dargs Documentation

Release 0.0.0-rc

DeepModeling

Apr 06, 2024

CONTENTS:

CHAPTER

ONE

ARGUMENT CHECKING FOR PYTHON PROGRAMS

This is a minimum version for checking the input argument dict. It would examine argument's type, as well as keys and types of its sub-arguments.

A special case called *variant* is also handled, where you can determine the items of a dict based the value of on one of its `flag_name` key.

There are three main methods of `Argument` class:

- `check` method that takes a dict and see if its type follows the definition in the class
- `normalize` method that takes a dict and convert alias and add default value into it
- `gendoc` method that outputs the defined argument structure and corresponding docs

There are also `check_value` and `normalize_value` that ignore the leading key comparing to the base version.

When targeting to html rendering, additional anchor can be made for cross reference. Set `make_anchor=True` when calling `gendoc` function and use standard ref syntax in rst. The id is the same as the argument path. Variant types would be in square brackets.

Please refer to test files for detailed usage.

CHAPTER

TWO

USE WITH SPHINX

Add *dargs.sphinx* to the extensions of conf.py:

```
extensions = [
    'dargs.sphinx',
]
```

Then *dargs* directive will be enabled:

```
.. dargs::
   :module: dargs.sphinx
   :func: _test_argument
```

where *_test_argument* returns an *Argument*. The documentation will be rendered as:

test:

type: dict | str
argument path: test

This argument/variant is only used to test.

test_argument:

type: str
argument path: test/test_argument

This argument/variant is only used to test.

test_list:

type: typing.List[int], optional
argument path: test/test_list

Depending on the value of *test_variant*, different sub args are accepted.

test_variant:

type: str (flag key)
argument path: test/test_variant
possible choices: *test_variant_argument*

This argument/variant is only used to test.

When test_variant is set to test_variant_argument:

This argument/variant is only used to test.

test_repeat:

type: `list`
argument path: `test[test_variant_argument]/test_repeat`

This argument/variant is only used to test.

This argument takes a list with each element containing the following:

test_repeat_item:

type: `bool`
argument path: `test[test_variant_argument]/test_repeat/test_repeat_item`

This argument/variant is only used to test.

A `list` of *`Argument`* is also accepted.

test1:

type: `int`
argument path: `test1`

Argument 1

test2:

type: `NoneType | float`
argument path: `test2`

Argument 2

test3:

type: `typing.List[str]`, optional, default: `['test']`
argument path: `test3`

Argument 3

2.1 Cross-referencing Arguments

Both the following ways will create a cross-reference to the argument:

```
Both :dargs:argument:`this <test/test_argument>` and :ref:`this <test/test_argument>`␣
↪will create a cross-reference to the argument!
```

It will be rendered as:

Both *`this`* and *this* will create a cross-reference to the argument!

2.2 Index page

The arguments will be added into the genindex page. See *test_argument* in the genindex page.

CHAPTER

THREE

USE WITH DP-GUI

Developers can export an *Argument* to DP-GUI for users, by adding a `dpgui` entry point to *pyproject.toml*:

```
[project.entry-points."dpgui"]
 "Test Argument" = "dargs.sphinx:_test_argument"
```

where *_test_argument* returns an *Argument* or a list of *Argument*, and `"Test Argument"` is its name that can be any string.

Users can install the *dpgui* Python package via *pip* and serve the DP-GUI instance using the *dpgui* command line:

```
pip install dpgui
dpgui
```

The served DP-GUI will automatically load all templates from the `dpgui` entry point.

CHAPTER

FOUR

USE WITH JUPYTER NOTEBOOK

In a Jupyter Notebook, with *dargs.notebook.JSON()*, one can render an JSON string with an Argument.

```
from dargs.sphinx import _test_argument
from dargs.notebook import JSON

jstr = """
{
    "test_argument": "test1",
    "test_list": [
        1,
        2,
        3
    ],
    "test_variant": "test_variant_argument",
    "test_repeat": [
        {"test_repeat_item": false},
        {"test_repeat_item": true}
    ],
    "_comment": "This is an example data"
}
"""
JSON(jstr, _test_argument())
```

```
<IPython.core.display.HTML object>
```

When the monse stays on an argument key, the documentation of this argument will pop up.

CHAPTER

FIVE

API DOCUMENTATION

5.1 dargs package

class dargs.**Argument**(*name: str*, *dtype: None | type | Iterable[type]*, *sub_fields: Iterable[*Argument*] | None = None*, *sub_variants: Iterable[*Variant*] | None = None*, *repeat: bool = False*, *optional: bool = False*, *default: Any = _Flags.NONE*, *alias: Iterable[str] | None = None*, *extra_check: Callable[[Any], bool] | None = None*, *doc: str = ''*, *fold_subdoc: bool = False*, *extra_check_errmsg: str = ''*)

Bases: object

Define possible arguments and their types and properties.

Each *Argument* instance contains a *name* and a *dtype*, that correspond to the key and value type of the actual argument. Additionally, it can include *sub_fields* and *sub_variants* to deal with nested dict arguments.

Parameters

name
[str] The name of the current argument, i.e. the key in the arg dict.

dtype
[type or list of type] The value type of the current argument, can be a list of possible types. *None* will be treated as *NoneType*

sub_fields: list of Argument, optional
If given, *dtype* is assumed to be dict, whose items correspond to the *Argument`s in the `sub_fields* list.

sub_variants: list of Variants, optional
If given, *dtype* is assumed to be dict, and its items are determined by the `Variant`s in the given list and the value of their flag keys.

repeat: bool, optional
If true, *dtype* is assume to be list of dict and each dict consists of sub fields and sub variants described above. Defaults to false.

optional: bool, optional
If true, consider the current argument to be optional in checking.

default: any value type
The default value of the argument,used in normalization.

alias: list of str
Alternative names of the current argument, used in normalization.

extra_check: callable
Additional check to be done on the value of the argument. Should be a function that takes the value and returns whether it passes.

doc: str
The doc string of the argument, used in doc generation.

fold_subdoc: bool, optional
If true, no doc will be generated for sub args.

extra_check_errmsg
[str] The error message if extra_check fails

Examples

```
>>> ca = Argument("base", dict, [Argument("sub", int)])
>>> ca.check({"base": {"sub1": 1}})
>>> ca.check_value({"sub1": 1})
```

for more detailed examples, please check the unit tests.

Attributes

I

Methods

add_subfield(name, *args, **kwargs)	Add a sub field to the current Argument.
add_subvariant(flag_name, *args, **kwargs)	Add a sub variant to the current Argument.
check(argdict[, strict])	Check whether *argdict* meets the structure defined in self.
check_value(value[, strict])	Check the value without the leading key.
extend_subfields(sub_fields)	Add a list of sub fields to the current Argument.
extend_subvariants(sub_variants)	Add a list of sub variants to the current Argument.
gen_doc([path])	Generate doc string for the current Argument.
normalize(argdict[, inplace, do_default, ...])	Modify *argdict* so that it meets the Argument structure.
normalize_value(value[, inplace, ...])	Modify the value so that it meets the Argument structure.
set_dtype(dtype)	Change the dtype of the current Argument.
set_repeat([repeat])	Change the repeat attribute of the current Argument.

flatten_sub
gen_doc_body
gen_doc_head
gen_doc_path
traverse
traverse_value

property I

add_subfield(*name: str | Argument*, **args*, ***kwargs*) → *Argument*

Add a sub field to the current Argument.

add_subvariant(*flag_name: str | Variant*, **args*, ***kwargs*) → *Variant*

Add a sub variant to the current Argument.

check(*argdict: dict*, *strict: bool = False*)

Check whether *argdict* meets the structure defined in self.

Will recursively check nested dicts according to sub_fields and sub_variants. Raise an error if the check fails.

Parameters

argdict
[dict] The arg dict to be checked

strict
[bool, optional] If true, only keys defined in *Argument* are allowed.

check_value(*value: Any*, *strict: bool = False*)

Check the value without the leading key.

Same as *check({self.name: value})*. Raise an error if the check fails.

Parameters

value
[any value type] The value to be checked

strict
[bool, optional] If true, only keys defined in *Argument* are allowed.

extend_subfields(*sub_fields: Iterable[Argument] | None*)

Add a list of sub fields to the current Argument.

extend_subvariants(*sub_variants: Iterable[Variant] | None*)

Add a list of sub variants to the current Argument.

flatten_sub(*value: dict*, *path=None*) → Dict[str, *Argument*]

gen_doc(*path: List[str] | None = None*, ***kwargs*) → str

Generate doc string for the current Argument.

gen_doc_body(*path: List[str] | None = None*, ***kwargs*) → str

gen_doc_head(*path: List[str] | None = None*, ***kwargs*) → str

gen_doc_path(*path: List[str] | None = None*, ***kwargs*) → str

normalize(*argdict: dict*, *inplace: bool = False*, *do_default: bool = True*, *do_alias: bool = True*, *trim_pattern: str | None = None*)

Modify *argdict* so that it meets the Argument structure.

Normalization can add default values to optional args, substitute alias by its standard names, and discard unnecessary args following given pattern.

Parameters

argdict
[dict] The arg dict to be normalized.

inplace
[bool, optional] If true, modify the given dict. Otherwise return a new one.

do_default
[bool, optional] Whether to add default values.

do_alias
[bool, optional] Whether to transform alias names.

trim_pattern
[str, optional] If given, discard keys that matches the glob pattern.

Returns

dict:
The normalized arg dict.

`normalize_value`(*value: Any*, *inplace: bool = False*, *do_default: bool = True*, *do_alias: bool = True*, *trim_pattern: str | None = None*)

Modify the value so that it meets the Argument structure.

Same as *normalize({self.name: value})[self.name]*.

Parameters

value
[any value type] The arg value to be normalized.

inplace
[bool, optional] If true, modify the given dict. Otherwise return a new one.

do_default
[bool, optional] Whether to add default values.

do_alias
[bool, optional] Whether to transform alias names.

trim_pattern
[str, optional] If given, discard keys that matches the glob pattern.

Returns

value:
The normalized arg value.

`set_dtype`(*dtype: None | type | Iterable[type]*)

Change the dtype of the current Argument.

`set_repeat`(*repeat: bool = True*)

Change the repeat attribute of the current Argument.

`traverse`(*argdict: dict, key_hook: ~typing.Callable[[~dargs.dargs.Argument, dict, ~typing.List[str]], None] = <function _DUMMYHOOK>, value_hook: ~typing.Callable[[~dargs.dargs.Argument, ~typing.Any, ~typing.List[str]], None] = <function _DUMMYHOOK>, sub_hook: ~typing.Callable[[~dargs.dargs.Argument, dict, ~typing.List[str]], None] = <function _DUMMYHOOK>, variant_hook: ~typing.Callable[[~dargs.dargs.Variant, dict, ~typing.List[str]], None] = <function _DUMMYHOOK>, path: ~typing.List[str] | None = None*)

traverse_value(*value: ~typing.Any, key_hook: ~typing.Callable[[~dargs.dargs.Argument, dict, ~typing.List[str]], None] = <function _DUMMYHOOK>, value_hook: ~typing.Callable[[~dargs.dargs.Argument, ~typing.Any, ~typing.List[str]], None] = <function _DUMMYHOOK>, sub_hook: ~typing.Callable[[~dargs.dargs.Argument, dict, ~typing.List[str]], None] = <function _DUMMYHOOK>, variant_hook: ~typing.Callable[[~dargs.dargs.Variant, dict, ~typing.List[str]], None] = <function _DUMMYHOOK>, path: ~typing.List[str] | None = None*)

class dargs.**ArgumentEncoder**(**, skipkeys=False, ensure_ascii=True, check_circular=True, allow_nan=True, sort_keys=False, indent=None, separators=None, default=None*)

Bases: JSONEncoder

Extended JSON Encoder to encode Argument object.

Examples

```
>>> json.dumps(some_arg, cls=ArgumentEncoder)
```

Methods

default(obj)	Generate a dict containing argument information, making it ready to be encoded to JSON string.
encode(o)	Return a JSON string representation of a Python data structure.
iterencode(o[, _one_shot])	Encode the given object and yield each string representation as available.

default(*obj*) → Dict[str, str | bool | List]

Generate a dict containing argument information, making it ready to be encoded to JSON string.

Returns

dict: Dict
a dict containing argument information

Notes

All object in the dict should be JSON serializable.

class dargs.**Variant**(*flag_name: str, choices: Iterable[*Argument*] | None = None, optional: bool = False, default_tag: str = '', doc: str = ''*)

Bases: object

Define multiple choices of possible argument sets.

Each Variant contains a *flag_name* and a list of choices that are represented by *Argument`s. The choice is picked if its name matches the value of `flag_name* in the actual arguments. The actual arguments should then be a dict containing *flag_name* and sub fields of the picked choice.

Parameters

flag_name: str
The name of the key to be used as the switching flag.

choices: list of Argument
A list of possible choices. Each of them should be an *Argument*. The name of the *Argument* serves as the tag in the switching flag.

optional: bool, optional
If true, the flag_name can be optional and defaults to *defalut_flag*.

default_tag: str, optional
Needed if optional is true.

doc: str, optional
The doc string used in document generation.

Notes

This class should only be used in sub variants of the *Argument* class.

Methods

add_choice(tag[, _dtype])	Add a choice Argument to the current Variant.
extend_choices(choices)	Add a list of choice Arguments to the current Variant.
set_default(default_tag)	Change the default tag of the current Variant.

dummy_argument
flatten_sub
gen_doc
gen_doc_flag
get_choice

add_choice(*tag: str | ~dargs.dargs.Argument*, *_dtype: None | type | ~typing.Iterable[type] = <class 'dict'>*, **args*, ***kwargs*) → *Argument*

Add a choice Argument to the current Variant.

dummy_argument()

extend_choices(*choices: Iterable[*Argument*] | None*)

Add a list of choice Arguments to the current Variant.

flatten_sub(*argdict: dict*, *path=None*) → Dict[str, *Argument*]

gen_doc(*path: List[str] | None = None*, *showflag: bool = False*, ***kwargs*) → str

gen_doc_flag(*path: List[str] | None = None*, ***kwargs*) → str

get_choice(*argdict: dict*, *path=None*) → *Argument*

set_default(*default_tag: bool | str*)

Change the default tag of the current Variant.

5.1.1 Submodules

5.1.2 dargs.dargs module

Some (ocaml) pseudo-code here to show the intended type structure.

type args = {key: str; value: data; optional: bool; doc: str} list and data =

Arg of dtype
Node of args
Repeat of args
Variant of (str * args) list

In actual implementation, We flatten this structure into on tree-like class *Argument* with (optional) attribute *dtype*, *sub_fields*, *repeat* and *sub_variants* to mimic the union behavior in the type structure.

Due to the complexity of *Variant* structure, it is implemented into a separate class *Variant* so that multiple choices can be handled correctly. We also need to pay special attention to flat the keys of its choices.

class dargs.dargs.**Argument**(*name: str*, *dtype: None | type | Iterable[type]*, *sub_fields: Iterable[*Argument*] | None = None*, *sub_variants: Iterable[*Variant*] | None = None*, *repeat: bool = False*, *optional: bool = False*, *default: Any = _Flags.NONE*, *alias: Iterable[str] | None = None*, *extra_check: Callable[[Any], bool] | None = None*, *doc: str = ''*, *fold_subdoc: bool = False*, *extra_check_errmsg: str = ''*)

Bases: `object`

Define possible arguments and their types and properties.

Each *Argument* instance contains a *name* and a *dtype*, that correspond to the key and value type of the actual argument. Additionally, it can include *sub_fields* and *sub_variants* to deal with nested dict arguments.

Parameters

name
[str] The name of the current argument, i.e. the key in the arg dict.

dtype
[type or list of type] The value type of the current argument, can be a list of possible types. *None* will be treated as *NoneType*

sub_fields: list of Argument, optional
If given, *dtype* is assumed to be dict, whose items correspond to the *Argument`s in the `sub_fields* list.

sub_variants: list of Variants, optional
If given, *dtype* is assumed to be dict, and its items are determined by the `Variant`s in the given list and the value of their flag keys.

repeat: bool, optional
If true, *dtype* is assume to be list of dict and each dict consists of sub fields and sub variants described above. Defaults to false.

optional: bool, optional
If true, consider the current argument to be optional in checking.

default: any value type
The default value of the argument,used in normalization.

alias: list of str
Alternative names of the current argument, used in normalization.

extra_check: callable
Additional check to be done on the value of the argument. Should be a function that takes the value and returns whether it passes.

doc: str
The doc string of the argument, used in doc generation.

fold_subdoc: bool, optional
If true, no doc will be generated for sub args.

extra_check_errmsg
[str] The error message if extra_check fails

Examples

```
>>> ca = Argument("base", dict, [Argument("sub", int)])
>>> ca.check({"base": {"sub1": 1}})
>>> ca.check_value({"sub1": 1})
```

for more detailed examples, please check the unit tests.

Attributes

I

Methods

add_subfield(name, *args, **kwargs)	Add a sub field to the current Argument.
add_subvariant(flag_name, *args, **kwargs)	Add a sub variant to the current Argument.
check(argdict[, strict])	Check whether *argdict* meets the structure defined in self.
check_value(value[, strict])	Check the value without the leading key.
extend_subfields(sub_fields)	Add a list of sub fields to the current Argument.
extend_subvariants(sub_variants)	Add a list of sub variants to the current Argument.
gen_doc([path])	Generate doc string for the current Argument.
normalize(argdict[, inplace, do_default, ...])	Modify *argdict* so that it meets the Argument structure.
normalize_value(value[, inplace, ...])	Modify the value so that it meets the Argument structure.
set_dtype(dtype)	Change the dtype of the current Argument.
set_repeat([repeat])	Change the repeat attribute of the current Argument.

flatten_sub
gen_doc_body
gen_doc_head
gen_doc_path
traverse
traverse_value

property I

add_subfield(*name: str | Argument*, **args*, ***kwargs*) → *Argument*

Add a sub field to the current Argument.

add_subvariant(*flag_name: str | Variant*, **args*, ***kwargs*) → *Variant*

Add a sub variant to the current Argument.

check(*argdict: dict*, *strict: bool = False*)

Check whether *argdict* meets the structure defined in self.

Will recursively check nested dicts according to sub_fields and sub_variants. Raise an error if the check fails.

Parameters

argdict
[dict] The arg dict to be checked

strict
[bool, optional] If true, only keys defined in *Argument* are allowed.

check_value(*value: Any*, *strict: bool = False*)

Check the value without the leading key.

Same as *check({self.name: value})*. Raise an error if the check fails.

Parameters

value
[any value type] The value to be checked

strict
[bool, optional] If true, only keys defined in *Argument* are allowed.

extend_subfields(*sub_fields: Iterable[Argument] | None*)

Add a list of sub fields to the current Argument.

extend_subvariants(*sub_variants: Iterable[Variant] | None*)

Add a list of sub variants to the current Argument.

flatten_sub(*value: dict*, *path=None*) → Dict[str, *Argument*]

gen_doc(*path: List[str] | None = None*, ***kwargs*) → str

Generate doc string for the current Argument.

gen_doc_body(*path: List[str] | None = None*, ***kwargs*) → str

gen_doc_head(*path: List[str] | None = None*, ***kwargs*) → str

gen_doc_path(*path: List[str] | None = None*, ***kwargs*) → str

normalize(*argdict: dict*, *inplace: bool = False*, *do_default: bool = True*, *do_alias: bool = True*, *trim_pattern: str | None = None*)

Modify *argdict* so that it meets the Argument structure.

Normalization can add default values to optional args, substitute alias by its standard names, and discard unnecessary args following given pattern.

Parameters

argdict
[dict] The arg dict to be normalized.

inplace
[bool, optional] If true, modify the given dict. Otherwise return a new one.

do_default
[bool, optional] Whether to add default values.

do_alias
[bool, optional] Whether to transform alias names.

trim_pattern
[str, optional] If given, discard keys that matches the glob pattern.

Returns

dict:
The normalized arg dict.

`normalize_value`(*value: Any*, *inplace: bool = False*, *do_default: bool = True*, *do_alias: bool = True*, *trim_pattern: str | None = None*)

Modify the value so that it meets the Argument structure.

Same as *normalize({self.name: value})[self.name]*.

Parameters

value
[any value type] The arg value to be normalized.

inplace
[bool, optional] If true, modify the given dict. Otherwise return a new one.

do_default
[bool, optional] Whether to add default values.

do_alias
[bool, optional] Whether to transform alias names.

trim_pattern
[str, optional] If given, discard keys that matches the glob pattern.

Returns

value:
The normalized arg value.

`set_dtype`(*dtype: None | type | Iterable[type]*)

Change the dtype of the current Argument.

`set_repeat`(*repeat: bool = True*)

Change the repeat attribute of the current Argument.

`traverse`(*argdict: dict, key_hook: ~typing.Callable[[~dargs.dargs.Argument, dict, ~typing.List[str]], None] = <function _DUMMYHOOK>, value_hook: ~typing.Callable[[~dargs.dargs.Argument, ~typing.Any, ~typing.List[str]], None] = <function _DUMMYHOOK>, sub_hook: ~typing.Callable[[~dargs.dargs.Argument, dict, ~typing.List[str]], None] = <function _DUMMYHOOK>, variant_hook: ~typing.Callable[[~dargs.dargs.Variant, dict, ~typing.List[str]], None] = <function _DUMMYHOOK>, path: ~typing.List[str] | None = None*)

traverse_value(*value: ~typing.Any, key_hook: ~typing.Callable[[~dargs.dargs.Argument, dict, ~typing.List[str]], None] = <function _DUMMYHOOK>, value_hook: ~typing.Callable[[~dargs.dargs.Argument, ~typing.Any, ~typing.List[str]], None] = <function _DUMMYHOOK>, sub_hook: ~typing.Callable[[~dargs.dargs.Argument, dict, ~typing.List[str]], None] = <function _DUMMYHOOK>, variant_hook: ~typing.Callable[[~dargs.dargs.Variant, dict, ~typing.List[str]], None] = <function _DUMMYHOOK>, path: ~typing.List[str] | None = None*)

class dargs.dargs.**ArgumentEncoder**(**, skipkeys=False, ensure_ascii=True, check_circular=True, allow_nan=True, sort_keys=False, indent=None, separators=None, default=None*)

Bases: JSONEncoder

Extended JSON Encoder to encode Argument object.

Examples

```
>>> json.dumps(some_arg, cls=ArgumentEncoder)
```

Methods

default(obj)	Generate a dict containing argument information, making it ready to be encoded to JSON string.
encode(o)	Return a JSON string representation of a Python data structure.
iterencode(o[, _one_shot])	Encode the given object and yield each string representation as available.

default(*obj*) → Dict[str, str | bool | List]

Generate a dict containing argument information, making it ready to be encoded to JSON string.

Returns

dict: Dict
a dict containing argument information

Notes

All object in the dict should be JSON serializable.

exception dargs.dargs.**ArgumentError**(*path: None | str | List[str] = None, message: str | None = None*)

Bases: Exception

Base error class for invalid argument values in argchecking.

exception dargs.dargs.**ArgumentKeyError**(*path: None | str | List[str] = None, message: str | None = None*)

Bases: *ArgumentError*

Error class for missing or invalid argument keys.

exception dargs.dargs.**ArgumentTypeError**(*path: None | str | List[str] = None*, *message: str | None = None*)

Bases: *ArgumentError*

Error class for invalid argument data types.

exception dargs.dargs.**ArgumentValueError**(*path: None | str | List[str] = None*, *message: str | None = None*)

Bases: *ArgumentError*

Error class for missing or invalid argument values.

class dargs.dargs.**Variant**(*flag_name: str*, *choices: Iterable[*Argument*] | None = None*, *optional: bool = False*, *default_tag: str = ''*, *doc: str = ''*)

Bases: object

Define multiple choices of possible argument sets.

Each Variant contains a *flag_name* and a list of choices that are represented by *Argument`s. The choice is picked if its name matches the value of `flag_name* in the actual arguments. The actual arguments should then be a dict containing *flag_name* and sub fields of the picked choice.

Parameters

flag_name: str
The name of the key to be used as the switching flag.

choices: list of Argument
A list of possible choices. Each of them should be an *Argument*. The name of the *Argument* serves as the tag in the switching flag.

optional: bool, optional
If true, the flag_name can be optional and defaults to *defalut_flag*.

default_tag: str, optional
Needed if optional is true.

doc: str, optional
The doc string used in document generation.

Notes

This class should only be used in sub variants of the *Argument* class.

Methods

add_choice(tag[, _dtype])	Add a choice Argument to the current Variant.
extend_choices(choices)	Add a list of choice Arguments to the current Variant.
set_default(default_tag)	Change the default tag of the current Variant.

dummy_argument
flatten_sub
gen_doc
gen_doc_flag
get_choice

add_choice(*tag: str | ~dargs.dargs.Argument*, *_dtype: None | type | ~typing.Iterable[type] = <class 'dict'>*, **args*, ***kwargs*) → *Argument*

Add a choice Argument to the current Variant.

dummy_argument()

extend_choices(*choices: Iterable[*Argument*] | None*)

Add a list of choice Arguments to the current Variant.

flatten_sub(*argdict: dict*, *path=None*) → Dict[str, *Argument*]

gen_doc(*path: List[str] | None = None*, *showflag: bool = False*, ***kwargs*) → str

gen_doc_flag(*path: List[str] | None = None*, ***kwargs*) → str

get_choice(*argdict: dict*, *path=None*) → *Argument*

set_default(*default_tag: bool | str*)

Change the default tag of the current Variant.

dargs.dargs.**did_you_mean**(*choice: str*, *choices: List[str]*) → str

Get did you mean message.

Parameters

choice
[str] the user's wrong choice

choices
[list[str]] all the choices

Returns

str
did you mean error message

dargs.dargs.**isinstance_annotation**(*value*, *dtype*) → bool

Same as isinstance(), but supports arbitrary type annotations.

dargs.dargs.**make_ref_pair**(*path*, *text=None*, *prefix=None*)

dargs.dargs.**make_rst_refid**(*name*)

dargs.dargs.**trim_by_pattern**(*argdict: dict*, *pattern: str*, *reserved: List[str] | None = None*, *use_regex: bool = False*)

dargs.dargs.**update_nodup**(*this: dict*, **others: dict | Iterable[tuple]*, *exclude: Iterable | None = None*, *err_msg: str | None = None*)

5.1.3 dargs.notebook module

IPython/Jupyter Notebook display for dargs.

It is expected to be used in Jupyter Notebook, where the IPython module is available.

Examples

```
>>> from dargs.sphinx import _test_argument
>>> from dargs.notebook import JSON
>>> jstr = """
... {
...     "test_argument": "test1",
...     "test_variant": "test_variant_argument",
...     "_comment": "This is an example data"
... }
... """
>>> JSON(jstr, _test_argument())
```

dargs.notebook.**JSON**(*data: dict | str*, *arg:* Argument | *List[*Argument*]*)

Display JSON data with Argument in the Jupyter Notebook.

Parameters

data
[dict or str] The JSON data to be displayed, either JSON string or a dict.

arg
[dargs.Argument or list[dargs.Argument]] The Argument that describes the JSON data.

5.1.4 dargs.sphinx module

Sphinx extension.

To enable dargs Sphinx extension, add *dargs.sphinx* to the extensions of conf.py:

```
extensions = [
    'dargs.sphinx',
]
```

Then *dargs* directive will be added:

```
.. dargs::
   :module: dargs.sphinx
   :func: _test_argument
```

where *_test_argument* returns an *Argument*. A list of *Argument* is also accepted.

class dargs.sphinx.**DargsDirective**(*name*, *arguments*, *options*, *content*, *lineno*, *content_offset*, *block_text*, *state*, *state_machine*)

Bases: Directive

dargs directive.

Methods

add_name(node)	Append self.options['name'] to node['names'] if it exists.
assert_has_content()	Throw an ERROR-level DirectiveError if the directive doesn't have contents.
directive_error(level, message)	Return a DirectiveError suitable for being thrown as an exception.

debug
error
info
run
severe
warning

has_content: ClassVar[bool] = True

May the directive have content?

option_spec: ClassVar[dict] = {'func': <function unchanged>, 'module': <function unchanged>}

Mapping of option names to validator functions.

run()

class dargs.sphinx.**DargsDomain**(*env: BuildEnvironment*)

Bases: Domain

Dargs domain.

Includes: - dargs::argument directive - dargs::argument role

Methods

add_object_type(name, objtype)	Add an object type.
check_consistency()	Do consistency checks (**experimental**).
clear_doc(docname)	Remove traces of a document in the domain-specific inventories.
directive(name)	Return a directive adapter class that always gives the registered directive its full name ('domain:name') as self.name.
get_enumerable_node_type(node)	Get type of enumerable nodes (experimental).
get_full_qualified_name(node)	Return full qualified name for given node.
get_objects()	Return an iterable of "object descriptions".
get_type_name(type[, primary])	Return full name for given ObjType.
merge_domaindata(docnames, otherdata)	Merge in data regarding *docnames* from a different domaindata inventory (coming from a subprocess in parallel builds).
process_doc(env, docname, document)	Process a document after it is read by the environment.
process_field_xref(pnode)	Process a pending xref created in a doc field.
resolve_any_xref(env, fromdocname, builder, ...)	Resolve the pending_xref *node* with the given *target*.
resolve_xref(env, fromdocname, builder, typ, ...)	Resolve cross-references.
role(name)	Return a role adapter function that always gives the registered role its full name ('domain:name') as the first argument.
setup()	Set up domain object.

directives: ClassVar[dict] = {'argument': <class 'dargs.sphinx.DargsObject'>}

directive name -> directive class

initial_data: ClassVar[dict] = {'arguments': {}}

data value for a fresh environment

label: ClassVar[str] = 'dargs'

domain label: longer, more descriptive (used in messages)

name: ClassVar[str] = 'dargs'

domain name: should be short, but unique

object_types: ClassVar[dict] = {'argument': <sphinx.domains.ObjType object>}

type (usually directive) name -> ObjType instance

resolve_xref(*env*, *fromdocname*, *builder*, *typ*, *target*, *node*, *contnode*)

Resolve cross-references.

roles: ClassVar[dict] = {'argument': <sphinx.roles.XRefRole object>}

role name -> role callable

class dargs.sphinx.**DargsObject**(*name*, *arguments*, *options*, *content*, *lineno*, *content_offset*, *block_text*, *state*, *state_machine*)

Bases: ObjectDescription

dargs::argument directive.

This directive creates a signature node for an argument.

Attributes

config
Reference to the `Config` object.

domain
env
Reference to the `BuildEnvironment` object.

Methods

add_name(node)	Append self.options['name'] to node['names'] if it exists.
add_target_and_index(name, sig, signode)	Add cross-reference IDs and entries to self.indexnode, if applicable.
after_content()	Called after parsing content.
assert_has_content()	Throw an ERROR-level DirectiveError if the directive doesn't have contents.
before_content()	Called before parsing content.
directive_error(level, message)	Return a DirectiveError suitable for being thrown as an exception.
get_location()	Get current location info for logging.
get_signatures()	Retrieve the signatures to document from the directive arguments.
get_source_info()	Get source and line number.
handle_signature(sig, signode)	Parse the signature *sig* into individual nodes and append them to *signode*.
run()	Main directive entry function, called by docutils upon encountering the directive.
set_source_info(node)	Set source and line number to the node.
transform_content(contentnode)	Called after creating the content through nested parsing, but before the `object-description-transform` event is emitted, and before the info-fields are transformed.

debug
error
get_field_type_map
info
severe
warning

add_target_and_index(*name*, *sig*, *signode*)

Add cross-reference IDs and entries to self.indexnode, if applicable.

name is whatever *handle_signature()* returned.

handle_signature(*sig*, *signode*)

Parse the signature *sig* into individual nodes and append them to *signode*. If ValueError is raised, parsing is aborted and the whole *sig* is put into a single desc_name node.

The return value should be a value that identifies the object. It is passed to *add_target_and_index()* unchanged, and otherwise only used to skip duplicates.

option_spec: ClassVar[dict] = {'path': <function unchanged>}

Mapping of option names to validator functions.

dargs.sphinx.**setup**(*app*)

Setup sphinx app.

CHAPTER

SIX

AUTHORS

- A bot of @njzjz
- Han Wang
- Jinzhe Zeng
- Yixiao Chen
- dependabot[bot]
- pre-commit-ci[bot]

CHAPTER

SEVEN

INDICES AND TABLES

- genindex
- modindex
- search

PYTHON MODULE INDEX

d

INDEX