PyDataFront User Guide

0. Hello, world!

PyDataFront enables you to build a web interface easily in Python. You will find it particularly useful for Machine Learning, Data Science, Bioinformatics, NLP, and Computer Vision.

To do so, simply define a Python function as usual, and put a decorator right above it. Like below:

```
@textea_export()
def hello_world(your_name: str) -> str:
    return f"Welcome to PyDataFront, {your_name}!"
```

Suppose you save this file as `hello_world.py` Then, you can start a PyDataFront server to turn this function into a web API:

```
python3 -m pydatafront ./hello_world
```

And fire up the web visualizer:

```
cd frontend
npm install react-scripts
yarn start
```

Finally, you have a web form that will change the displayed message based the value of `your_name`:

More Complicated Example: calc()

Our Products

Textea Sheet: Our commercial sheet program with PyDataFront integration

Textea Redstone: Serverless, swift deployment of code to customer with PyDataFront

Git Branches

dev BRANCH: `34a5bdf`

main BRANCH: `ac64e88`

1. Typing and Widgets

Every argument or return of a function to be converted by PyDataFront must have a typing hint, without which PyDataFront cannot properly display the widgets to accept user inputs or display execution results. We use Python's type hints (including `typing` with Python doc here), and widgets of Material UI (MUI), React JSON Schema Form (RJSF, ONLY FOR dev BRANCH), and Textea JSON Viewer (TJV).

Argument Type

An **argument/input** of a function to be converted by PyDataFront must be of the following types:

Type	Description	Widget	Future
`int`		1. MUI AutoComplete TextField accepting integers only (ONLY FOR dev BRANCH) 2. MUI TextField (ONLY FOR main BRANCH)	MUI Slider widget option
`float`		1. MUI AutoComplete TextField accepting numbers only (ONLY FOR dev BRANCH) 2. MUI TextField (ONLY FOR main BRANCH)	MUI Slider widget option

Type	Description	Widget	Future
`str`		1. MUI AutoComplete TextField (ONLY FOR dev BRANCH) 2. MUI TextField (ONLY FOR main BRANCH)	
`bool`	ONLY FOR dev BRANCH	MUI Checkbox	MUI Switch widget option
`typing.List[X]`	1. The only way to define columns 2. `X` can only be `int`, `float`, `str` or `bool` 3. All `typing.List[X]` arguments will be displayed in one DataGrid together 4. (ONLY FOR dev BRANCH) Nested lists for PyDataFront	1. RJSF ArrayField (with MUI theme) 2. JSON editor (legacy, pre-RJSF, ONLY FOR main BRANCH)	1. MUIX DataGrid widget option (without nested list support) 2. JSON editor widget option
`typing.Optional[X]`	1. NOT RECOMMENDED, ONLY FOR COMPATIBILITY 2. For Python, equivalent to `typing.Union[X, None]`; for PyDataFront, equivalent to `X`	Same as widget for type `X`	
`typing.Literal`	1. (ONLY FOR dev BRANCH) Replacement for deprecated `whitelist` parameter for a list of possible function argument to reject other values 2. CONFLICT WITH `example` 3. (ONLY FOR dev BRANCH) ONLY SUPPORT singular arguments	MUI AutoComplete TextField accepting whitelisted elements only	Non-singular `typing.Literal` value support
`list`, `dict`, `typing.Dict`, Any	(ONLY FOR dev BRANCH) NOT SUPPORTED (ONLY FOR main BRANCH) Through legacy function	JSON editor (legacy, pre-RJSF, ONLY FOR main BRANCH)	JSON editor widget option

An Example of bool: calc_boolean_add()

PyDataFront's web frontend will enforce the type check. For example, if a `string` is entered in an input box intended for an `integer`, an error will be thrown out. For the backend, there is no type check intended in nature of Python.

Return Type

A **return** of a function to be converted by PyDataFront must be of the following types:

Type	Description	Widget	Future
`int`, `float`, `str`, `bool`, Any	ONLY FOR PyDataFront AND dev BRANCH	string in `<code>`	
`typing.List`, `list`	ONLY FOR PyDataFront AND dev BRANCH	TJV ReactJson	MUI Table (without nested list support) option
`typing.TypedDict`	1. ALLOW FOR Textea Sheet AND PyDataFront 2. In Textea Sheet, if `TypedDict` is a return, then it is the only return	TJV ReactJson	MUI Table (without nested list support) option
`typing.Dict`, `dict`	1. ONLY FOR PyDataFront AND dev BRANCH 2. Simpler version of `TypedDict`	TJV ReactJson	MUI Table (without nested list support) option
Error	1. Catched Python error through `except` 2. Stringified JSON dict with two keys `error_type` (which could only be `function` or `wrapper`) and `error_body` (to be `traceback.format_exc()`)		

An Example of Error

Default Value

(ONLY FOR dev BRANCH) The default value of a function argument will automatically be the default value in the JSON schema to be rendered by PyDataFront frontend, as well as an element in the list of `example` which will be introduced in Section **Parameters for Arguments of Decorated Function**.

There are special cases of the default value: even if the default value is not stated at the argument property in the function signature, there are two default values to be automatically assigned if missing (i.e., the default value of the default value, or the default assigned value). If the argument type is `bool`, the default assigned value is `False`; if the argument type is of `typing.Optional`, the default assigned value is `None`.

An Example of Default Value: calc_default_add()

2. PyDataFront Decorator and Parameters

PyDataFront decorator gives Python programmers more control on their web UI.

There are three optional parameters for the decorated function itself, and the remaining parameters are for the arguments of the decorated function, as defined below.

```
def textea_export(path: Optional[str] = None, description: Optional[str] = "",
                  destination: Literal["column", "row", "sheet", None] = None,
                  **decorator_kwargs):translate from English to French: I'm very happy
```

Parameters for Decorated Function

Parameter	Description	Default Value	Future
`path`	1. The path for the frontend to fetch parameters (i.e., URL `/param/{path}`) (ONLY FOR dev BRANCH); the path for the frontend to both fetch parameters and call function (i.e., URL `/param/{path}`) (ONLY FOR main BRANCH) 2. MUST BE UNIQUE or be failed to launch	(ONLY FOR dev BRANCH) function name (so that the function name must be unique if `path` not defined)	
`description`	A paragraph describing the function, to be displayed on the frontend	`""` (empty string)	Markdown-based rich text
`destination`	1. ONLY FOR Textea Sheet 2. One of `column`, `row`, and `sheet` 3. See detailed description below	`None`	

destination — Parameter for Decorated Function

Parameters for Arguments of Decorated Function

The other parameters are optional and for the arguments instead of the function itself. We currently support three sub-parameters for arguments as follows.

```
arg_name={
  sub_param_1: ..., # sub-parameter
  ...
}
```

Sub-Parameter	Type	Description	Default Value	Future
`treat_as`	`typing.Literal["config", "column", "cell"]`	1. CURRENTLY ONLY FOR Textea Sheet 2. one of `config`, `column`, and `cell` 3. `cell` is NOT SUPPORTED 4. See detailed description below	(ONLY FOR dev BRANCH) `config`	`cell` support with partial function
`whitelist`	`typing.List[X]` with `X` be argument type	1. ONLY FOR main BRANCH, DEPRECATED FOR dev BRANCH 2. CONFLICT WITH `example` 3. Same functionality as `typing.Literal` (dev ONLY) FOR main BRANCH	`undefined`	Subject to removal
`example`	`typing.List[X]` with `X` be argument type	1. CONFlCT WITH `whitelist`, `typing.Literal` 2. (ONLY FOR dev BRANCH) Include default value at the end of list 3. (ONLY FOR dev BRANCH) ONLY SUPPORT singular arguments	`undefined`	Non-singular `example` value support

treat_as — Parameter for Arguments of Decorated Function

3. Web API

What do we pass to PyDataFront's frontend or Textea Sheet. The JSON contents of endpoints are automatically generated by the PyDataFront decorator.

Endpoints

- `/list`: Returns a list of functions under the top-level `"list"` key
  - (ONLY FOR dev BRANCH) The element under the top-level `"list"` key is a list of functions with `"name"` and `"path"` keys and `string` elements

```
{
    "list": [
        {
            "name": "func1",
            "path": "path1"
        },
        {
            "name": "...",
            "path": "..."
        }
    ]
}
```

  - (ONLY FOR main BRANCH) The element under the top-level `"list"` key is a list of functions with `"id"`, `"name"`, `"path"`, and `"description"` keys and `string` elements

```
{
    "list": [
        {
            "id": "uuid1",
            "name": "func1",
            "path": "path1",
            "description": "desc1"
        },
        {
            "id": "...",
            "name": "...",
            "path": "...",
            "description": "..."
        }
    ]
}
```

  - Fields

    - `id`: For the function call, we are using `id` to support multiple servers hosting same function (as not necessary, only kept in `/param/{PATH}` endpoint ONLY FOR dev BRANCH)
    - `name`: Function name
    - `path`: Same as described in Section **Parameters for Decorated Function**
    - `description`: Same as described in Section **Parameters for Decorated Function** (as not necessary, only kept in `/param/{PATH}` endpoint ONLY FOR dev BRANCH)
- `/param/{PATH}`: Returns a list of parameters for the frontend, the information is mainly parsed from the decorator parameters and function signature
    - (ONLY FOR dev BRANCH) Example

```
{
    "description": "perform some basic math calculation",
    "destination": "column",
    "id": "d742a684-0d43-45bd-a54d-224652b779a8",
    "name": "calc",
    "return_type": {
        "output": "typing.List[int]"
    },
    "params": {
        "a": {
            "treat_as": "column",
            "type": "typing.List[int]"
        },
        "b": {
            "treat_as": "column",
            "type": "typing.List[int]"
        },
        "op": {
            "treat_as": "config",
            "type": "str",
            "whitelist": [
                "add",
                "minus"
            ]
        }
    },
    "schema": {
        "description": "perform some basic math calculation",
        "properties": {
            "a": {
                "items": {
                    "type": "integer"
                },
                "type": "array"
            },
```

```
            "b": {
                "items": {
                    "type": "integer"
                },
                "type": "array"
            },
            "op": {
                "type": "string",
                "whitelist": [
                    "add",
                    "minus"
                ]
            }
        },
        "title": "calc",
        "type": "object"
    }
}
```

- Fields
  - `description`: Same as `description` described in Section **Fields of `/list`**
  - `destination`: Same as `destination` described in Section **Parameters for Decorated Function**
  - `id`: Same as `id` described in Section **Fields of `/list`**
  - `name`: Same as `name` described in Section **Fields of `/list`**
  - `return_type`: (ONLY FOR dev BRANCH) Frontend parameter for Textea Sheet, as described in Section **Return Type**
  - `output_type`: (ONLY FOR main BRANCH) Same as `return_type`
  - `params`: Frontend parameters for Textea Sheet and PyDataFront frontend (ONLY FOR main BRANCH), for every argument name, we have
    - `treat_as`: Same as `treat_as` described in Section **Parameters for Arguments of Decorated Function**
    - `whitelist`: (ONLY FOR main BRANCH) Same as `whitelist` described in Section **Parameters for Arguments of Decorated Function**; (ONLY FOR dev BRANCH) Same as `typing.Literal` described in Section **Argument Type**
    - `example`: Same as `example` described in Section **Parameters for Arguments of Decorated Function**
    - `default`: Same as described in Section **Default Value**

    - `schema`: (ONLY FOR dev BRANCH) JSON schema of the function as `object` for RJSF (doc here)
    - `callee`: (ONLY FOR main BRANCH) The path to call the function, i.e., `/call/{PATH}`

- `/call/{ID}`: (ONLY FOR dev BRANCH) Returns function call result as `string` (ONLY FOR PyDataFront) or stringified JSON, type as described in Section **Return Type**
- `/call/{PATH}`: (ONLY FOR main BRANCH) Returns function call result as stringified JSON, type as described in Section **Return Type**