywebio.input.`**input_update**(*name: Optional[str] = None*, ***spec*)

Update attributes of input field. This function can only be called in `onchange` callback of input functions.

Parameters

- **name** (*str*) – The `name` of the target input item. Optional, default is the name of input field which triggers `onchange`
- **spec** – The input parameters need to be updated. Note that those parameters can not be updated: `type`, `name`, `validate`, `action`, `code`, `onchange`, `multiple`

An example of implementing dependent input items in an input group:

```
country2city = {
    'China': ['Beijing', 'Shanghai', 'Hong Kong'],
    'USA': ['New York', 'Los Angeles', 'San Francisco'],
}
countries = list(country2city.keys())
location = input_group("Select a location", [
    select('Country', options=countries, name='country',
           onchange=lambda c: input_update('city', options=country2city[c])),
    select('City', options=country2city[countries[0]], name='city'),
])
```

4.3 pywebio.output — Make output to web browser

This module provides functions to output all kinds of content to the user's browser, and supply flexible output control.

4.3.1 Functions list

The following table shows the output-related functions provided by PyWebIO.
The functions marked with `*` indicate that they accept `put_xxx` calls as arguments.
The functions marked with † indicate that they can use as context manager.

	Name	Description
Output Scope	*put_scope*	Create a new scope
	use_scope†	Enter a scope
	get_scope	Get the current scope name in the runtime scope stack
	clear	Clear the content of scope
	remove	Remove the scope
	scroll_to	Scroll the page to the scope

continues on next page

Table 1 – continued from previous page

Content Outputting	*put_text*	Output plain text
	put_markdown	Output Markdown
	put_info[*†] *put_success*[*†] *put_warning*[*†] *put_error*[*†]	Output Messages.
	put_html	Output html
	put_link	Output link
	put_progressbar	Output a progress bar
	put_loading[†]	Output loading prompt
	put_code	Output code block
	put_table[*]	Output table
	put_datatable *datatable_update* *datatable_insert* *datatable_remove*	Output and update data table
	put_button *put_buttons*	Output button and bind click event
	put_image	Output image
	put_file	Output a link to download a file
	put_tabs[*]	Output tabs
	put_collapse[*†]	Output collapsible content
	put_scrollable[*†]	Output a fixed height content area, scroll bar is displayed when the content exceeds the limit
	put_widget[*]	Output your own widget
Other Interactions	*toast*	Show a notification message
	popup[*†]	Show popup
	close_popup	Close the current popup window.
Layout and Style	*put_row*[*†]	Use row layout to output content
	put_column[*†]	Use column layout to output content
	put_grid[*]	Output content using grid layout
	span	Cross-cell content
	style[*]	Customize the css style of output content

4.3.2 Output Scope

See also:

- *Use Guide: Output Scope*

pywebio.output.**put_scope**(*name: str*, *content: Union[pywebio.io_ctrl.Output, List[pywebio.io_ctrl.Output]] = []*, *scope: str = None*, *position: int = - 1*) → pywebio.io_ctrl.Output

Output a scope

Parameters

- **name** (*str*) –
- **content** (*list/put_xxx()*) – The initial content of the scope, can be `put_xxx()` or a list of it.
- **scope, position** (*int*) – Those arguments have the same meaning as for *put_text()*

pywebio.output.**use_scope**(*name=None*, *clear=False*)

Open or enter a scope. Can be used as context manager and decorator.

See *User manual - use_scope()*

Parameters

- **name** (*str*) – Scope name. If it is None, a globally unique scope name is generated. (When used as context manager, the context manager will return the scope name)
- **clear** (*bool*) – Whether to clear the contents of the scope before entering the scope.

Usage

```
with use_scope(...) as scope_name:
    put_xxx()

@use_scope(...)
def app():
    put_xxx()
```

pywebio.output.**get_scope**(*stack_idx: int = - 1*)

Get the scope name of runtime scope stack

Parameters **stack_idx** (*int*) – The index of the runtime scope stack. Default is -1.

0 means the top level scope(the `ROOT` Scope), -1 means the current Scope, -2 means the scope used before entering the current scope, ...

Returns Returns the scope name with the index, and returns `None` when occurs index error

pywebio.output.**clear**(*scope: Optional[str] = None*)

Clear the content of the specified scope

Parameters **scope** (*str*) – Target scope name. Default is the current scope.

pywebio.output.**remove**(*scope: Optional[str] = None*)

Remove the specified scope

Parameters **scope** (*str*) – Target scope name. Default is the current scope.

pywebio.output.**scroll_to**(*scope: Optional[str] = None*, *position: str = 'top'*)

Scroll the page to the specified scope

Parameters

- **scope** (*str*) – Target scope. Default is the current scope.
- **position** (*str*) – Where to place the scope in the visible area of the page. Available value:
  - `'top'` : Keep the scope at the top of the visible area of the page
  - `'middle'` : Keep the scope at the middle of the visible area of the page
  - `'bottom'` : Keep the scope at the bottom of the visible area of the page

4.3.3 Content Outputting

Scope related parameters of output function

The output function will output the content to the "current scope" by default, and the "current scope" for the runtime context can be set by *use_scope()*.

In addition, all output functions support a `scope` parameter to specify the destination scope to output:

```
with use_scope('scope3'):
    put_text('text1 in scope3')   # output to current scope: scope3
    put_text('text in ROOT scope', scope='ROOT')   # output to ROOT Scope

put_text('text2 in scope3', scope='scope3')   # output to scope3
```

The results of the above code are as follows:

```
text1 in scope3
text2 in scope3
text in ROOT scope
```

A scope can contain multiple output items, the default behavior of output function is to append its content to target scope. The `position` parameter of output function can be used to specify the insert position in target scope.

Each output item in a scope has an index, the first item's index is 0, and the next item's index is incremented by one. You can also use a negative number to index the items in the scope, -1 means the last item, -2 means the item before the last, ...

The `position` parameter of output functions accepts an integer. When `position>=0`, it means to insert content before the item whose index equal `position`; when `position<0`, it means to insert content after the item whose index equal `position`:

```
with use_scope('scope1'):
    put_text('A')
    put_text('B', position=0)   # insert B before A -> B A
    put_text('C', position=-2)  # insert C after B -> B C A
    put_text('D', position=1)   # insert D before C B -> B D C A
```

Output functions

pywebio.output.**put_text**(**texts: Any*, *sep: str = ' '*, *inline: bool = False*, *scope: Optional[str] = None*, *position: int = - 1*) → pywebio.io_ctrl.Output

Output plain text

Parameters

- **texts** – Texts need to output. The type can be any object, and the `str()` function will be used for non-string objects.

- **sep** (*str*) – The separator between the texts
- **inline** (*bool*) – Use text as an inline element (no line break at the end of the text). Default is `False`
- **scope** (*str*) – The target scope to output. If the scope does not exist, no operation will be performed.

  Can specify the scope name or use a integer to index the runtime scope stack.
- **position** (*int*) – The position where the content is output in target scope

For more information about `scope` and `position` parameter, please refer to *User Manual*

`pywebio.output.`**put_markdown**(*mdcontent: str*, *lstrip: bool = True*, *options: Optional[Dict[str, Union[str, bool]]] = None*, *sanitize: bool = True*, *scope: Optional[str] = None*, *position: int = - 1*, ***kwargs*) → pywebio.io_ctrl.Output

Output Markdown

Parameters

- **mdcontent** (*str*) – Markdown string
- **lstrip** (*bool*) – Whether to remove the leading whitespace in each line of `mdcontent`. The number of the whitespace to remove will be decided cleverly.
- **options** (*dict*) – Configuration when parsing Markdown. PyWebIO uses marked library to parse Markdown, the parse options see: https://marked.js.org/using_advanced#options (Only supports members of string and boolean type)
- **sanitize** (*bool*) – Whether to use DOMPurify to filter the content to prevent XSS attacks.
- **scope, position** (*int*) – Those arguments have the same meaning as for *put_text()*

When using Python triple quotes syntax to output multi-line Markdown in a function, you can indent the Markdown text to keep a good code format. PyWebIO will cleverly remove the indent for you when show the Markdown:

```
# good code format
def hello():
    put_markdown(r""" # H1
    This is content.
    """)
```

Changed in version 1.5: Enable `lstrip` by default. Deprecate `strip_indent`.

`pywebio.output.`**put_info**(**contents*, *closable=False*, *scope=None*, *position=- 1*) → Output:
`pywebio.output.`**put_success**(**contents*, *closable=False*, *scope=None*, *position=- 1*) → Output:
`pywebio.output.`**put_warning**(**contents*, *closable=False*, *scope=None*, *position=- 1*) → Output:
`pywebio.output.`**put_error**(**contents*, *closable=False*, *scope=None*, *position=- 1*) → Output:

Output Messages.

Parameters

- **contents** – Message contents. The item is `put_xxx()` call, and any other type will be converted to `put_text(content)`.
- **closable** (*bool*) – Whether to show a dismiss button on the right of the message.
- **scope, position** (*int*) – Those arguments have the same meaning as for *put_text()*

New in version 1.2.

pywebio.output.**put_html**(*html: Any*, *sanitize: bool = False*, *scope: Optional[str] = None*, *position: int = - 1*) → pywebio.io_ctrl.Output

Output HTML content

Parameters

- **html** – html string
- **sanitize** (*bool*) – Whether to use DOMPurify to filter the content to prevent XSS attacks.
- **scope, position** (*int*) – Those arguments have the same meaning as for *put_text()*

pywebio.output.**put_link**(*name: str*, *url: Optional[str] = None*, *app: Optional[str] = None*, *new_window: bool = False*, *scope: Optional[str] = None*, *position: int = - 1*) → pywebio.io_ctrl.Output

Output hyperlinks to other web page or PyWebIO Application page.

Parameters

- **name** (*str*) – The label of the link
- **url** (*str*) – Target url
- **app** (*str*) – Target PyWebIO Application name. See also: *Server mode*
- **new_window** (*bool*) – Whether to open the link in a new window
- **scope, position** (*int*) – Those arguments have the same meaning as for *put_text()*

The `url` and `app` parameters must specify one but not both

pywebio.output.**put_progressbar**(*name: str*, *init: float = 0*, *label: Optional[str] = None*, *auto_close: bool = False*, *scope: Optional[str] = None*, *position: int = - 1*) → pywebio.io_ctrl.Output

Output a progress bar

Parameters

- **name** (*str*) – The name of the progress bar, which is the unique identifier of the progress bar
- **init** (*float*) – The initial progress value of the progress bar. The value is between 0 and 1
- **label** (*str*) – The label of progress bar. The default is the percentage value of the current progress.
- **auto_close** (*bool*) – Whether to remove the progress bar after the progress is completed
- **scope, position** (*int*) – Those arguments have the same meaning as for *put_text()*

Example:

```
import time

put_progressbar('bar');
for i in range(1, 11):
    set_progressbar('bar', i / 10)
    time.sleep(0.1)
```

See also:

use *set_progressbar()* to set the progress of progress bar

pywebio.output.**set_progressbar**(*name: str*, *value: float*, *label: Optional[str] = None*)
Set the progress of progress bar

Parameters

- **name** (*str*) – The name of the progress bar
- **value** (*float*) – The progress value of the progress bar. The value is between 0 and 1
- **label** (*str*) – The label of progress bar. The default is the percentage value of the current progress.

See also: *put_progressbar()*

pywebio.output.**put_loading**(*shape: str = 'border'*, *color: str = 'dark'*, *scope: Optional[str] = None*, *position: int = - 1*) → pywebio.io_ctrl.Output
Output loading prompt

Parameters

- **shape** (*str*) – The shape of loading prompt. The available values are: `'border'` (default) `'grow'`
- **color** (*str*) – The color of loading prompt. The available values are: `'primary'` `'secondary'` `'success'` `'danger'` `'warning'` `'info'` `'light'` `'dark'` (default)
- **scope, position** (*int*) – Those arguments have the same meaning as for *put_text()*

put_loading() can be used in 2 ways: direct call and context manager:

```
for shape in ('border', 'grow'):
    for color in ('primary', 'secondary', 'success', 'danger', 'warning', 'info',
↪'light', 'dark'):
        put_text(shape, color)
        put_loading(shape=shape, color=color)

# The loading prompt and the output inside the context will disappear
# automatically when the context block exits.
with put_loading():
    put_text("Start waiting...")
    time.sleep(3)  # Some time-consuming operations
put_text("The answer of the universe is 42")

# using style() to set the size of the loading prompt
put_loading().style('width:4rem; height:4rem')
```

Changed in version 1.8: when use *put_loading()* as context manager, the output inside the context will also been removed after the context block exits.

pywebio.output.**put_code**(*content: str*, *language: str = ''*, *rows: Optional[int] = None*, *scope: Optional[str] = None*, *position: int = - 1*) → pywebio.io_ctrl.Output
Output code block

Parameters

- **content** (*str*) – code string
- **language** (*str*) – language of code

- **rows** (*int*) – The max lines of code can be displayed, no limit by default. The scroll bar will be displayed when the content exceeds.
- **scope, position** (*int*) – Those arguments have the same meaning as for *put_text()*

pywebio.output.**put_table**(*tdata: List[Union[List, Dict]]*, *header: List[Union[str, Tuple[Any, str]]] = None*, *scope: str = None*, *position: int = - 1*) → pywebio.io_ctrl.Output

Output table

Parameters

- **tdata** (*list*) – Table data, which can be a two-dimensional list or a list of dict. The table cell can be a string or `put_xxx()` call. The cell can use the *span()* to set the cell span.
- **header** (*list*) – Table header. When the item of `tdata` is of type `list`, if the `header` parameter is omitted, the first item of `tdata` will be used as the header. The header item can also use the *span()* function to set the cell span.

  When `tdata` is list of dict, `header` can be used to specify the order of table headers. In this case, the `header` can be a list of dict key or a list of `(<label>, <dict key>)`.
- **scope, position** (*int*) – Those arguments have the same meaning as for *put_text()*

Example:

```
# 'Name' cell across 2 rows, 'Address' cell across 2 columns
put_table([
    [span('Name',row=2), span('Address', col=2)],
    ['City', 'Country'],
    ['Wang', 'Beijing', 'China'],
    ['Liu', 'New York', 'America'],
])

# Use `put_xxx()` in `put_table()`
put_table([
    ['Type', 'Content'],
    ['html', put_html('X<sup>2</sup>')],
    ['text', '<hr/>'],
    ['buttons', put_buttons(['A', 'B'], onclick=...)],
    ['markdown', put_markdown('`Awesome PyWebIO!`')],
    ['file', put_file('hello.text', b'hello world')],
    ['table', put_table([['A', 'B'], ['C', 'D']])]
])

# Set table header
put_table([
    ['Wang', 'M', 'China'],
    ['Liu', 'W', 'America'],
], header=['Name', 'Gender', 'Address'])

# When ``tdata`` is list of dict
put_table([
    {"Course":"OS", "Score": "80"},
    {"Course":"DB", "Score": "93"},
], header=["Course", "Score"])  # or header=[(put_markdown("*Course*"), "Course"),
↪ (put_markdown("*Score*") ,"Score")]
```

New in version 0.3: The cell of table support `put_xxx()` calls.

pywebio.output.**span**(*content: Union[str, pywebio.io_ctrl.Output]*, *row: int = 1*, *col: int = 1*)

Create cross-cell content in *put_table()* and *put_grid()*

Parameters

- **content** – cell content. It can be a string or `put_xxx()` call.
- **row** (*int*) – Vertical span, that is, the number of spanning rows
- **col** (*int*) – Horizontal span, that is, the number of spanning columns

Example

```
put_table([
    ['C'],
    [span('E', col=2)],  # 'E' across 2 columns
], header=[span('A', row=2), 'B'])  # 'A' across 2 rows

put_grid([
    [put_text('A'), put_text('B')],
    [span(put_text('A'), col=2)],  # 'A' across 2 columns
])
```

pywebio.output.**put_buttons**(*buttons: List[Union[Dict[str, Any], Tuple[str, Any], List, str]]*, *onclick: Union[Callable[[Any], None], Sequence[Callable[], None]]]*, *small: Optional[bool] = None*, *link_style: bool = False*, *outline: bool = False*, *group: bool = False*, *scope: Optional[str] = None*, *position: int = - 1*, ***callback_options*) → pywebio.io_ctrl.Output

Output a group of buttons and bind click event

Parameters

- **buttons** (*list*) – Button list. The available formats of list items are:
  - dict:

    ```
    {
        "label":(str)button label,
        "value":(str)button value,
        "color":(str, optional)button color,
        "disabled":(bool, optional) whether the button is disabled
    }
    ```

  - tuple or list: `(label, value)`
  - single value: label and value of option use the same value

  The `value` of button can be any type. The `color` of button can be one of: `primary`, `secondary`, `success`, `danger`, `warning`, `info`, `light`, `dark`.

  Example:

  ```
  put_buttons([dict(label='success', value='s', color='success')],
  ↪onclick=...)
  ```

- **onclick** (*callable / list*) – Callback which will be called when button is clicked. `onclick` can be a callable object or a list of it.

  If `onclick` is callable object, its signature is `onclick(btn_value)`. `btn_value` is `value` of the button that is clicked.

  If `onclick` is a list, the item receives no parameter. In this case, each item in the list corresponds to the buttons one-to-one.

Tip: You can use `functools.partial` to save more context information in `onclick`.

Note: When in *Coroutine-based session*, the callback can be a coroutine function.

- **small** (*bool*) – Whether to use small size button. Default is False.
- **link_style** (*bool*) – Whether to use link style button. Default is False
- **outline** (*bool*) – Whether to use outline style button. Default is False
- **group** (*bool*) – Whether to group the buttons together. Default is False
- **scope, position** (*int*) – Those arguments have the same meaning as for *put_text()*
- **callback_options** – Other options of the `onclick` callback. There are different options according to the session implementation

  When in Coroutine-based Session:

  – mutex_mode: Default is `False`. If set to `True`, new click event will be ignored when the current callback is running. This option is available only when `onclick` is a coroutine function.

  When in Thread-based Session:

  – serial_mode: Default is `False`, and every time a callback is triggered, the callback function will be executed immediately in a new thread.

  If set `serial_mode` to `True` After enabling serial_mode, the button's callback will be executed serially in a resident thread in the session, and all other new click event callbacks (including the `serial_mode=False` callback) will be queued for the current click event to complete. If the callback function runs for a short time, you can turn on `serial_mode` to improve performance.

Example:

```
from functools import partial

def row_action(choice, id):
    put_text("You click %s button with id: %s" % (choice, id))

put_buttons(['edit', 'delete'], onclick=partial(row_action, id=1))

def edit():
    put_text("You click edit button")
def delete():
    put_text("You click delete button")

put_buttons(['edit', 'delete'], onclick=[edit, delete])
```

Changed in version 1.5: Add `disabled` button support. The `value` of button can be any object.

pywebio.output.**put_button**(*label: str*, *onclick: Callable[], None]*, *color: Optional[str] = None*, *small: Optional[bool] = None*, *link_style: bool = False*, *outline: bool = False*, *disabled: bool = False*, *scope: Optional[str] = None*, *position: int = - 1*) → pywebio.io_ctrl.Output

Output a single button and bind click event to it.

Parameters

- **label** (*str*) – Button label
- **onclick** (*callable*) – Callback which will be called when button is clicked.

- **color** (*str*) – The color of the button, can be one of: `primary`, `secondary`, `success`, `danger`, `warning`, `info`, `light`, `dark`.
- **disabled** (*bool*) – Whether the button is disabled
- **small, link_style, outline, scope, position** (–) – Those arguments have the same meaning as for *put_buttons()*

Example:

```
put_button("click me", onclick=lambda: toast("Clicked"), color='success',␣
↪outline=True)
```

New in version 1.4.

Changed in version 1.5: add `disabled` parameter

pywebio.output.**put_image**(*src: Union[str, bytes, PIL.Image.Image]*, *format: Optional[str] = None*, *title: str = ''*, *width: Optional[str] = None*, *height: Optional[str] = None*, *scope: Optional[str] = None*, *position: int = - 1*) → pywebio.io_ctrl.Output

Output image

Parameters

- **src** – Source of image. It can be a string specifying image URL, a bytes-like object specifying the binary content of an image or an instance of `PIL.Image.Image`
- **title** (*str*) – Image description.
- **width** (*str*) – The width of image. It can be CSS pixels (like `'30px'`) or percentage (like `'10%'`).
- **height** (*str*) – The height of image. It can be CSS pixels (like `'30px'`) or percentage (like `'10%'`). If only one value of `width` and `height` is specified, the browser will scale image according to its original size.
- **format** (*str*) – Image format, optinoal. e.g.: `png`, `jpeg`, `gif`, etc. Only available when `src` is non-URL
- **scope, position** (*int*) – Those arguments have the same meaning as for *put_text()*

Example:

```
img = open('/path/to/some/image.png', 'rb').read()
put_image(img, width='50px')

put_image('https://www.python.org/static/img/python-logo.png')
```

pywebio.output.**put_file**(*name: str*, *content: bytes*, *label: Optional[str] = None*, *scope: Optional[str] = None*, *position: int = - 1*) → pywebio.io_ctrl.Output

Output a link to download a file

To show a link with the file name on the browser. When click the link, the browser automatically downloads the file.

Parameters

- **name** (*str*) – File name downloaded as
- **content** – File content. It is a bytes-like object

- **label** (*str*) – The label of the download link, which is the same as the file name by default.
- **scope, position** (*int*) – Those arguments have the same meaning as for *put_text()*

Example:

```
content = open('./some-file', 'rb').read()
put_file('hello-world.txt', content, 'download me')
```

pywebio.output.**put_tabs**(*tabs: List[Dict[str, Any]]*, *scope: str = None*, *position: int = - 1*) → pywebio.io_ctrl.Output

Output tabs.

Parameters

- **tabs** (*list*) – Tab list, each item is a dict: `{"title": "Title", "content": ...}` . The `content` can be a string, the `put_xxx()` calls , or a list of them.
- **scope, position** (*int*) – Those arguments have the same meaning as for *put_text()*

```
put_tabs([
    {'title': 'Text', 'content': 'Hello world'},
    {'title': 'Markdown', 'content': put_markdown('~~Strikethrough~~')},
    {'title': 'More content', 'content': [
        put_table([
            ['Commodity', 'Price'],
            ['Apple', '5.5'],
            ['Banana', '7'],
        ]),
        put_link('pywebio', 'https://github.com/wang0618/PyWebIO')
    ]},
])
```

New in version 1.3.

pywebio.output.**put_collapse**(*title: str*, *content: Union[str, pywebio.io_ctrl.Output, List[Union[str, pywebio.io_ctrl.Output]]] = []*, *open: bool = False*, *scope: str = None*, *position: int = - 1*) → pywebio.io_ctrl.Output

Output collapsible content

Parameters

- **title** (*str*) – Title of content
- **content** (*list/str/put_xxx()*) – The content can be a string, the `put_xxx()` calls , or a list of them.
- **open** (*bool*) – Whether to expand the content. Default is `False`.
- **scope, position** (*int*) – Those arguments have the same meaning as for *put_text()*

Example:

```
put_collapse('Collapse title', [
    'text',
    put_markdown('~~Strikethrough~~'),
    put_table([
        ['Commodity', 'Price'],
```

(continues on next page)

(continued from previous page)

```
        ['Apple', '5.5'],
    ])
], open=True)

put_collapse('Large text', 'Awesome PyWebIO! '*30)
```

pywebio.output.**put_scrollable**(*content: Union[str, pywebio.io_ctrl.Output, List[Union[str, pywebio.io_ctrl.Output]]] = []*, *height: Union[int, Tuple[int, int]] = 400*, *keep_bottom: bool = False*, *border: bool = True*, *scope: str = None*, *position: int = - 1*, ***kwargs*) → pywebio.io_ctrl.Output

Output a fixed height content area. scroll bar is displayed when the content exceeds the limit

Parameters

- **content** (*list/str/put_xxx()*) – The content can be a string, the `put_xxx()` calls , or a list of them.
- **height** (*int/tuple*) – The height of the area (in pixels). `height` parameter also accepts `(min_height, max_height)` to indicate the range of height, for example, `(100, 200)` means that the area has a minimum height of 100 pixels and a maximum of 200 pixels. Set `None` if you don't want to limit the height
- **keep_bottom** (*bool*) – Whether to keep the content area scrolled to the bottom when updated.
- **border** (*bool*) – Whether to show border
- **scope, position** (*int*) – Those arguments have the same meaning as for *put_text()*

Example:

```
import time

put_scrollable(put_scope('scrollable'), height=200, keep_bottom=True)
put_text("You can click the area to prevent auto scroll.", scope='scrollable')

while 1:
    put_text(time.time(), scope='scrollable')
    time.sleep(0.5)
```

Changed in version 1.1: add `height` parameterremove `max_height` parameter add `keep_bottom` parameter

Changed in version 1.5: remove `horizon_scroll` parameter

pywebio.output.**put_datatable**(*records: Sequence[Mapping]*, *actions: Sequence[Tuple[str, Callable[[Union[str, int, List[Union[str, int]]]], None]]] = None*, *onselect: Callable[[Union[str, int, List[Union[str, int]]]], None] = None*, *multiple_select=False*, *id_field: str = None*, *height: Union[str, int] = 600*, *theme: Literal['alpine', 'alpine-dark', 'balham', 'balham-dark', 'material'] = 'balham'*, *cell_content_bar=True*, *instance_id=''*, *column_order: Union[Sequence[str], Mapping] = None*, *column_args: Mapping[Union[str, Tuple], Mapping] = None*, *grid_args: Mapping[str, Mapping] = None*, *enterprise_key=''*, *scope: str = None*, *position: int = - 1*) → pywebio.io_ctrl.Output

Output a datatable.

Compared with *put_table()*, *put_datatable()* is more suitable for displaying large amounts of data (both data fields and data entries), while *put_table()* is more suitable for displaying diverse data types (pictures, buttons, etc.) in cells.

This widget is powered by the awesome ag-grid library.

Parameters

- **records** (*list[dict]*) – data of rows, each row is a python `dict`, which can be nested.
- **actions** (*list*) – actions for selected row(s), they will be shown as buttons when row is selected. The format of the action item: `(button_label:str, on_click:callable)`. Specifically, `None` item is allowed, which will be rendered as a separator. The `on_click` callback receives the selected row ID as parameter.
- **onselect** (*callable*) – callback when row is selected, receives the selected row ID as parameter.
- **multiple_select** (*bool*) – whether multiple rows can be selected. When enabled, the `on_click` callback in `actions` and the `onselect` callback will receive ID list of selected rows as parameter.
- **id_field** (*str/tuple*) – row ID field, that is, the key of the row dict to uniquely identifies a row. When not provide, the datatable will use the index in `records` to assign row ID.

  To specify the ID field of a nested dict, use a tuple to specify the path of the ID field. For example, if the row record is in `{'a': {'b': ...}}` format, you can use `id_field=('a', 'b')` to set `'b'` column as the ID field.
- **height** (*int/str*) – widget height. When pass `int` type, the unit is pixel, when pass `str` type, you can specify any valid CSS height value. In particular, you can use `'auto'` to make the datatable auto-size it's height to fit the content.
- **theme** (*str*) – datatable theme. Available themes are: `'balham'` (default), `'alpine'`, `'alpine-dark'`, `'balham-dark'`, `'material'`. You can preview the themes in ag-grid official example.
- **cell_content_bar** (*bool*) – whether to add a text bar to datatable to show the content of current focused cell. Default is `True`.
- **instance_id** (*str*) – Assign a unique ID to the datatable, so that you can refer this datatable in *datatable_update()*, *datatable_insert()* and *datatable_remove()* functions.
- **column_order** (*list*) – column order, the order of the column names in the list will be used as the column order. If not provided, the column order will be the same as the order of the keys in the first row of `records`. When provided, the column not in the list will not be shown.

  Since the `dict` in python is ordered after py3.7, you can use dict to specify the column order when the row record is nested dict. For example:

```
column_order = {'a': {'b': {'c': None, 'd': None}, 'e': None}, 'f
↪': None}
```

- **column_args** – column properties. Dict type, the key is str to specify the column field, the value is ag-grid column properties in dict.

  Given the row record is in this format:

```
{
    "a": {"b": ..., "c": ...},
    "b": ...,
    "c": ...
}
```

When you set `column_args={"b": settings}`, the column settings will be applied to the column `a.b` and `b`. Use tuple as key to specify the nested key path, for example, `column_args={("a", "b"): settings}` will only apply the settings to column `a.b`.

- **grid_args** – ag-grid grid options. Refer ag-grid doc - grid options for more information.
- **enterprise_key** (*str*) – ag-grid enterprise license key. When not provided, will use the ag-grid community version.

The ag-grid library is so powerful, and you can use the `column_args` and `grid_args` parameters to achieve high customization.

Example of `put_datatable()`:

```
import urllib.request
import json

with urllib.request.urlopen('https://fakerapi.it/api/v1/persons?_quantity=30') as␣
↪f:
    data = json.load(f)['data']

put_datatable(
    data,
    actions=[
        ("Edit Email", lambda row_id: datatable_update('user', input("Email"),␣
↪row_id, "email")),
        ("Insert a Row", lambda row_id: datatable_insert('user', data[0], row_
↪id)),
        None,  # separator
        ("Delete", lambda row_id: datatable_remove('user', row_id)),
    ],
    onselect=lambda row_id: toast(f'Selected row: {row_id}'),
    instance_id='user'
)
```

The ag-grid instance can be accessed with JS global variable `ag_grid_${instance_id}_promise`:

```
ag_grid_xxx_promise.then(function(gridOptions) {
    // gridOptions is the ag-grid gridOptions object
    gridOptions.columnApi.autoSizeAllColumns();
});
```

To pass JS functions as value of `column_args` or `grid_args`, you can use `JSFunction` object:

`pywebio.output.`**`JSFunction`**(*[param1][, param2], ...[, param n], body*)

Example:

```
put_datatable(..., grid_args=dict(sortChanged=JSFunction("event",
↪"console.log(event.source)")))
```

Since the ag-grid don't native support nested dict as row record, PyWebIO will internally flatten the nested dict before passing to ag-grid. So when you access or modify data in ag-grid directly, you need to use the following

functions to help you convert the data:

- `gridOptions.flatten_row(nested_dict_record)`: flatten the nested dict record to a flat dict record
- `gridOptions.path2field(field_path_array)`: convert the field path array to field name used in ag-grid
- `gridOptions.field2path(ag_grid_column_field_name)`: convert the field name back to field path array

The implement of *datatable_update()*, *datatable_insert* and *datatable_remove* functions are good examples to show how to interact with ag-grid in Javascript.

pywebio.output.**datatable_update**(*instance_id: str*, *data: Any*, *row_id: Optional[Union[str, int]] = None*, *field: Optional[Union[str, List[str], Tuple[str]]] = None*)

Update the whole data / a row / a cell of the datatable.

To use *datatable_update()*, you need to specify the `instance_id` parameter when calling *put_datatable()*.

When `row_id` and `field` is not specified (`datatable_update(instance_id, data)`), the whole data of datatable will be updated, in this case, the `data` parameter should be a list of dict (same as `records` in *put_datatable()*).

To update a row, specify the `row_id` parameter and pass the row data in dict to `data` parameter (`datatable_update(instance_id, data, row_id)`). See `id_field` of *put_datatable()* for more info of `row_id`.

To update a cell, specify the `row_id` and `field` parameters, in this case, the `data` parameter should be the cell value To update a row, specify the `row_id` parameter and pass the row data in dict to `data` parameter (`datatable_update(instance_id, data, row_id, field)`). The `field` can be a tuple to indicate nested key path.

pywebio.output.**datatable_insert**(*instance_id: str*, *records: List*, *row_id=None*)

Insert rows to datatable.

Parameters

- **instance_id** (*str*) – Datatable instance id (i.e., the `instance_id` parameter when calling *put_datatable()*)
- **records** (*dict/list[dict]*) – row record or row record list to insert
- **row_id** (*str/int*) – row id to insert before, if not specified, insert to the end

Note: When use `id_field=None` (default) in *put_datatable()*, the row id of new inserted rows will auto increase from the last max row id.

pywebio.output.**datatable_remove**(*instance_id: str*, *row_ids: List*)

Remove rows from datatable.

Parameters

- **instance_id** (*str*) – Datatable instance id (i.e., the `instance_id` parameter when calling *put_datatable()*)
- **row_ids** (*int/str/list*) – row id or row id list to remove

pywebio.output.**put_widget**(*template: str*, *data: Dict[str, Any]*, *scope: str = None*, *position: int = -1*) → pywebio.io_ctrl.Output

Output your own widget

Parameters

- **template** – html template, using mustache.js syntax
- **data** (*dict*) – Data used to render the template.

  The data can include the `put_xxx()` calls, and the JS function `pywebio_output_parse` can be used to parse the content of `put_xxx()`. For string input, `pywebio_output_parse` will parse into text.

  When using the `pywebio_output_parse` function, you need to turn off the html escaping of mustache: `{{& pywebio_output_parse}}`, see the example below.
- **scope, position** (*int*) – Those arguments have the same meaning as for *put_text()*

Example

```
tpl = '''
<details {{#open}}open{{/open}}>
    <summary>{{title}}</summary>
    {{#contents}}
        {{& pywebio_output_parse}}
    {{/contents}}
</details>
'''

put_widget(tpl, {
    "open": True,
    "title": 'More content',
    "contents": [
        'text',
        put_markdown('~~Strikethrough~~'),
        put_table([
            ['Commodity', 'Price'],
            ['Apple', '5.5'],
            ['Banana', '7'],
        ])
    ]
})
```

4.3.4 Other Interactions

pywebio.output.**toast**(*content: str*, *duration: float = 2*, *position: str = 'center'*, *color: str = 'info'*, *onclick: Optional[Callable[], None]] = None*)

Show a notification message.

Parameters

- **content** (*str*) – Notification content.
- **duration** (*float*) – The duration of the notification display, in seconds. `0` means not to close automatically (at this time, a close button will be displayed next to the message, and the user can close the message manually)
- **position** (*str*) – Where to display the notification message. Available values are `'left'`, `'center'` and `'right'`.
- **color** (*str*) – Background color of the notification. Available values are `'info'`, `'error'`, `'warn'`, `'success'` or hexadecimal color value starting with `'#'`

- **onclick** (*callable*) – The callback function when the notification message is clicked. The callback function receives no parameters.

  Note: When in *Coroutine-based session*, the callback can be a coroutine function.

Example:

```
def show_msg():
    put_text("You clicked the notification.")

toast('New messages', position='right', color='#2188ff', duration=0, onclick=show_
↪msg)
```

pywebio.output.**popup**(*title: str*, *content: Union[str, pywebio.io_ctrl.Output, List[Union[str, pywebio.io_ctrl.Output]]] = None*, *size: str = 'normal'*, *implicit_close: bool = True*, *closable: bool = True*)

Show a popup.

: In PyWebIO, you can't show multiple popup windows at the same time. Before displaying a new pop-up window, the existing popup on the page will be automatically closed. You can use *close_popup()* to close the popup manually.

Parameters

- **title** (*str*) – The title of the popup.
- **content** (*list/str/put_xxx()*) – The content of the popup. Can be a string, the put_xxx() calls, or a list of them.
- **size** (*str*) – The size of popup window. Available values are: `'large'`, `'normal'` and `'small'`.
- **implicit_close** (*bool*) – If enabled, the popup can be closed implicitly by clicking the content outside the popup window or pressing the `Esc` key. Default is `False`.
- **closable** (*bool*) – Whether the user can close the popup window. By default, the user can close the popup by clicking the close button in the upper right of the popup window. When set to `False`, the popup window can only be closed by `popup_close()`, at this time the `implicit_close` parameter will be ignored.

`popup()` can be used in 2 ways: direct call and context manager.

- direct call:

```
popup('popup title', 'popup text content', size=PopupSize.SMALL)

popup('Popup title', [
    put_html('<h3>Popup Content</h3>'),
    'html: <br/>',
    put_table([['A', 'B'], ['C', 'D']]),
    put_buttons(['close_popup()'], onclick=lambda _: close_popup())
])
```

- context manager:

```
with popup('Popup title') as s:
    put_html('<h3>Popup Content</h3>')
    put_text('html: <br/>')
    put_buttons([('clear()', s)], onclick=clear)
```

(continues on next page)

(continued from previous page)

```
put_text('Also work!', scope=s)
```

The context manager will open a new output scope and return the scope name. The output in the context manager will be displayed on the popup window by default. After the context manager exits, the popup window will not be closed. You can still use the `scope` parameter of the output function to output to the popup.

pywebio.output.**close_popup**()
Close the current popup window.

See also: *popup()*

4.3.5 Layout and Style

pywebio.output.**put_row**(*content: List[Optional[pywebio.io_ctrl.Output]] = []*, *size: str = None*, *scope: str = None*, *position: int = - 1*) → pywebio.io_ctrl.Output
Use row layout to output content. The content is arranged horizontally

Parameters

- **content** (*list*) – Content list, the item is `put_xxx()` call or `None`. `None` represents the space between the output
- **size** (*str*) –

  Used to indicate the width of the items, is a list of width values separated by space.
  Each width value corresponds to the items one-to-one. (`None` item should also correspond to a width value).
  By default, `size` assigns a width of 10 pixels to the `None` item, and distributes the width equally to the remaining items.

  Available format of width value are:

  - pixels: like `100px`
  - percentage: Indicates the percentage of available width. like `33.33%`
  - `fr` keyword: Represents a scale relationship, 2fr represents twice the width of 1fr
  - `auto` keyword: Indicates that the length is determined by the browser
  - `minmax(min, max)` : Generate a length range, indicating that the length is within this range. It accepts two parameters, minimum and maximum. For example: `minmax(100px, 1fr)` means the length is not less than 100px and not more than 1fr
- **scope, position** (*int*) – Those arguments have the same meaning as for *put_text()*

Example

```
# Two code blocks of equal width, separated by 10 pixels
put_row([put_code('A'), None, put_code('B')])

# The width ratio of the left and right code blocks is 2:3, which is equivalent␣
↪to size='2fr 10px 3fr'
put_row([put_code('A'), None, put_code('B')], size='40% 10px 60%')
```

pywebio.output.**put_column**(*content: List[Optional[pywebio.io_ctrl.Output]] = []*, *size: str = None*, *scope: str = None*, *position: int = - 1*) → pywebio.io_ctrl.Output

Use column layout to output content. The content is arranged vertically

Parameters

- **content** (*list*) – Content list, the item is `put_xxx()` call or `None`. `None` represents the space between the output
- **size** (*str*) – Used to indicate the width of the items, is a list of width values separated by space. The format is the same as the `size` parameter of the *put_row()* function.
- **scope, position** (*int*) – Those arguments have the same meaning as for *put_text()*

pywebio.output.**put_grid**(*content: List[List[Optional[pywebio.io_ctrl.Output]]]*, *cell_width: str = 'auto'*, *cell_height: str = 'auto'*, *cell_widths: str = None*, *cell_heights: str = None*, *direction: str = 'row'*, *scope: str = None*, *position: int = - 1*) → pywebio.io_ctrl.Output

Output content using grid layout

Parameters

- **content** – Content of grid, which is a two-dimensional list. The item of list is `put_xxx()` call or `None`. `None` represents the space between the output. The item can use the *span()* to set the cell span.
- **cell_width** (*str*) – The width of grid cell.
- **cell_height** (*str*) – The height of grid cell.
- **cell_widths** (*str*) – The width of each column of the grid. The width values are separated by a space. Can not use `cell_widths` and `cell_width` at the same time
- **cell_heights** (*str*) – The height of each row of the grid. The height values are separated by a space. Can not use `cell_heights` and `cell_height` at the same time
- **direction** (*str*) – Controls how auto-placed items get inserted in the grid. Can be `'row'``(default) or ``'column'`.

  `'row'` : Places items by filling each row

  `'column'` : Places items by filling each column
- **scope, position** (*int*) – Those arguments have the same meaning as for *put_text()*

The format of width/height value in `cell_width`,``cell_height``,``cell_widths``,``cell_heights`` can refer to the `size` parameter of the *put_row()* function.

Example:

```
put_grid([
    [put_text('A'), put_text('B'), put_text('C')],
    [None, span(put_text('D'), col=2, row=1)],
    [put_text('E'), put_text('F'), put_text('G')],
], cell_width='100px', cell_height='100px')
```

pywebio.output.**style**(*outputs: Union[pywebio.io_ctrl.Output, List[pywebio.io_ctrl.Output]]*, *css_style: str*) → Union[pywebio.io_ctrl.Output, pywebio.io_ctrl.OutputList]

Customize the css style of output content

Deprecated since version 1.3: See *User Guide* for new way to set css style for output.

Parameters

- **outputs** (*list/put_xxx()*) – The output content can be a `put_xxx()` call or a list of it.
- **css_style** (*str*) – css style string

Returns

The output contents with css style added:

Note: If `outputs` is a list of `put_xxx()` calls, the style will be set for each item of the list. And the return value can be used in anywhere accept a list of `put_xxx()` calls.

Example

```
style(put_text('Red'), 'color:red')

style([
    put_text('Red'),
    put_markdown('~~del~~')
], 'color:red')

put_table([
    ['A', 'B'],
    ['C', style(put_text('Red'), 'color:red')],
])

put_collapse('title', style([
    put_text('text'),
    put_markdown('~~del~~'),
], 'margin-left:20px'))
```

4.4 pywebio.session — More control to session

pywebio.session.**download**(*name*, *content*)

Send file to user, and the user browser will download the file to the local

Parameters

- **name** (*str*) – File name when downloading
- **content** – File content. It is a bytes-like object

Example:

```
put_button('Click to download', lambda: download('hello-world.txt', b'hello world!
↪'))
```

pywebio.session.**run_js**(*code_*, ***args*)

Execute JavaScript code in user browser.

The code is run in the browser's JS global scope.

Parameters

- **code** (*str*) – JavaScript code
- **args** – Local variables passed to js code. Variables need to be JSON-serializable.

Example:

```
run_js('console.log(a + b)', a=1, b=2)
```

pywebio.session.**eval_js**(*expression_*, ***args*)

Execute JavaScript expression in the user's browser and get the value of the expression

Parameters

- **expression** (*str*) – JavaScript expression. The value of the expression need to be JSON-serializable. If the value of the expression is a promise, `eval_js()` will wait for the promise to resolve and return the value of it. When the promise is rejected, `None` is returned.
- **args** – Local variables passed to js code. Variables need to be JSON-serializable.

Returns The value of the expression.

Note: When using *coroutine-based session*, you need to use the `await eval_js(expression)` syntax to call the function.

Example:

```
current_url = eval_js("window.location.href")

function_res = eval_js('''(function(){
    var a = 1;
    a += b;
    return a;
})()''', b=100)

promise_res = eval_js('''new Promise(resolve => {
    setTimeout(() => {
        resolve('Returned inside callback.');
    }, 2000);
});''')
```

Changed in version 1.3: The JS expression support return promise.

pywebio.session.**register_thread**(*thread: threading.Thread*)

Register the thread so that PyWebIO interactive functions are available in the thread.

Can only be used in the thread-based session.

See *Concurrent in Server mode*

Parameters **thread** (*threading.Thread*) – Thread object

pywebio.session.**defer_call**(*func*)

Set the function to be called when the session closes.

Whether it is because the user closes the page or the task finishes to cause session closed, the function set by `defer_call(func)` will be executed. Can be used for resource cleaning.

You can call `defer_call(func)` multiple times in the session, and the set functions will be executed sequentially after the session closes.

`defer_call()` can also be used as decorator:

```
@defer_call
def cleanup():
    pass
```

> **Attention:** PyWebIO interactive functions cannot be called inside the deferred functions.

pywebio.session.**local**

The session-local object for current session.

`local` is a dictionary object that can be accessed through attributes, it aim to be used to save some session-local state of your application. Attributes of `local` are not shared between sessions, each session sees only the attributes it itself placed in there.

Usage Scenes

When you need to share some session-independent data with multiple functions, it is more convenient to use session-local objects to save state than to use function parameters.

Here is a example of a session independent counter implementation:

```
from pywebio.session import local
def add():
    local.cnt = (local.cnt or 0) + 1

def show():
    put_text(local.cnt or 0)

def main():
    put_buttons(['Add counter', 'Show counter'], [add, show])
```

The way to pass state through function parameters is:

```
from functools import partial
def add(cnt):
    cnt[0] += 1

def show(cnt):
    put_text(cnt[0])

def main():
    cnt = [0]  # Trick: to pass by reference
    put_buttons(['Add counter', 'Show counter'], [partial(add, cnt), partial(show,
↪ cnt)])
```

Of course, you can also use function closures to achieved the same:

```
def main():
    cnt = 0

    def add():
        nonlocal cnt
        cnt += 1

    def show():
        put_text(cnt)

    put_buttons(['Add counter', 'Show counter'], [add, show])
```

Operations supported by local object

`local` is a dictionary object that can be accessed through attributes. When accessing a property that does not

exist in the data object, it returns `None` instead of throwing an exception. The method of dictionary is not supported in `local`. It supports the `in` operator to determine whether the key exists. You can use `local._dict` to get the underlying dictionary data.

```
local.name = "Wang"
local.age = 22
assert local.foo is None
local[10] = "10"

for key in local:
    print(key)

assert 'bar' not in local
assert 'name' in local

print(local._dict)
```

New in version 1.1.

pywebio.session.**set_env**(***env_info*)

configure the environment of current session.

Available configuration are:

- `title` (str): Title of current page.
- `output_animation` (bool): Whether to enable output animation, enabled by default
- `auto_scroll_bottom` (bool): Whether to automatically scroll the page to the bottom after output content, it is closed by default. Note that after enabled, only outputting to ROOT scope can trigger automatic scrolling.
- `http_pull_interval` (int): The period of HTTP polling messages (in milliseconds, default 1000ms), only available in sessions based on HTTP connection.
- `input_panel_fixed` (bool): Whether to make input panel fixed at bottom, enabled by default
- `input_panel_min_height` (int): The minimum height of input panel (in pixel, default 300px), it should be larger than 75px. Available only when `input_panel_fixed=True`
- `input_panel_init_height` (int): The initial height of input panel (in pixel, default 300px), it should be larger than 175px. Available only when `input_panel_fixed=True`
- `input_auto_focus` (bool): Whether to focus on input automatically after showing input panel, default is `True`
- `output_max_width` (str): The max width of the page content area (in pixel or percentage, e.g. `'1080px'`, `'80%'`. Default is 880px).

Example:

```
set_env(title='Awesome PyWebIO!!', output_animation=False)
```

Changed in version 1.4: Added the `output_max_width` parameter

pywebio.session.**go_app**(*name*, *new_window=True*)

Jump to another task of a same PyWebIO application. Only available in PyWebIO Server mode

Parameters

- **name** (*str*) – Target PyWebIO task name.
- **new_window** (*bool*) – Whether to open in a new window, the default is `True`

See also: *Server mode*

pywebio.session.**info**

The session information data object, whose attributes are:

- `user_agent` : The Object of the user browser information, whose attributes are
  - `is_mobile` (bool): whether user agent is identified as a mobile phone (iPhone, Android phones, Blackberry, Windows Phone devices etc)
  - `is_tablet` (bool): whether user agent is identified as a tablet device (iPad, Kindle Fire, Nexus 7 etc)
  - `is_pc` (bool): whether user agent is identified to be running a traditional "desktop" OS (Windows, OS X, Linux)
  - `is_touch_capable` (bool): whether user agent has touch capabilities
  - `browser.family` (str): Browser family. such as 'Mobile Safari'
  - `browser.version` (tuple): Browser version. such as (5, 1)
  - `browser.version_string` (str): Browser version string. such as '5.1'
  - `os.family` (str): User OS family. such as 'iOS'
  - `os.version` (tuple): User OS version. such as (5, 1)
  - `os.version_string` (str): User OS version string. such as '5.1'
  - `device.family` (str): User agent's device family. such as 'iPhone'
  - `device.brand` (str): Device brand. such as 'Apple'
  - `device.model` (str): Device model. such as 'iPhone'
- `user_language` (str): Language used by the user's operating system. (e.g., `'zh-CN'`)
- `server_host` (str): PyWebIO server host, including domain and port, the port can be omitted when 80.
- `origin` (str): Indicate where the user from. Including protocol, host, and port parts. Such as `'http://localhost:8080'`. It may be empty, but it is guaranteed to have a value when the user's page address is not under the server host. (that is, the host, port part are inconsistent with `server_host`).
- `user_ip` (str): User's ip address.
- `backend` (str): The current PyWebIO backend server implementation. The possible values are `'tornado'`, `'flask'`, `'django'`, `'aiohttp'`, `'starlette'`.
- `protocol` (str): The communication protocol between PyWebIO server and browser. The possible values are `'websocket'`, `'http'`
- `request` (object): The request object when creating the current session. Depending on the backend server, the type of `request` can be:
  - When using Tornado, `request` is instance of tornado.httputil.HTTPServerRequest
  - When using Flask, `request` is instance of flask.Request
  - When using Django, `request` is instance of django.http.HttpRequest
  - When using aiohttp, `request` is instance of aiohttp.web.BaseRequest
  - When using FastAPI/Starlette, `request` is instance of starlette.websockets.WebSocket

The `user_agent` attribute of the session information object is parsed by the user-agents library. See https://github.com/selwin/python-user-agents#usage

Changed in version 1.2: Added the `protocol` attribute.

Example:

```
import json
from pywebio.session import info as session_info

put_code(json.dumps({
    k: str(getattr(session_info, k))
    for k in ['user_agent', 'user_language', 'server_host',
              'origin', 'user_ip', 'backend', 'protocol', 'request']
}, indent=4), 'json')
```

class pywebio.session.coroutinebased.**TaskHandler**(*close*, *closed*)

The handler of coroutine task

See also: *run_async()*

close()

Close the coroutine task.

closed() → bool

Returns a bool stating whether the coroutine task is closed.

pywebio.session.**hold**()

Keep the session alive until the browser page is closed by user.

> **Attention:** Since PyWebIO v1.4, in *server mode*, it's no need to call this function manually, PyWebIO will automatically hold the session for you when needed. The only case to use it is to prevent the application from exiting in script mode.

In case you use the previous version of PyWebIO (we strongly recommend that you upgrade to the latest version), here is the old document for `hold()`:

> After the PyWebIO session closed, the functions that need communicate with the PyWebIO server (such as the event callback of `put_buttons()` and download link of `put_file()`) will not work. You can call the `hold()` function at the end of the task function to hold the session, so that the event callback and download link will always be available before the browser page is closed by user.

pywebio.session.**run_async**(*coro_obj*)

Run the coroutine object asynchronously. PyWebIO interactive functions are also available in the coroutine.

`run_async()` can only be used in *coroutine-based session*.

Parameters **coro_obj** – Coroutine object

Returns `TaskHandle` instance, which can be used to query the running status of the coroutine or close the coroutine.

See also: *Concurrency in coroutine-based sessions*

pywebio.session.**run_asyncio_coroutine**(*coro_obj*)

If the thread running sessions are not the same as the thread running the asyncio event loop, you need to wrap `run_asyncio_coroutine()` to run the coroutine in asyncio.

Can only be used in *coroutine-based session*.

Parameters **coro_obj** – Coroutine object in `asyncio`

Example:

```
async def app():
    put_text('hello')
    await run_asyncio_coroutine(asyncio.sleep(1))
    put_text('world')

pywebio.platform.flask.start_server(app)
```

4.5 pywebio.platform — Deploy applications

The platform module provides support for deploying PyWebIO applications in different ways.

See also:

- *Use Guide: Server mode and Script mode*
- *Advanced Topic: Integration with Web Framework*

4.5.1 Directory Deploy

You can use path_deploy() or path_deploy_http() to deploy the PyWebIO applications from a directory. The python file under this directory need contain the main function to be seen as the PyWebIO application. You can access the application by using the file path as the URL.

Note that users can't view and access files or folders whose name begin with the underscore in this directory.

For example, given the following folder structure:

```
.
├── A
│   └── a.py
├── B
│   └── b.py
└── c.py
```

All three python files contain main PyWebIO application function.

If you use this directory in *path_deploy()*, you can access the PyWebIO application in `b.py` by using URL `http://<host>:<port>/A/b`. And if the files have been modified after run *path_deploy()*, you can use `reload` URL parameter to reload application in the file: `http://<host>:<port>/A/b?reload`

You can also use the command `pywebio-path-deploy` to start a server just like using *path_deploy()*. For more information, refer `pywebio-path-deploy --help`

pywebio.platform.**path_deploy**(*base*, *port=0*, *host=''*, *index=True*, *static_dir=None*, *reconnect_timeout=0*, *cdn=True*, *debug=False*, *allowed_origins=None*, *check_origin=None*, *max_payload_size='200M'*, ***tornado_app_settings*)

Deploy the PyWebIO applications from a directory.

The server communicates with the browser using WebSocket protocol.

Parameters

- **base** (*str*) – Base directory to load PyWebIO application.
- **port** (*int*) – The port the server listens on.
- **host** (*str*) – The host the server listens on.
- **index** (*bool/callable*) – Whether to provide a default index page when request a directory, default is `True`. `index` also accepts a function to custom index page, which receives the requested directory path as parameter and return HTML content in string.

  You can override the index page by add a `index.py` PyWebIO app file to the directory.
- **static_dir** (*str*) – Directory to store the application static files. The files in this directory can be accessed via `http://<host>:<port>/static/files`. For example, if there is a `A/B.jpg` file in `static_dir` path, it can be accessed via `http://<host>:<port>/static/A/B.jpg`.
- **reconnect_timeout** (*int*) – The client can reconnect to server within `reconnect_timeout` seconds after an unexpected disconnection. If set to 0 (default), once the client disconnects, the server session will be closed.

The rest arguments of `path_deploy()` have the same meaning as for *pywebio.platform.tornado.start_server()*

pywebio.platform.**path_deploy_http**(*base*, *port=0*, *host=''*, *index=True*, *static_dir=None*, *cdn=True*, *debug=False*, *allowed_origins=None*, *check_origin=None*, *session_expire_seconds=None*, *session_cleanup_interval=None*, *max_payload_size='200M'*, ***tornado_app_settings*)

Deploy the PyWebIO applications from a directory.

The server communicates with the browser using HTTP protocol.

The `base`, `port`, `host`, `index`, `static_dir` arguments of `path_deploy_http()` have the same meaning as for *pywebio.platform.path_deploy()*

The rest arguments of `path_deploy_http()` have the same meaning as for *pywebio.platform.tornado_http.start_server()*

4.5.2 Application Deploy

The `start_server()` functions can start a Python Web server and serve given PyWebIO applications on it.

The `webio_handler()` and `webio_view()` functions can be used to integrate PyWebIO applications into existing Python Web project.

The `wsgi_app()` and `asgi_app()` is used to get the WSGI or ASGI app for running PyWebIO applications. This is helpful when you don't want to start server with the Web framework built-in's. For example, you want to use other WSGI server, or you are deploying app in a cloud environment. Note that only Flask, Django and FastApi backend support it.

Changed in version 1.1: Added the `cdn` parameter in `start_server()`, `webio_handler()` and `webio_view()`.

Changed in version 1.2: Added the `static_dir` parameter in `start_server()`.

Changed in version 1.3: Added the `wsgi_app()` and `asgi_app()`.

Tornado support

There are two protocols (WebSocket and HTTP) can be used to communicates with the browser:

WebSocket

pywebio.platform.tornado.**start_server**(*applications: Union[Callable[], None], List[Callable[], None]], Dict[str, Callable[], None]]]*, *port: int = 0*, *host: str = ''*, *debug: bool = False*, *cdn: Union[bool, str] = True*, *static_dir: Optional[str] = None*, *remote_access: bool = False*, *reconnect_timeout: int = 0*, *allowed_origins: Optional[List[str]] = None*, *check_origin: Optional[Callable[[str], bool]] = None*, *auto_open_webbrowser: bool = False*, *max_payload_size: Union[int, str] = '200M'*, ***tornado_app_settings*)

Start a Tornado server to provide the PyWebIO application as a web service.

The Tornado server communicates with the browser by WebSocket protocol.

Tornado is the default backend server for PyWebIO applications, and `start_server` can be imported directly using `from pywebio import start_server`.

Parameters

- **applications** (*list/dict/callable*) – PyWebIO application. Can be a task function, a list of functions, or a dictionary. Refer to *Advanced topic: Multiple applications in start_server()* for more information.

  When the task function is a coroutine function, use *Coroutine-based session* implementation, otherwise, use thread-based session implementation.

- **port** (*int*) – The port the server listens on. When set to `0`, the server will automatically select a available port.

- **host** (*str*) – The host the server listens on. `host` may be either an IP address or hostname. If it's a hostname, the server will listen on all IP addresses associated with the name. `host` may be an empty string or None to listen on all available interfaces.

- **debug** (*bool*) – Tornado Server's debug mode. If enabled, the server will automatically reload for code changes. See tornado doc for more detail.
- **cdn** (*bool/str*) – Whether to load front-end static resources from CDN, the default is `True`. Can also use a string to directly set the url of PyWebIO static resources.
- **static_dir** (*str*) – The directory to store the application static files. The files in this directory can be accessed via `http://<host>:<port>/static/files`. For example, if there is a `A/B.jpg` file in `static_dir` path, it can be accessed via `http://<host>:<port>/static/A/B.jpg`.
- **remote_access** (*bool*) – Whether to enable remote access, when enabled, you can get a temporary public network access address for the current application, others can access your application via this address.
- **auto_open_webbrowser** (*bool*) – Whether or not auto open web browser when server is started (if the operating system allows it) .
- **reconnect_timeout** (*int*) – The client can reconnect to server within `reconnect_timeout` seconds after an unexpected disconnection. If set to 0 (default), once the client disconnects, the server session will be closed.
- **allowed_origins** (*list*) – The allowed request source list. (The current server host is always allowed) The source contains the protocol, domain name, and port part. Can use Unix shell-style wildcards:
  - `*` matches everything
  - `?` matches any single character
  - `[seq]` matches any character in *seq*
  - `[!seq]` matches any character not in *seq*

  Such as: `https://*.example.com *://*.example.com`

  For detail, see Python Doc
- **check_origin** (*callable*) – The validation function for request source. It receives the source string (which contains protocol, host, and port parts) as parameter and return `True/False` to indicate that the server accepts/rejects the request. If `check_origin` is set, the `allowed_origins` parameter will be ignored.
- **auto_open_webbrowser** – Whether or not auto open web browser when server is started (if the operating system allows it) .
- **max_payload_size** (*int/str*) – Max size of a websocket message which Tornado can accept. Messages larger than the `max_payload_size` (default 200MB) will not be accepted. `max_payload_size` can be a integer indicating the number of bytes, or a string ending with `K` / `M` / `G` (representing kilobytes, megabytes, and gigabytes, respectively). E.g: `500`, `'40K'`, `'3M'`
- **tornado_app_settings** – Additional keyword arguments passed to the constructor of `tornado.web.Application`. For details, please refer: https://www.tornadoweb.org/en/stable/web.html#tornado.web.Application.settings

`pywebio.platform.tornado.`**webio_handler**(*applications*, *cdn=True*, *reconnect_timeout=0*, *allowed_origins=None*, *check_origin=None*)

Get the `RequestHandler` class for running PyWebIO applications in Tornado. The `RequestHandler` communicates with the browser by WebSocket protocol.

The arguments of `webio_handler()` have the same meaning as for *pywebio.platform.tornado.start_server()*

HTTP

pywebio.platform.tornado_http.**start_server**(*applications*, *port=8080*, *host=''*, *debug=False*, *cdn=True*, *static_dir=None*, *allowed_origins=None*, *check_origin=None*, *auto_open_webbrowser=False*, *session_expire_seconds=None*, *session_cleanup_interval=None*, *max_payload_size='200M'*, ***tornado_app_settings*)

Start a Tornado server to provide the PyWebIO application as a web service.

The Tornado server communicates with the browser by HTTP protocol.

Parameters

- **session_expire_seconds** (*int*) – Session expiration time, in seconds(default 60s). If no client message is received within `session_expire_seconds`, the session will be considered expired.
- **session_cleanup_interval** (*int*) – Session cleanup interval, in seconds(default 120s). The server will periodically clean up expired sessions and release the resources occupied by the sessions.
- **max_payload_size** (*int/str*) – Max size of a request body which Tornado can accept.

The rest arguments of `start_server()` have the same meaning as for *pywebio.platform.tornado.start_server()*

New in version 1.2.

pywebio.platform.tornado_http.**webio_handler**(*applications*, *cdn=True*, *session_expire_seconds=None*, *session_cleanup_interval=None*, *allowed_origins=None*, *check_origin=None*)

Get the `RequestHandler` class for running PyWebIO applications in Tornado. The `RequestHandler` communicates with the browser by HTTP protocol.

The arguments of `webio_handler()` have the same meaning as for *pywebio.platform.tornado_http.start_server()*

New in version 1.2.

Flask support

When using the Flask as PyWebIO backend server, you need to install Flask by yourself and make sure the version is not less than `0.10`. You can install it with the following command:

```
pip3 install -U flask>=0.10
```

pywebio.platform.flask.**webio_view**(*applications*, *cdn=True*, *session_expire_seconds=None*, *session_cleanup_interval=None*, *allowed_origins=None*, *check_origin=None*)

Get the view function for running PyWebIO applications in Flask. The view communicates with the browser by HTTP protocol.

The arguments of `webio_view()` have the same meaning as for *pywebio.platform.flask.start_server()*

pywebio.platform.flask.**wsgi_app**(*applications*, *cdn=True*, *static_dir=None*, *allowed_origins=None*, *check_origin=None*, *session_expire_seconds=None*, *session_cleanup_interval=None*, *max_payload_size='200M'*)

Get the Flask WSGI app for running PyWebIO applications.

The arguments of `wsgi_app()` have the same meaning as for *pywebio.platform.flask.start_server()*

pywebio.platform.flask.**start_server**(*applications*, *port=8080*, *host=''*, *cdn=True*, *static_dir=None*, *remote_access=False*, *allowed_origins=None*, *check_origin=None*, *session_expire_seconds=None*, *session_cleanup_interval=None*, *debug=False*, *max_payload_size='200M'*, ***flask_options*)

Start a Flask server to provide the PyWebIO application as a web service.

Parameters

- **session_expire_seconds** (*int*) – Session expiration time, in seconds(default 600s). If no client message is received within `session_expire_seconds`, the session will be considered expired.
- **session_cleanup_interval** (*int*) – Session cleanup interval, in seconds(default 300s). The server will periodically clean up expired sessions and release the resources occupied by the sessions.
- **debug** (*bool*) – Flask debug mode. If enabled, the server will automatically reload for code changes.
- **max_payload_size** (*int/str*) – Max size of a request body which Flask can accept.
- **flask_options** – Additional keyword arguments passed to the `flask.Flask.run`. For details, please refer: https://flask.palletsprojects.com/en/1.1.x/api/#flask.Flask.run

The arguments of `start_server()` have the same meaning as for *pywebio.platform.tornado.start_server()*

Django support

When using the Django as PyWebIO backend server, you need to install Django by yourself and make sure the version is not less than `2.2`. You can install it with the following command:

```
pip3 install -U django>=2.2
```

pywebio.platform.django.**webio_view**(*applications*, *cdn=True*, *session_expire_seconds=None*, *session_cleanup_interval=None*, *allowed_origins=None*, *check_origin=None*)

Get the view function for running PyWebIO applications in Django. The view communicates with the browser by HTTP protocol.

The arguments of `webio_view()` have the same meaning as for *pywebio.platform.flask.webio_view()*

pywebio.platform.django.**wsgi_app**(*applications*, *cdn=True*, *static_dir=None*, *allowed_origins=None*, *check_origin=None*, *session_expire_seconds=None*, *session_cleanup_interval=None*, *debug=False*, *max_payload_size='200M'*, ***django_options*)

Get the Django WSGI app for running PyWebIO applications.

The arguments of `wsgi_app()` have the same meaning as for *pywebio.platform.django.start_server()*

pywebio.platform.django.**start_server**(*applications*, *port=8080*, *host=''*, *cdn=True*, *static_dir=None*, *remote_access=False*, *allowed_origins=None*, *check_origin=None*, *session_expire_seconds=None*, *session_cleanup_interval=None*, *debug=False*, *max_payload_size='200M'*, ***django_options*)

Start a Django server to provide the PyWebIO application as a web service.

Parameters

- **debug** (*bool*) – Django debug mode. See Django doc for more detail.
- **django_options** – Additional settings to django server. For details, please refer: https://docs.djangoproject.com/en/3.0/ref/settings/ . Among them, `DEBUG`, `ALLOWED_HOSTS`, `ROOT_URLCONF`, `SECRET_KEY` are set by PyWebIO and cannot be specified in `django_options`.

The rest arguments of `start_server()` have the same meaning as for *pywebio.platform.flask.start_server()*

aiohttp support

When using the aiohttp as PyWebIO backend server, you need to install aiohttp by yourself and make sure the version is not less than `3.1`. You can install it with the following command:

```
pip3 install -U aiohttp>=3.1
```

pywebio.platform.aiohttp.**webio_handler**(*applications*, *cdn=True*, *reconnect_timeout=0*, *allowed_origins=None*, *check_origin=None*, *max_payload_size='200M'*, *websocket_settings=None*)

Get the Request Handler coroutine for running PyWebIO applications in aiohttp. The handler communicates with the browser by WebSocket protocol.

The arguments of `webio_handler()` have the same meaning as for *pywebio.platform.aiohttp.start_server()*

Returns aiohttp Request Handler

pywebio.platform.aiohttp.**start_server**(*applications*, *port=0*, *host=''*, *debug=False*, *cdn=True*, *static_dir=None*, *remote_access=False*, *reconnect_timeout=0*, *allowed_origins=None*, *check_origin=None*, *auto_open_webbrowser=False*, *max_payload_size='200M'*, *websocket_settings=None*, ***aiohttp_settings*)

Start a aiohttp server to provide the PyWebIO application as a web service.

Parameters

- **websocket_settings** (*dict*) – The parameters passed to the constructor of `aiohttp.web.WebSocketResponse`. For details, please refer: https://docs.aiohttp.org/en/stable/web_reference.html#websocketresponse
- **aiohttp_settings** – Additional keyword arguments passed to the constructor of `aiohttp.web.Application`. For details, please refer: https://docs.aiohttp.org/en/stable/web_reference.html#application

The rest arguments of `start_server()` have the same meaning as for *pywebio.platform.tornado.start_server()*

FastAPI/Starlette support

When using the FastAPI/Starlette as PyWebIO backend server, you need to install `fastapi` or `starlette` by yourself. Also other dependency packages are required. You can install them with the following command:

```
pip3 install -U fastapi starlette uvicorn aiofiles websockets
```

pywebio.platform.fastapi.**webio_routes**(*applications*, *cdn=True*, *reconnect_timeout=0*, *allowed_origins=None*, *check_origin=None*)

Get the FastAPI/Starlette routes for running PyWebIO applications.

The API communicates with the browser using WebSocket protocol.

The arguments of `webio_routes()` have the same meaning as for *pywebio.platform.fastapi.start_server()*

New in version 1.3.

Returns FastAPI/Starlette routes

pywebio.platform.fastapi.**asgi_app**(*applications*, *cdn=True*, *reconnect_timeout=0*, *static_dir=None*, *debug=False*, *allowed_origins=None*, *check_origin=None*)

Get the starlette/Fastapi ASGI app for running PyWebIO applications.

Use *pywebio.platform.fastapi.webio_routes()* if you prefer handling static files yourself.

The arguments of `asgi_app()` have the same meaning as for *pywebio.platform.fastapi.start_server()*

Example

To be used with `FastAPI.mount()` to include pywebio as a subapp into an existing Starlette/FastAPI application:

```
from fastapi import FastAPI
from pywebio.platform.fastapi import asgi_app
from pywebio.output import put_text
app = FastAPI()
subapp = asgi_app(lambda: put_text("hello from pywebio"))
app.mount("/pywebio", subapp)
```

Returns Starlette/Fastapi ASGI app

New in version 1.3.

pywebio.platform.fastapi.**start_server**(*applications*, *port=0*, *host=''*, *cdn=True*, *reconnect_timeout=0*, *static_dir=None*, *remote_access=False*, *debug=False*, *allowed_origins=None*, *check_origin=None*, *auto_open_webbrowser=False*, *max_payload_size='200M'*, ***uvicorn_settings*)

Start a FastAPI/Starlette server using uvicorn to provide the PyWebIO application as a web service.

Parameters

- **debug** (*bool*) – Boolean indicating if debug tracebacks should be returned on errors.

- **uvicorn_settings** – Additional keyword arguments passed to `uvicorn.run()`. For details, please refer: https://www.uvicorn.org/settings/

The rest arguments of `start_server()` have the same meaning as for *pywebio.platform.tornado.start_server()*

New in version 1.3.

4.5.3 Other

pywebio.**config**(*, *title=None*, *description=None*, *theme=None*, *js_code=None*, *js_file=[]*, *css_style=None*, *css_file=[]*, *manifest=True*)

PyWebIO application configuration

Parameters

- **title** (*str*) – Application title
- **description** (*str*) – Application description
- **theme** (*str*) – Application theme. Available themes are: `dark`, `sketchy`, `minty`, `yeti`. You can also use environment variable `PYWEBIO_THEME` to specify the theme (with high priority).

  Theme preview demo

  The dark theme is modified from ForEvolve's bootstrap-dark. The sketchy, minty and yeti theme are from bootswatch.
- **js_code** (*str*) – The javascript code that you want to inject to page.
- **js_file** (*str/list*) – The javascript files that inject to page, can be a URL in str or a list of it.
- **css_style** (*str*) – The CSS style that you want to inject to page.
- **css_file** (*str/list*) – The CSS files that inject to page, can be a URL in str or a list of it.
- **manifest** (*bool/dict*) – Web application manifest configuration. This feature allows you to add a shortcut to the home screen of your mobile device, and launch the app like a native app. If set to `True`, the default manifest will be used. You can also specify the manifest content in dict. If `False`, the manifest will be disabled.

  Currently, the icons field of the manifest is not supported. Instead, you can use the `icon` field to specify the icon url.

`config()` can be used in 2 ways: direct call and decorator. If you call `config()` directly, the configuration will be global. If you use `config()` as decorator, the configuration will only work on single PyWebIO application function.

```
config(title="My application")  # global configuration

@config(css_style="* { color:red }")  # only works on this application
def app():
    put_text("hello PyWebIO")
```

Note: The configuration will affect all sessions

`title` and `description` are used for SEO, which are provided when indexed by search engines. If no `title` and `description` set for a PyWebIO application function, the docstring of the function will be used as title and description by default:

```
def app():
    """Application title

    Application description...
    (A empty line is used to separate the description and title)
    """
    pass
```

The above code is equal to:

```
@config(title="Application title", description="Application description...")
def app():
    pass
```

New in version 1.4.

Changed in version 1.5: add `theme` parameter

`pywebio.platform.`**`run_event_loop`**(*debug=False*)

run asyncio event loop

See also: *Integration coroutine-based session with Web framework*

Parameters **`debug`** – Set the debug mode of the event loop. See also: https://docs.python.org/3/library/asyncio-dev.html#asyncio-debug-mode

4.6 `pywebio.pin` — Persistent input

pin == Persistent input == Pinning input widget to the page

4.6.1 Overview

As you already know, the input function of PyWebIO is blocking and the input form will be destroyed after successful submission. In most cases, it enough to use this way to get input. However in some cases, you may want to make the input form **not** disappear after submission, and can continue to receive input.

So PyWebIO provides the `pin` module to achieve persistent input by pinning input widgets to the page.

The `pin` module achieves persistent input in 3 parts:

First, this module provides some pin widgets. Pin widgets are not different from output widgets in `pywebio.output` module, besides that they can also receive input.

This code outputs an text input pin widget:

```
put_input('input', label='This is a input widget')
```

In fact, the usage of pin widget function is same as the output function. You can use it as part of the combined output, or you can output pin widget to a scope:

```
put_row([
    put_input('input'),
    put_select('select', options=['A', 'B', 'C'])
])

with use_scope('search-area'):
    put_input('search', placeholder='Search')
```

Then, you can use the *pin* object to get the value of pin widget:

```
put_input('pin_name')
put_buttons(['Get Pin Value'], lambda _: put_text(pin.pin_name))
```

The first parameter that the pin widget function receives is the name of the pin widget. You can get the current value of the pin widget via the attribute of the same name of the *pin* object.

In addition, the *pin* object also supports getting the value of the pin widget by index, that is to say:

```
pin['pin_name'] == pin.pin_name
```

There are also two useful functions when you use the pin module: *pin_wait_change()* and *pin_update()*.

Since the pin widget functions is not blocking, *pin_wait_change()* is used to wait for the value of one of a list of pin widget to change, it 's a blocking function.

pin_update() can be used to update attributes of pin widgets.

4.6.2 Pin widgets

Each pin widget function corresponds to an input function of *input* module.

The function of pin widget supports most of the parameters of the corresponding input function. Here lists the difference between the two in parameters:

- The first parameter of pin widget function is always the name of the widget, and if you output two pin widgets with the same name, the previous one will expire.
- Pin functions don't support the `on_change` and `validate` callbacks, and the `required` parameter. (There is a *pin_on_change()* function as an alternative to `on_change`)
- Pin functions have additional `scope` and `position` parameters for output control.

pywebio.pin.**put_input**(*name: str*, *type: str = 'text'*, ***, *label: str = ''*, *value: Optional[str] = None*, *placeholder: Optional[str] = None*, *readonly: Optional[bool] = None*, *datalist: Optional[List[str]] = None*, *help_text: Optional[str] = None*, *scope: Optional[str] = None*, *position: int = - 1*) → pywebio.io_ctrl.Output

Output an input widget. Refer to: *pywebio.input.input()*

pywebio.pin.**put_textarea**(*name: str*, ***, *label: str = ''*, *rows: int = 6*, *code: Optional[Union[bool, Dict]] = None*, *maxlength: Optional[int] = None*, *minlength: Optional[int] = None*, *value: Optional[str] = None*, *placeholder: Optional[str] = None*, *readonly: Optional[bool] = None*, *help_text: Optional[str] = None*, *scope: Optional[str] = None*, *position: int = - 1*) → pywebio.io_ctrl.Output

Output a textarea widget. Refer to: *pywebio.input.textarea()*

pywebio.pin.**put_select**(*name: str*, *options: Optional[List[Union[Dict[str, Any], Tuple, List, str]]] = None*, ***, *label: str = ''*, *multiple: Optional[bool] = None*, *value: Optional[Union[List, str]] = None*, *native: Optional[bool] = None*, *help_text: Optional[str] = None*, *scope: Optional[str] = None*, *position: int = - 1*) → pywebio.io_ctrl.Output

Output a select widget. Refer to: *pywebio.input.select()*

Note: Unlike *pywebio.input.select()*, when `multiple=True` and the user is using PC/macOS, *put_select()* will use bootstrap-select by default. Setting `native=True` will force PyWebIO to use native select component on all platforms and vice versa.

---

pywebio.pin.**put_checkbox**(*name: str*, *options: Optional[List[Union[Dict[str, Any], Tuple, List, str]]] = None*, ***, *label: str = ''*, *inline: Optional[bool] = None*, *value: Optional[List] = None*, *help_text: Optional[str] = None*, *scope: Optional[str] = None*, *position: int = - 1*) → pywebio.io_ctrl.Output

Output a checkbox widget. Refer to: *pywebio.input.checkbox()*

pywebio.pin.**put_radio**(*name: str*, *options: Optional[List[Union[Dict[str, Any], Tuple, List, str]]] = None*, ***, *label: str = ''*, *inline: Optional[bool] = None*, *value: Optional[str] = None*, *help_text: Optional[str] = None*, *scope: Optional[str] = None*, *position: int = - 1*) → pywebio.io_ctrl.Output

Output a radio widget. Refer to: *pywebio.input.radio()*

pywebio.pin.**put_slider**(*name: str*, ***, *label: str = ''*, *value: Union[int, float] = 0*, *min_value: Union[int, float] = 0*, *max_value: Union[int, float] = 100*, *step: int = 1*, *required: Optional[bool] = None*, *help_text: Optional[str] = None*, *scope: Optional[str] = None*, *position: int = - 1*) → pywebio.io_ctrl.Output

Output a slide widget. Refer to: *pywebio.input.slider()*

pywebio.pin.**put_actions**(*name: str*, ***, *label: str = ''*, *buttons: Optional[List[Union[Dict[str, Any], Tuple, List, str]]] = None*, *help_text: Optional[str] = None*, *scope: Optional[str] = None*, *position: int = - 1*) → pywebio.io_ctrl.Output

Output a group of action button. Refer to: *pywebio.input.actions()*

Unlike the `actions()`, `put_actions()` won't submit any form, it will only set the value of the pin widget. Only 'submit' type button is available in pin widget version.

New in version 1.4.

pywebio.pin.**put_file_upload**(*name: str*, ***, *label: str = ''*, *accept: Optional[Union[List, str]] = None*, *placeholder: str = 'Choose file'*, *multiple: bool = False*, *max_size: Union[int, str] = 0*, *max_total_size: Union[int, str] = 0*, *help_text: Optional[str] = None*, *scope: Optional[str] = None*, *position: int = - 1*) → pywebio.io_ctrl.Output

Output a file uploading widget. Refer to: *pywebio.input.file_upload()*

4.6.3 Pin utils

pywebio.pin.**pin**

Pin widgets value getter and setter.

You can use attribute or key index of `pin` object to get the current value of a pin widget. By default, when accessing the value of a widget that does not exist, it returns `None` instead of throwing an exception. You can enable the error raising by `pin.use_strict()` method.

You can also use the `pin` object to set the value of pin widget:

```
put_input('counter', type='number', value=0)

while True:
    pin.counter = pin.counter + 1  # Equivalent to: pin['counter'] = pin['counter
↪'] + 1
    time.sleep(1)
```

Note: When using *coroutine-based session*, you need to use the `await pin.name` (or `await pin['name']`) syntax to get pin widget value.

Use `pin.pin.use_strict()` to enable strict mode for getting pin widget value. An `AssertionError` will be raised when try to get value of pin widgets that are currently not in the page.

pywebio.pin.**pin_wait_change**(**names*, *timeout: Optional[int] = None*)

`pin_wait_change()` listens to a list of pin widgets, when the value of any widgets changes, the function returns with the name and value of the changed widget.

Parameters

- **names** (*str*) – List of names of pin widget
- **timeout** (*int/None*) – If `timeout` is a positive number, `pin_wait_change()` blocks at most `timeout` seconds and returns `None` if no changes to the widgets within that time. Set to `None` (the default) to disable timeout.

Return dict/None `{"name": name of the changed widget, "value": current value of the changed widget }` , when a timeout occurs, return `None`.

Example:

```
put_input('a', type='number', value=0)
put_input('b', type='number', value=0)

while True:
    changed = pin_wait_change('a', 'b')
    with use_scope('res', clear=True):
        put_code(changed)
        put_text("a + b = %s" % (pin.a + pin.b))
```

Here is an demo of using *pin_wait_change()* to make a markdown previewer.

Note that: updating value with the *pin* object or *pin_update()* does not trigger *pin_wait_change()* to return.

When using *coroutine-based session*, you need to use the `await pin_wait_change()` syntax to invoke this function.

pywebio.pin.**pin_update**(*name: str*, ***spec*)

Update attributes of pin widgets.

Parameters

- **name** (*str*) – The `name` of the target input widget.
- **spec** – The pin widget parameters need to be updated. Note that those parameters can not be updated: `type`, `name`, `code`, `multiple`

pywebio.pin.**pin_on_change**(*name: str*, *onchange: Optional[Callable[[Any], None]] = None*, *clear: bool = False*, *init_run: bool = False*, ***callback_options*)

Bind a callback function to pin widget, the function will be called when user change the value of the pin widget.

The `onchange` callback is invoked with one argument, the changed value of the pin widget. You can bind multiple functions to one pin widget, those functions will be invoked sequentially (default behavior, can be changed by `clear` parameter).

Parameters

- **name** (*str*) – pin widget name
- **onchange** (*callable*) – callback function
- **clear** (*bool*) – whether to clear the previous callbacks bound to this pin widget. If you just want to clear callbacks and not set new callback, use `pin_on_change(name, clear=True)`.
- **init_run** (*bool*) – whether to run the `onchange` callback once immediately before the pin widget changed. This parameter can be used to initialize the output.
- **callback_options** – Other options of the `onclick` callback. Refer to the `callback_options` parameter of *put_buttons()*

New in version 1.6.

4.7 Advanced topic

This section will introduce the advanced features of PyWebIO.

4.7.1 Start multiple applications with start_server()

start_server() accepts a function as PyWebIO application. In addition, *start_server()* also accepts a list of application function or a dictionary of it to start multiple applications. You can use *pywebio.session.go_app()* or *put_link()* to jump between application:

```
def task_1():
    put_text('task_1')
    put_buttons(['Go task 2'], [lambda: go_app('task_2')])

def task_2():
    put_text('task_2')
    put_buttons(['Go task 1'], [lambda: go_app('task_1')])

def index():
    put_link('Go task 1', app='task_1')  # Use `app` parameter to specify the task␣
↪name
    put_link('Go task 2', app='task_2')

# equal to `start_server({'index': index, 'task_1': task_1, 'task_2': task_2})`
start_server([index, task_1, task_2])
```

When the first parameter of *start_server()* is a dictionary, whose key is application name and value is application function. When it is a list, PyWebIO will use function name as application name.

You can select which application to access through the `app` URL parameter (for example, visit `http://host:port/?app=foo` to access the `foo` application), By default, the `index` application is opened when no `app` URL parameter provided. When the `index` application doesn't exist, PyWebIO will provide a default index application.

4.7.2 Integration with web framework

The PyWebIO application can be integrated into an existing Python Web project, the PyWebIO application and the Web project share a web framework. PyWebIO currently supports integration with Flask, Tornado, Django, aiohttp and FastAPI(Starlette) web frameworks.

The integration methods of those web frameworks are as follows:

Tornado

Tornado

Use *pywebio.platform.tornado.webio_handler()* to get the WebSocketHandler class for running PyWebIO applications in Tornado:

```
import tornado.ioloop
import tornado.web
from pywebio.platform.tornado import webio_handler

class MainHandler(tornado.web.RequestHandler):
    def get(self):
        self.write("Hello, world")

if __name__ == "__main__":
    application = tornado.web.Application([
        (r"/", MainHandler),
        (r"/tool", webio_handler(task_func)),  # `task_func` is PyWebIO task function
    ])
    application.listen(port=80, address='localhost')
    tornado.ioloop.IOLoop.current().start()
```

In above code, we add a routing rule to bind the `WebSocketHandler` of the PyWebIO application to the `/tool` path. After starting the Tornado server, you can visit `http://localhost/tool` to open the PyWebIO application.

Attention: PyWebIO uses the WebSocket protocol to communicate with the browser in Tornado. If your Tornado application is behind a reverse proxy (such as Nginx), you may need to configure the reverse proxy to support the WebSocket protocol. Here is an example of Nginx WebSocket configuration.

Flask

Flask

Use *pywebio.platform.flask.webio_view()* to get the view function for running PyWebIO applications in Flask:

```
from pywebio.platform.flask import webio_view
from flask import Flask

app = Flask(__name__)

# `task_func` is PyWebIO task function
app.add_url_rule('/tool', 'webio_view', webio_view(task_func),
            methods=['GET', 'POST', 'OPTIONS'])  # need GET,POST and OPTIONS methods

app.run(host='localhost', port=80)
```

In above code, we add a routing rule to bind the view function of the PyWebIO application to the `/tool` path. After starting the Flask application, visit `http://localhost/tool` to open the PyWebIO application.

Django

Django

Use *pywebio.platform.django.webio_view()* to get the view function for running PyWebIO applications in Django:

```
# urls.py

from django.urls import path
from pywebio.platform.django import webio_view

# `task_func` is PyWebIO task function
webio_view_func = webio_view(task_func)

urlpatterns = [
    path(r"tool", webio_view_func),
]
```

In above code, we add a routing rule to bind the view function of the PyWebIO application to the `/tool` path. After starting the Django server, visit `http://localhost/tool` to open the PyWebIO application

aiohttp

aiohttp

Use *pywebio.platform.aiohttp.webio_handler()* to get the Request Handler coroutine for running PyWebIO applications in aiohttp:

```
from aiohttp import web
from pywebio.platform.aiohttp import webio_handler

app = web.Application()
# `task_func` is PyWebIO task function
app.add_routes([web.get('/tool', webio_handler(task_func))])

web.run_app(app, host='localhost', port=80)
```

After starting the aiohttp server, visit `http://localhost/tool` to open the PyWebIO application

Attention: PyWebIO uses the WebSocket protocol to communicate with the browser in aiohttp. If your aiohttp server is behind a reverse proxy (such as Nginx), you may need to configure the reverse proxy to support the WebSocket protocol. Here is an example of Nginx WebSocket configuration.

FastAPI/Starlette

FastAPI/Starlette

Use *pywebio.platform.fastapi.webio_routes()* to get the FastAPI/Starlette routes for running PyWebIO applications. You can mount the routes to your FastAPI/Starlette app.

FastAPI:

```
from fastapi import FastAPI
from pywebio.platform.fastapi import webio_routes

app = FastAPI()
```

(continues on next page)

(continued from previous page)

```
@app.get("/app")
def read_main():
    return {"message": "Hello World from main app"}

# `task_func` is PyWebIO task function
app.mount("/tool", FastAPI(routes=webio_routes(task_func)))
```

Starlette:

```
from starlette.applications import Starlette
from starlette.responses import JSONResponse
from starlette.routing import Route, Mount
from pywebio.platform.fastapi import webio_routes

async def homepage(request):
    return JSONResponse({'hello': 'world'})

app = Starlette(routes=[
    Route('/', homepage),
    Mount('/tool', routes=webio_routes(task_func))  # `task_func` is PyWebIO task␣
↪function
])
```

After starting the server by using `uvicorn <module>:app`, visit `http://localhost:8000/tool/` to open the PyWebIO application

See also: FastAPI doc , Starlette doc

Attention: PyWebIO uses the WebSocket protocol to communicate with the browser in FastAPI/Starlette. If your server is behind a reverse proxy (such as Nginx), you may need to configure the reverse proxy to support the WebSocket protocol. Here is an example of Nginx WebSocket configuration.

Notes

Deployment in production

In your production system, you may want to deploy the web applications with some WSGI/ASGI servers such as uWSGI, Gunicorn, and Uvicorn. Since PyWebIO applications store session state in memory of process, when you use HTTP-based sessions (Flask and Django) and spawn multiple workers to handle requests, the request may be dispatched to a process that does not hold the session to which the request belongs. So you can only start one worker to handle requests when using Flask or Django backend.

If you still want to use multiple processes to increase concurrency, one way is to use Uvicorn+FastAPI, or you can also start multiple Tornado/aiohttp processes and add external load balancer (such as HAProxy or nginx) before them. Those backends use the WebSocket protocol to communicate with the browser in PyWebIO, so there is no the issue as described above.

Static resources Hosting

By default, the front-end of PyWebIO gets required static resources from CDN. If you want to deploy PyWebIO applications in an offline environment, you need to host static files by yourself, and set the `cdn` parameter of `webio_view()` or `webio_handler()` to `False`.

When setting `cdn=False` , you need to host the static resources in the same directory as the PyWebIO application. In addition, you can also pass a string to `cdn` parameter to directly set the URL of PyWebIO static resources directory.

The path of the static file of PyWebIO is stored in `pywebio.STATIC_PATH`, you can use the command `python3 -c "import pywebio; print(pywebio.STATIC_PATH)"` to print it out.

Note: `start_server()` and `path_deploy()` also support `cdn` parameter, if it is set to `False`, the static resource will be hosted in local server automatically, without manual hosting.

4.7.3 Coroutine-based session

In most cases, you don't need the coroutine-based session. All functions or methods in PyWebIO that are only used for coroutine sessions are specifically noted in the document.

PyWebIO's session is based on thread by default. Each time a user opens a session connection to the server, PyWebIO will start a thread to run the task function. In addition to thread-based sessions, PyWebIO also provides coroutine-based sessions. Coroutine-based sessions accept coroutine functions as task functions.

The session based on the coroutine is a single-thread model, which means that all sessions run in a single thread. For IO-bound tasks, coroutines take up fewer resources than threads and have performance comparable to threads. In addition, the context switching of the coroutine is predictable, which can reduce the need for program synchronization and locking, and can effectively avoid most critical section problems.

Using coroutine session

To use coroutine-based session, you need to use the `async` keyword to declare the task function as a coroutine function, and use the `await` syntax to call the PyWebIO input function:

```
from pywebio.input import *
from pywebio.output import *
from pywebio import start_server

async def say_hello():
    name = await input("what's your name?")
    put_text('Hello, %s' % name)

start_server(say_hello, auto_open_webbrowser=True)
```

In the coroutine task function, you can also use `await` to call other coroutines or (awaitable objects) in the standard library asyncio:

```
import asyncio
from pywebio import start_server

async def hello_word():
    put_text('Hello ...')
    await asyncio.sleep(1)  # await awaitable objects in asyncio
    put_text('... World!')

async def main():
    await hello_word()  # await coroutine
    put_text('Bye, bye')

start_server(main, auto_open_webbrowser=True)
```

Attention: In coroutine-based session, all input functions defined in the *pywebio.input* module need to use `await` syntax to get the return value. Forgetting to use `await` will be a common error when using coroutine-based session.

Other functions that need to use `await` syntax in the coroutine session are:

- *pywebio.session.run_asyncio_coroutine(coro_obj)*
- *pywebio.session.eval_js(expression)*

Warning: Although the PyWebIO coroutine session is compatible with the `awaitable objects` in the standard library `asyncio`, the `asyncio` library is not compatible with the `awaitable objects` in the PyWebIO coroutine session.

That is to say, you can't pass PyWebIO `awaitable objects` to the `asyncio` functions that accept `awaitable objects`. For example, the following calls are **not supported**

```
await asyncio.shield(pywebio.input())
await asyncio.gather(asyncio.sleep(1), pywebio.session.eval_js('1+1'))
task = asyncio.create_task(pywebio.input())
```

Concurrency in coroutine-based sessions

In coroutine-based session, you can start new thread, but you cannot call PyWebIO interactive functions in it (*register_thread()* is not available in coroutine session). But you can use *run_async(coro)* to execute a coroutine object asynchronously, and PyWebIO interactive functions can be used in the new coroutine:

```
from pywebio import start_server
from pywebio.session import run_async

async def counter(n):
    for i in range(n):
        put_text(i)
        await asyncio.sleep(1)

async def main():
    run_async(counter(10))
    put_text('Main coroutine function exited.')


start_server(main, auto_open_webbrowser=True)
```

run_async(coro) returns a *TaskHandler*, which can be used to query the running status of the coroutine or close the coroutine.

Close of session

Similar to thread-based session, when user close the browser page, the session will be closed.

After the browser page closed, PyWebIO input function calls that have not yet returned in the current session will cause `SessionClosedException`, and subsequent calls to PyWebIO interactive functions will cause `SessionNotFoundException` or `SessionClosedException`.

`defer_call(func)` also available in coroutine session.

Integration with Web Framework

The PyWebIO application that using coroutine-based session can also be integrated to the web framework.

However, there are some limitations when using coroutine-based sessions to integrate into Flask or Django:

First, when `await` the coroutine objects/awaitable objects in the `asyncio` module, you need to use *`run_asyncio_coroutine()`* to wrap the coroutine object.

Secondly, you need to start a new thread to run the event loop before starting a Flask/Django server.

Example of coroutine-based session integration into Flask:

```
import asyncio
import threading
from flask import Flask, send_from_directory
from pywebio import STATIC_PATH
from pywebio.output import *
from pywebio.platform.flask import webio_view
from pywebio.platform import run_event_loop
from pywebio.session import run_asyncio_coroutine

async def hello_word():
    put_text('Hello ...')
    await run_asyncio_coroutine(asyncio.sleep(1))  # can't just "await asyncio.
↪sleep(1)"
    put_text('... World!')

app = Flask(__name__)
app.add_url_rule('/hello', 'webio_view', webio_view(hello_word),
                 methods=['GET', 'POST', 'OPTIONS'])

# thread to run event loop
threading.Thread(target=run_event_loop, daemon=True).start()
app.run(host='localhost', port=80)
```

Finally, coroutine-based session is not available in the script mode. You always need to use `start_server()` to run coroutine task function or integrate it to a web framework.

4.8 Libraries support

4.8.1 Build stand-alone App

PyInstaller bundles a Python application and all its dependencies into a folder or executable. The user can run the packaged app without installing a Python interpreter or any modules.

You can use PyInstaller to packages PyWebIO application into a stand-alone executable or folder:

1. Create a pyinstaller spec (specification) file:

   ```
   pyi-makespec <options> app.py
   ```

   You need replace `app.py` to your PyWebIO application file name.

2. Only for PyWebIO before v1.8: Edit the spec file, change the `datas` parameter of `Analysis`:

   ```
   from pywebio.utils import pyinstaller_datas

   a = Analysis(
       ...
       datas=pyinstaller_datas(),
       ...
   ```

3. Build the application by passing the spec file to the pyinstaller command:

   ```
   pyinstaller app.spec
   ```

If you want to create a one-file bundled executable, you need pass `--onefile` option in first step.

For more information, please visit: https://pyinstaller.readthedocs.io/en/stable/spec-files.html

4.8.2 Data visualization

PyWebIO supports for data visualization with the third-party libraries.

Bokeh

Bokeh is an interactive visualization library for modern web browsers. It provides elegant, concise construction of versatile graphics, and affords high-performance interactivity over large or streaming datasets.

You can use `bokeh.io.output_notebook(notebook_type='pywebio')` in the PyWebIO session to setup Bokeh environment. Then you can use `bokeh.io.show()` to output a boken chart:

```
from bokeh.io import output_notebook
from bokeh.io import show

output_notebook(notebook_type='pywebio')
fig = figure(...)
...
show(fig)
```

See related demo on bokeh demo

In addition to creating ordinary charts, Bokeh can also build the Bokeh applications by starting the Bokeh server. The purpose of the Bokeh server is to make it easy for Python users to create interactive web applications that can connect front-end UI events to real, running Python code.

In PyWebIO, you can also use `bokeh.io.show()` to display a Bokeh App. For the example, see bokeh_app.py.

Note: Bokeh App currently is only available in the default Tornado backend

pyecharts

pyecharts is a python plotting library which uses Echarts as underlying implementation.

In PyWebIO, you can use the following code to output the pyecharts chart instance:

```
# `chart` is pyecharts chart instance
pywebio.output.put_html(chart.render_notebook())
```

See related demo on pyecharts demo

plotly

plotly.py is an interactive, open-source, and browser-based graphing library for Python.

In PyWebIO, you can use the following code to output the plotly chart instance:

```
# `fig` is plotly chart instance
html = fig.to_html(include_plotlyjs="require", full_html=False)
pywebio.output.put_html(html)
```

See related demo on plotly demo

pyg2plot

pyg2plot is a python plotting library which uses G2Plot as underlying implementation.

In PyWebIO, you can use the following code to output the pyg2plot chart instance:

```
# `chart` is pyg2plot chart instance
pywebio.output.put_html(chart.render_notebook())
```

See related demo on plotly demo

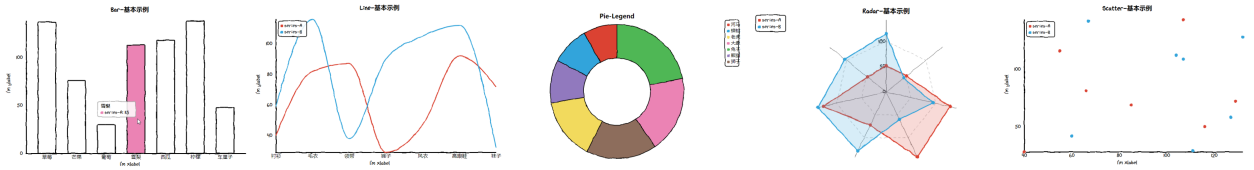
cutecharts.py

cutecharts.py is a hand drawing style charts library for Python which uses chart.xkcd as underlying implementation.

In PyWebIO, you can use the following code to output the cutecharts.py chart instance:

```
# `chart` is cutecharts chart instance
pywebio.output.put_html(chart.render_notebook())
```

See related demo on cutecharts demo



4.9 Cookbook

See also:

PyWebIO Battery

4.9.1 Interaction related

Equivalent to "Press any key to continue"

```
actions(buttons=["Continue"])
```

Output pandas dataframe

```
import numpy as np
import pandas as pd

df = pd.DataFrame(np.random.randn(6, 4), columns=list("ABCD"))
put_html(df.to_html(border=0))
```

See also:

pandas.DataFrame.to_html — pandas documentation

Output Matplotlib figure

Instead of using `matplotlib.pyplot.show()`, to show matplotlib figure in PyWebIO, you need to save the figure to in-memory buffer fist and then output the buffer via *`pywebio.output.put_image()`*:

```
import matplotlib
import matplotlib.pyplot as plt
import io
import pywebio

matplotlib.use('agg')  # required, use a non-interactive backend

fig, ax = plt.subplots()  # Create a figure containing a single axes.
```

(continues on next page)

(continued from previous page)

```
ax.plot([1, 2, 3, 4], [1, 4, 2, 3])  # Plot some data on the axes.

buf = io.BytesIO()
fig.savefig(buf)
pywebio.output.put_image(buf.getvalue())
```

The `matplotlib.use('agg')` is required so that the server does not try to create (and then destroy) GUI windows that will never be seen.

When using Matplotlib in a web server (multiple threads environment), pyplot may cause some conflicts in some cases, read the following articles for more information:

- Multi Threading in Python and Pyplot | by Ranjitha Korrapati | Medium
- Embedding in a web application server (Flask) — Matplotlib documentation

Add new syntax highlight for code output

When output code via `put_markdown()` or `put_code()`, PyWebIO provides syntax highlight for some common languages. If you find your code have no syntax highlight, you can add the syntax highlighter by two following steps:

1. Go to prismjs CDN page to get your syntax highlighter link.
2. Use *`config(js_file=...)`* to load the syntax highlight module

```
@config(js_file="https://cdn.jsdelivr.net/npm/prismjs@1.23.0/components/prism-diff.
↪min.js")
def main():
    put_code("""
+ AAA
- BBB
CCC
    """.strip(), language='diff')

    put_markdown("""
    ```diff
    + AAA
    - BBB
    CCC
    ```
    """, lstrip=True)
```

4.9.2 Web application related

Add Google AdSense/Analytics code

When you setup Google AdSense/Analytics, you will get a javascript file and a piece of code that needs to be inserted into your application page, you can use *`pywebio.config()`* to inject js file and code to your PyWebIO application:

```
from pywebio import start_server, output, config

js_file = "https://www.googletagmanager.com/gtag/js?id=G-xxxxxxx"
js_code = """
window.dataLayer = window.dataLayer || [];
function gtag(){dataLayer.push(arguments);}
```

(continues on next page)

(continued from previous page)

```
gtag('js', new Date());

gtag('config', 'G-xxxxxxx');
"""

@config(js_file=js_file, js_code=js_code)
def main():
    output.put_text("hello world")

start_server(main, port=8080)
```

Refresh page on connection lost

Add the following code to the beginning of your PyWebIO application main function:

```
session.run_js('WebIO._state.CurrentSession.on_session_close(()=>{setTimeout(()=>
↪location.reload(), 4000})')
```

4.10 Release notes

4.10.1 What's new in PyWebIO 1.8

2022/4/10

Highlights

- Add datatable widget (*put_datatable()*)
- Build reliable message transmission over HTTP-based backends (Flask and Django)

Backwards-incompatible changes

- When use *put_loading()* as context manager, the output inside the context will also been removed after the context block exits.

Detailed changes

- Add *put_file_upload()* pin widget.
- Add WPA support (via *config(manifest)*), so PyWebIO apps can be launched like a native app on mobile devices.
- Add type hints to all public functions (#501, thanks to)
- Add Uzbek language support for UI labels (#539, thanks to Ulugbek)
- Remove the `NullHandler()` logging handler added to `pywebio` logger, so the exception log from PyWebIO can be output by default.
- Add `max_payload_size` param to `start_server()` and `webio_handler()` for aiohttp and fastapi backends.

- When `tdata` of `put_table()` is list of dict, `header` parameter is not mandatory anymore.
- Add pyinstaller hook, so PyWebIO apps can be packaged to executable file with pyinstaller without any extra configuration.
- No traceback expose to user in production environment (`start_server(debug=False)`, the default setting).

Bug fix

- Fix memory leak after close session (#545)

4.10.2 What's new in PyWebIO 1.7

2022/10/17

Highlights

- add session reconnect to aiohttp and fastapi backends (now, all platforms support session reconnect)

Detailed changes

- auto use local static when CND is not available
- refine *`use_scope(clear=True)`* to avoid page flashing

Bug fix

- fix: *`textarea(code=True, required=True)`* can't submit
- fix: auto hold don't work on script mode
- fix (#389): `put_select()` was hidden by `put_tabs()`
- fix: `input_update(datalist)` don't work when `datalist` is not provided in `input()`
- fix (#459): code textarea onchange fired when set value
- fix (#453): `put_table()` error when table data is empty with rich header
- fix load old static resource after version upgrade
- fix cancel type raise error in single `action()`
- fix (#377): error on nested `onchange` callback
- fix (#468): can't reset `select()`
- fix `set_env(output_animation=False)` don't work for image

4.10.3 What's new in PyWebIO 1.6

2022/3/23

Highlights

- add *pywebio.pin.pin_on_change()*

Detailed changes

- use bootstrap-select to provide more user-friendly select input
- add `pin.pin.use_strict()` to enable strict mode for getting pin widget value
- Persian language support for default labels, thanks to Pikhosh
- add color input type (#310)
- add input check on number and float type input

Bug fix

- fix: uncaught `SessionClosedException` in callback of thread-based session
- fix(#313): slider value label don't sync when set value

v1.6.1 (2022/5/22)

- fix (#380): `put_processbar()` don't work when name contains space
- fix (#385): bootstrap-select issue
- fix (#389): `put_select()` was hidden by `put_tabs()`
- fix auto hold don't work on script mode
- provide a fallback way when CDN is not available

v1.6.2 (2022/7/16)

- fix: plotly.js version error due to outdated CDN link

4.10.4 What's new in PyWebIO 1.5

2021/11/20

Highlights

- theme support via *pywebio.config()*, demo
- deprecate `pywebio.output.output()`, use *pywebio.output.use_scope()* instead (`output()` still work)

Detailed changes

- enable `lstrip` by default in *`put_markdown()`*, and the behavior of `lstrip` is more clever than previous version. Deprecate `strip_indent` since `lstrip` is sufficient.
- button disabled state support in *`pywebio.output.put_buttons()`* and *`pywebio.output.put_button()`*, and button value can be any type
- buttons in *`pywebio.input.actions()`* support color setting
- russian language support for frontend labels and messages. Thanks to @Priler.
- improve default index page of *`pywebio.platform.path_deploy()`*: improve pywebio app detection and show app title.
- compatible with latest aiohttp(v3.8)
- enable `websocket_ping_interval` by default in tornado server to avoid idle connections being close in some cloud platform (like heroku)
- exception traceback will be show in page when enable debug
- *`slider`* input add indicator to show its current value

Bug fix

- deep copy `options` and `buttons` parameters to avoid potential error - 81d57ba4, cb5ac8d5 - e262ea43
- fix page width exceeding screen width (mostly on mobile devices) - 536d09e3
- fix `put_buttons()` issue when buttons have same value - cb5ac8d5
- fix layout issue when use `put_markdown()` - 364059ae
- fix style issue in `put_tabs()` widget - f056f1ac
- fix sibling import issue in `path_deploy()` - 35209a7e
- fix "Address already in use" error when enable remote access in some cases - 8dd9877d

v1.5.1 (2021/12/21)

- fix setitem error of `pin.pin` object - 3f5cf1e5
- fix thread-based session tot closed properly - 22fbbf86..3bc7d36b>
- fix OverflowError on 32-bit Windows - 4ac7f0e5
- fix a sample error from cookbook - 99593db4
- fix spawn 2 remote access processes when enable debug in flask backed - 073f8ace

v1.5.2 (2021/12/30)

- fix #243: thread keep alive after session closed
- fix #247: can't use coroutine callback in `put_button()`

4.10.5 What's new in PyWebIO 1.4

2021/10/4

Highlights

- automatically hold session when needed
- support for binding onclick callback on any output widget

Detailed changes

- migrate to a open-source remote access service
- add `output_max_width` parameter to *`set_env()`*
- can use `Esc`/`F11` to toggle fullscreen of codemirror textarea
- *`pin_wait_change()`* support `timeout` parameter
- add *`pywebio.config()`*
- add *`pywebio.output.put_button()`*
- add *`pywebio.pin.put_actions()`*
- rearrange document

Bug fix

- fix(#148): form can't be submit after validation failed - e262ea43
- fix some codemirror issues: codemirror refresh and mode auto load - b7957891, 50cc41a9
- fix: `run_js()` return `None` when empty-value - 89ce352d
- fix: whole output crash when a sub output fail - 31b26d09

4.10.6 What's new in PyWebIO 1.3

2021/6/12

Highlights

- New module *pin* to provide persistent input support.
- Add a remote access service to `start_server()`. See *server mode - User Guide* for detail.
- Add *`input_update()`*, add `onchange` callback in input functions.
- Add support for FastAPI and Starlette.

Detailed changes

- *input* module
  - Add *input_update()*, add `onchange` callback in input functions.
  - Add *pywebio.input.slider()* to get range input.
- *output* module
  - Mark *style()* as deprecated, see *style - User Guide* for new method.
  - Add *pywebio.output.put_tabs()* to output tabs.
  - *put_html()* adds compatibility with ipython rich output.
  - Add `group` and `outline` parameters in *put_buttons()*.
- *session* module
  - Add promise support in *eval_js()*.
  - Support config input panel via *set_env()*.
- *platform* module
  - Add support for FastAPI and Starlette.
  - Add `wsgi_app()` / `asgi_app()` for Flask/Dgango/FastAPI backend.
  - Add remote access service to `start_server()`
  - Add `max_file_upload`/`payload_size_limit`/`upload_size_limit`/`max_payload_size` parameters to `start_server()`.
- So many other improvements.

Bug fix

- Fix table style.
- Fix large file uploading error.
- Fix server start error when enabled `auto_open_webbrowser`.
- Fix file names overflow in file input.
- Fix *put_image()* raise 'unknown file extension' error when use PIL Image as `src`.
- Sanitize the returned `filename` of *file_upload()* to avoid interpreting as path accidentally.
- So many other bugs fixed.

4.10.7 What's new in PyWebIO 1.2

2021 3/18

Highlights

- Support reconnect to server in websocket connection by setting `reconnect_timeout` parameter in *`start_server()`*.
- Add *`path_deploy()`*, *`path_deploy_http()`* and `pywebio-path-deploy` command to deploy PyWebIO applications from a directory.
- All documents and demos are now available in English version.
- Some output-related functions support context manager, see *output functions list* .

Detailed changes

- Add `put_info()`, `put_error()`, `put_warning()`, `put_success()`
- Add `pywebio.utils.pyinstaller_datas()` to get PyWebIO data files when using pyinstaller to bundle PyWebIO application.
- Add documentation for data visualization using pyg2plot.
- The `reset()`, `append()`, `insert()` of `output()` accept any type as content.
- Add `static_dir` parameter to `start_server()` to serve static files.
- Deprecated `pywebio.session.get_info()`, use *`pywebio.session.info`* instead.
- Alert not supporting message when the user using IE browser.

4.10.8 What's new in PyWebIO 1.1

2021 2/7

It's been a whole year since the first line of PyWebIO code was written. There have been too many things in 2020, but it has a special meaning to me. In 2021, we will continue to work hard to make PyWebIO better and better.

Highlights

- Security support: *`put_html()`*, *`put_markdown()`* can use `sanitize` parameter to prevent XSS attack.
- UI internationalization support
- SEO support: Set SEO info through `pywebio.platform.seo()` or function docstring
- CDN support, more convenient to web framework integration
- Application access speed is improved, and no probe requests are used to determine the communication protocol

Backwards-incompatible changes

- Remove `disable_asyncio` parameter of `start_server()` in django and flask.
- Deprecated `pywebio.session.data()`, use *pywebio.session.local* instead
- Application integrated into the web framework, the access address changes, see *Web framework integration*
- Remove `max_height` parameter of *put_scrollable()*, use `height` instead

Detailed changes

- *put_code()* add `rows` parameter to limit the maximum number of displayed lines
- *put_scrollable()* add `keep_bottom` parameter
- *put_markdown()* add `options` to config Markdown parsing options.
- Add html escaping for parameters of *put_code()*, *put_image()*, *put_link()*, *put_row()*, *put_grid()*
- Methods `reset()`, `append()`, `insert()` of `output()` accept string content
- Fix: Parsing error in `max_size` and `max_total_size` parameters of *file_upload()*
- Fix: Auto open browser failed in python 3.6

4.10.9 What's new in PyWebIO 1.0

2021 1/17

PyWebIO 1.0 v0.3

Highlights

- `start_server` PyWebIO *go_app()*
- Scope
- *put_grid()*, *put_row()*, *put_column()* *style()*
- : *toast()*, *popup()*, *put_widget()*, *put_collapse()*, *put_link()*, *put_scrollable()*, *put_loading()*, `put_processbar()`
- *span()*, `output()`
- JS: *run_js()*, *eval_js()*
- UI: console

Backwards-incompatible changes

-
- `pywebio.output.set_output_fixed_height()`
- `pywebio.output.set_title()` , `pywebio.output.set_auto_scroll_bottom()` *pywebio.session.set_env()*
- `pywebio.output.table_cell_buttons()` *pywebio.output.put_buttons()*

Detailed changes by module

- *input()* action
- *file_upload()*
- *put_buttons()*
- *put_widget()* *popup()* *put_table()* Html
- *put_text()*
- *put_image()* Url

4.10.10 What's new in PyWebIO 0.3

2020 5/13

Highlights

- bokeh
- `session.get_info()`
- jstypescript
- *output.put_table()* / `put_xxx`

Detailed changes by module

UI

-

`pywebio.output`

-
- `table_cell_buttons()`

4.10.11 What's new in PyWebIO 0.2

2020 4/30

Highlights

- Djangoaiohttp Web
- plotlypyecharts
- Web
- *defer_call() hold()*
- *put_image() remove(anchor)*
- UI
- CI

Detailed changes by module

UI

-
- footer

pywebio.input

- *input_group()* cancelable
- *actions()* button reset cancel

pywebio.output

- anchor
- clear_range()
- *scroll_to(anchor, position)* position

pywebio.platform

- start_server webio_view webio_handle

pywebio.session

- Session PyWebIO `SessionClosedException`
- fix: Session `functools.partial`

4.11 pywebio_battery — PyWebIO battery

Utilities that help write PyWebIO apps quickly and easily.

Note: `pywebio_battery` is an extension package of PyWebIO, you must install it before using it. To install this package, run `pip3 install -U pywebio-battery`

4.11.1 Functions list

Interaction related

Function name	Description
confirm	Confirmation modal
popup_input	Show a form in popup window
redirect_stdout	redirecting stdout to pywebio
run_shell	Run command in shell
put_logbox, *logbox_append*	Logbox widget
put_video	Output video
put_audio	Output audio

Web application related

Function name	Description
get_all_query, *get_query*	Get URL parameter
set_localstorage, *get_localstorage*	User browser storage
set_cookie, *get_cookie*	Web Cookie
basic_auth, *custom_auth*, *revoke_auth*	Authentication

pywebio_battery.**confirm**(*title: str*, *content: Optional[Union[str, pywebio.io_ctrl.Output, Sequence[Union[str, pywebio.io_ctrl.Output]]]] = None*, ***, *timeout: Optional[int] = None*) → Optional[bool]

Show a confirmation modal.

Parameters

- **title** (*str*) – Model title.
- **content** (*list/put_xxx()/str*) – The content of the confirmation modal. Can be a string, the put_xxx() calls , or a list of them.
- **timeout** (*None/float*) – Seconds for operation time out.

Returns Return `True` when the "CONFIRM" button is clicked, return `False` when the "CANCEL" button is clicked, return `None` when a timeout is given and the operation times out.

```
choice = confirm("Delete File", "Are you sure to delete this file?")
put_text("Your choice", choice)
```

pywebio_battery.**popup_input**(*pins: Sequence[pywebio.io_ctrl.Output]*, *title='Please fill out the form below'*) → Optional[dict]

Show a form in popup window.

Parameters

- **pins** (*list*) – *pin* widget list. It can also contain ordinary output widgets.
- **title** (*str*) – model title.

Returns return the form value as dict, return `None` when user cancel the form.

```
form = popup_input([
    put_input("username", label="User name"),
    put_input("password", type=PASSWORD, label="Password"),
    put_info("If you forget your password, please contact the administrator."),
], "Login")
put_text("Login info:", form)
```

pywebio_battery.**redirect_stdout**(*output_func=functools.partial(<function put_text>, inline=True)*)

Context manager for temporarily redirecting stdout to pywebio.

```
with redirect_stdout():
    print("Hello world.")
```

pywebio_battery.**run_shell**(*cmd: str*, *output_func=functools.partial(<function put_text>, inline=True)*, *encoding='utf8'*) → int

Run command in shell and output the result to pywebio

Parameters

- **cmd** (*str*) – command to run
- **output_func** (*callable*) – output function, default to `put_text()`. the function should accept one argument, the output text of command.
- **encoding** (*str*) – command output encoding

Returns shell command return code

Changed in version 0.4: add `encoding` parameter and return code

pywebio_battery.**put_logbox**(*name: str*, *height=400*, *keep_bottom=True*) → pywebio.io_ctrl.Output

Output a logbox widget

```
import time

put_logbox("log")
while True:
    logbox_append("log", f"{time.time()}\n")
    time.sleep(0.2)
```

Parameters

- **name** (*str*) – the name of the widget, must unique in session-wide.
- **height** (*int*) – the height of the widget in pixel

- **keep_bottom** (*bool*) – Whether to scroll to bottom when new content is appended (via *logbox_append()*).

Changed in version 0.3: add `keep_bottom` parameter

pywebio_battery.**logbox_append**(*name: str*, *text: str*)
Append text to a logbox widget

pywebio_battery.**put_video**(*src: Union[str, bytes]*, *autoplay: bool = False*, *loop: bool = False*, *height: Optional[int] = None*, *width: Optional[int] = None*, *muted: bool = False*, *poster: Optional[str] = None*, *scope: Optional[str] = None*, *position: int = - 1*) → pywebio.io_ctrl.Output
Output video

Parameters

- **src** (*str/bytes*) – Source of video. It can be a string specifying video URL, a bytes-like object specifying the binary content of the video.
- **autoplay** (*bool*) – Whether to autoplay the video. In some browsers (e.g. Chrome 70.0) autoplay doesn't work if not enable `muted`.
- **loop** (*bool*) – If True, the browser will automatically seek back to the start upon reaching the end of the video.
- **width** (*int*) – The width of the video's display area, in CSS pixels. If not specified, the intrinsic width of the video is used.
- **height** (*int*) – The height of the video's display area, in CSS pixels. If not specified, the intrinsic height of the video is used.
- **muted** (*bool*) – If set, the audio will be initially silenced.
- **poster** (*str*) – A URL for an image to be shown while the video is downloading. If this attribute isn't specified, nothing is displayed until the first frame is available, then the first frame is shown as the poster frame.
- **scope, position** (*int*) – Those arguments have the same meaning as for *put_text()*

Example:

```
url = "https://interactive-examples.mdn.mozilla.net/media/cc0-videos/flower.mp4"
put_video(url)
```

New in version 0.4.

pywebio_battery.**put_audio**(*src: Union[str, bytes]*, *autoplay: bool = False*, *loop: bool = False*, *muted: bool = False*, *scope: Optional[str] = None*, *position: int = - 1*) → pywebio.io_ctrl.Output
Output audio

Parameters

- **src** (*str/bytes*) – Source of audio. It can be a string specifying video URL, a bytes-like object specifying the binary content of the audio.
- **autoplay** (*bool*) – Whether to autoplay the audio.
- **loop** (*bool*) – If True, the browser will automatically seek back to the start upon reaching the end of the audio.
- **muted** (*bool*) – If set, the audio will be initially silenced.

- **scope** – The scope of the video. It can be `"session"` or `"page"`. If not specified, the video will be automatically removed when the session is closed.
- **scope, position** (*int*) – Those arguments have the same meaning as for *put_text()*

Example:

```
url = "https://interactive-examples.mdn.mozilla.net/media/cc0-audio/t-rex-roar.mp3
↪"
put_audio(url)
```

New in version 0.4.

pywebio_battery.**get_all_query**()
Get URL parameter (also known as "query strings" or "URL query parameters") as a dict

pywebio_battery.**get_query**(*name: str*)
Get URL parameter value

pywebio_battery.**set_localstorage**(*key: str*, *value: str*)
Save data to user's web browser

The data is specific to the origin (protocol+domain+port) of the app. Different origins use different web browser local storage.

Parameters

- **key** – the key you want to create/update.
- **value** – the value you want to give the key you are creating/updating.

You can read the value by using *get_localstorage(key)*

pywebio_battery.**get_localstorage**(*key: str*) → str
Get the key's value in user's web browser local storage

pywebio_battery.**set_cookie**(*key: str*, *value: str*, *days=7*)
Set cookie

pywebio_battery.**get_cookie**(*key: str*)
Get cookie

pywebio_battery.**basic_auth**(*verify_func: Callable[[str, str], bool]*, *secret: Union[str, bytes]*, *expire_days=7*, *token_name='pywebio_auth_token'*) → str
Persistence authentication with username and password.

You need to provide a function to verify the current user based on username and password. The `basic_auth()` function will save the authentication state in the user's web browser, so that the authed user does not need to log in again.

Parameters

- **verify_func** (*callable*) – User authentication function. It should receive two arguments: username and password. If the authentication is successful, it should return `True`, otherwise return `False`.
- **secret** (*str*) – HMAC secret for the signature. It should be a long, random str.
- **expire_days** (*int*) – how many days the auth state can keep valid. After this time, authed users need to log in again.
- **token_name** (*str*) – the name of the token to store the auth state in user browser.

Return str username of the current authed user

Example:

```
user_name = basic_auth(lambda username, password: username == 'admin' and␣
↪password == '123',
                       secret="__TODO:_GENERATE_YOUR_OWN_RANDOM_VALUE_HERE__")
put_text("Hello, %s. You can refresh this page and see what happen" % user_name)
```

New in version 0.4.

pywebio_battery.**custom_auth**(*login_func: Callable[], str]*, *secret=typing.Union[str, bytes]*, *expire_days=7*, *token_name='pywebio_auth_token'*) → str

Persistence authentication with custom logic.

You need to provide a function to determine the current user and return the username. The `custom_auth()` function will save the authentication state in the user's web browser, so that the authed user does not need to log in again.

Parameters

- **login_func** (*callable*) – User login function. It should receive no arguments and return the username of the current user. If fail to verify the current user, it should return `None`.
- **secret** (*str*) – HMAC secret for the signature. It should be a long, random str.
- **expire_days** (*int*) – how many days the auth state can keep valid. After this time,authed users need to log in again.
- **token_name** (*str*) – the name of the token to store the auth state in user browser.

Return str username of the current authed user.

New in version 0.4.

pywebio_battery.**revoke_auth**(*token_name='pywebio_auth_token'*)

Revoke the auth state of current user

Parameters **token_name** (*str*) – the name of the token to store the auth state in user browser.

New in version 0.4.

4.12 Server-Client communication protocol

PyWebIO uses a server-client architecture, the server executes task code, and interacts with the client (that is, the user browser) through the network. This section introduce the protocol specification for the communication between PyWebIO server and client.

There are two communication methods between server and client: WebSocket and Http.

When using Tornado or aiohttp backend, the server and client communicate through WebSocket, when using Flask or Django backend, the server and client communicate through Http.

WebSocket communication

The server and the client send json-serialized message through WebSocket connection

Http communication

- The client polls the backend through Http GET requests, and the backend returns a list of PyWebIO messages serialized in json

- When the user submits the form or clicks the button, the client submits data to the backend through Http POST request

In the following, the data sent by the server to the client is called command, and the data sent by the client to the server is called event.

The following describes the format of command and event

4.12.1 Command

Command is sent by the server to the client. The basic format of command is:

```
{
    "command": ""
    "task_id": ""
    "spec": {}
}
```

Each fields are described as follows:

- `command` : command name
- `task_id` : Id of the task that send the command
- `spec` : the data of the command, which is different depending on the command name

Note that: the arguments shown above are merely the same with the parameters of corresponding PyWebIO functions, but there are some differences.

The following describes the `spec` fields of different commands:

input_group

Show a form in user's browser.

Table 2: fields of `spec`

Field	Required	Type	Description
label	False	str	Title of the form
inputs	True	list	Input items
cancelable	False	bool	Whether the form can be cancelled If cancelable=True, a "Cancel" button will be displayed at the bottom of the form. A `from_cancel` event is triggered after the user clicks the cancel button.

The `inputs` field is a list of input items, each input item is a `dict`, the fields of the item are as follows:

- label: Label of input field, required.
- type: Input type, required.

- name: Identifier of the input field, required.
- onchange: bool, whether to push input value when input change
- onbulr: bool, whether to push input value when input field `onblur`
- auto_focus: Set focus automatically. At most one item of `auto_focus` can be true in the input item list
- help_text: Help text for the input
- Additional HTML attribute of the input element
- Other attributes of different input types

Currently supported `type` are:

- text: Plain text input
- number: Number input
- password: Password input
- checkbox: Checkbox
- radio: Radio
- select: Drop-down selection
- textarea: Multi-line text input
- file: File uploading
- actions: Actions selection.

Correspondence between different input types and html input elements:

- text: input[type=text]
- number: input[type=number]
- float: input[type=text], and transform input value to float
- password: input[type=password]
- checkbox: input[type=checkbox]
- radio: input[type=radio]
- select: select https://developer.mozilla.org/zh-CN/docs/Web/HTML/Element/select
- textarea: textarea https://developer.mozilla.org/zh-CN/docs/Web/HTML/Element/textarea
- file: input[type=file]
- actions: button[type=submit] https://developer.mozilla.org/zh-CN/docs/Web/HTML/Element/button

Unique attributes of different input types:

- text,number,password:
  - action: Display a button on the right of the input field. The format of `action` is `{label: button label, callback_id: button click callback id}`
- textarea:
  - code: Codemirror options or boolean, same as `code` parameter of *`pywebio.input.textarea()`*
- select
  - options: `{label:, value: , [selected:,] [disabled:]}`

- checkbox:
  - options: `{label:, value: , [selected:,] [disabled:]}`
  - inline
- radio:
  - options: `{label:, value: , [selected:,] [disabled:]}`
  - inline
- actions
  - buttons: `{label:, value:, [type: 'submit'/'reset'/'cancel'], [disabled:], [color:]}`.
- file:
  - multiple: Whether to allow upload multiple files.
  - max_size: The maximum size of a single file, in bytes.
  - max_total_size: The maximum size of all files, in bytes.
- slider
  - min_value: The minimum permitted value.
  - max_value: The maximum permitted value.
  - step: The stepping interval.
  - float: If need return a float value

update_input

Update the input item, you can update the `spec` of the input item of the currently displayed form

The `spec` fields of `update_input` commands:

- target_name: str The name of the target input item.
- target_value: str, optional. Used to filter item in checkbox, radio
- attributes: dist, fields need to be updated
  - valid_status: When it is bool, it means setting the state of the input value, pass/fail; when it is 0, it means clear the valid_status flag
  - value: Set the value of the item
  - label
  - placeholder
  - invalid_feedback
  - valid_feedback
  - help_text
  - options: only available in checkbox, radio and select type
  - other fields of item's `spec` // not support the `inline` field

close_session

Indicates that the server has closed the connection. `spec` of the command is empty.

set_session_id

Send current session id to client, used to reconnect to server (Only available in websocket connection). `spec` of the command is session id.

destroy_form

Destroy the current form. `spec` of the command is empty.

Note: The form will not be automatically destroyed after it is submitted, it needs to be explicitly destroyed using this command

output

Output content

The `spec` fields of `output` commands:

- type: content type
- style: str, Additional css style
- container_selector: The css selector of output widget's content slot. If empty(default), use widget self as container
- container_dom_id: The dom id need to be set to output widget's content slot.
- scope: str, CSS selector of the output container. If multiple containers are matched, the content will be output to every matched container
- position: int, see *scope - User manual*
- click_callback_id:
- Other attributes of different types

`container_selector` and `container_dom_id` is used to implement output context manager.

Unique attributes of different types:

- type: markdown
  - content: str
  - options: dict, marked.js options
  - sanitize: bool, Whether to enable a XSS sanitizer for HTML
- type: html
  - content: str
  - sanitize: bool, Whether to enable a XSS sanitizer for HTML
- type: text
  - content: str
  - inline: bool, Use text as an inline element (no line break at the end of the text)

- type: buttons
  - callback_id:
  - buttons:[{value:, label:, [color:], [disabled:]},. . .]
  - small: bool, Whether to enable small button
  - group: bool, Whether to group the buttons together
  - link: bool, Whether to make button seem as link.
  - outline: bool, Whether enable outline style.
- type: file
  - name: File name when downloading
  - content: File content with base64 encoded
- type: table
  - data: Table data, which is a two-dimensional list, the first row is table header.
  - span: cell span info. Format: {"[row id],[col id]": {"row":row span, "col":col span }}
- type: pin
  - input: input spec, same as the item of `input_group.inputs`
- type: scope
  - dom_id: the DOM id need to be set to this widget
  - contents list: list of output spec
- type: scrollable
  - contents:
  - min_height:
  - max_height:
  - keep_bottom:
  - border:
- type: tabs
  - tabs:
- type: custom_widget
  - template:
  - data:

pin_value

The `spec` fields of `pin_value` commands:

- name

pin_update

The `spec` fields of `pin_update` commands:

- name
- attributes: dist, fields need to be updated

pin_wait

The `spec` fields of `pin_wait` commands:

- names: list,
- timeout: int,

pin_onchange

set a callback which is invoked when the value of pin widget is changed

The `spec` fields of `pin_onchange` commands:

- name: string
- callback_id: string, if `None`, not set callback
- clear: bool

popup

Show popup

The `spec` fields of `popup` commands:

- title
- content
- size: `large`, `normal`, `small`
- implicit_close
- closable
- dom_id: DOM id of popup container element

toast

Show a notification message

The `spec` fields of `popup` commands:

- content
- duration
- position: `'left'`/`'center'`/`'right'`
- color: hexadecimal color value starting with '#'
- callback_id

close_popup

Close the current popup window.

`spec` of the command is empty.

set_env

Config the environment of current session.

The `spec` fields of `set_env` commands:

- title (str)
- output_animation (bool)
- auto_scroll_bottom (bool)
- http_pull_interval (int)
- input_panel_fixed (bool)
- input_panel_min_height (int)
- input_panel_init_height (int)
- input_auto_focus (bool)

output_ctl

Output control

The `spec` fields of `output_ctl` commands:

- set_scope: scope name
  - container: Specify css selector to the parent scope of target scope.
  - position: int, The index where this scope is created in the parent scope.
  - if_exist: What to do when the specified scope already exists:
    - null: Do nothing
    - `'remove'`: Remove the old scope first and then create a new one
    - `'clear'`: Just clear the contents of the old scope, but don't create a new scope

    * `'blank'`: Clear the contents of the old scope and keep the height, don't create a new scope

- loose: css selector of the scope, set the scope not to keep the height (i.e., revoke the effect of `set_scope(if_exist='blank')`)
- clear: css selector of the scope need to clear
- clear_before
- clear_after
- clear_range:[,]
- **scroll_to**
  - position: top/middle/bottom, Where to place the scope in the visible area of the page
- remove: Remove the specified scope

run_script

run javascript code in user's browser

The `spec` fields of `run_script` commands:

- code: str, code
- args: dict, Local variables passed to js code
- eval: bool, whether to submit the return value of javascript code

download

Send file to user

The `spec` fields of `download` commands:

- name: str, File name when downloading
- content: str, File content in base64 encoding.

4.12.2 Event

Event is sent by the client to the server. The basic format of event is:

```
{
    event: event name
    task_id: ""
    data: object/str
}
```

The `data` field is the data carried by the event, and its content varies according to the event. The `data` field of different events is as follows:

input_event

Triggered when the form changes

- event_name: Current available value is `'blur'`, which indicates that the input item loses focus
- name: name of input item
- value: value of input item

note: checkbox and radio do not generate blur events

callback

Triggered when the user clicks the button in the page

In the `callback` event, `task_id` is the `callback_id` field of the `button`; The `data` of the event is the `value` of the button that was clicked

from_submit

Triggered when the user submits the form

The `data` of the event is a dict, whose key is the name of the input item, and whose value is the value of the input item.

from_cancel

Cancel input form

The `data` of the event is `None`

js_yield

submit data from js. It's a common event to submit data to backend.

The `data` of the event is the data need to submit

CHAPTER

FIVE

INDICES AND TABLES

- genindex
- modindex
- search

CHAPTER

SIX

DISCUSSION AND SUPPORT

- Need help when use PyWebIO? Make a new discussion on Github Discussions.
- Report bugs on the GitHub issue.

PYTHON MODULE INDEX

p

INDEX

R

S

T

U

W