paramspace Documentation

Release 2.5.7

Yunus Sevinchan

Aug 09, 2021

YAML TOOLS

1 Supported YAML Tags 3

2 paramspace package 7

Python Module Index 31

Index 33

This is the documentation of the `paramspace` package.

It currently is very much *Work in Progress*; in its current state, it only contains an API reference.

In the meantime, refer to the README on the project page for installation instructions, example usage, and development information.

Note: If you find any errors in this documentation or would like to contribute to the project, a visit to the project page is appreciated.

CHAPTER

ONE

SUPPORTED YAML TAGS

YAML allows defining custom so-called tags which can be distinguished during loading and serialization of objects. *paramspace* makes heavy use of this possibility, as it greatly simplifies the definition and usage of configuration files.

1.1 paramspace-related tags

The *paramspace.yaml* module implements constructors and representers for the following classes:

- `!pspace` constructs a *ParamSpace*
- `!pdim` constructs a *ParamDim*
- `!coupled-pdim` constructs a *CoupledParamDim*

This is a very convenient way of defining these objects.

Hint: For the *ParamDim* and derived classes, there additionally are the `!pdim-default` and `!coupled-pdim-default` tags. These do not create a *ParamDim* objects but directly return the default value. By adding the `-default` in the end, they can be quickly deactivated inside the configuration file (as an alternative to commenting them out).

1.2 Python builtins and basic operators

paramspace.yaml adds YAML constructors for a number of frequently used Python built-in functions and operators. Having these available while specifying configurations can make the definition of configurations files more versatile.

Warning: The YAML tags provided here are only meant to allow basic operations, i.e. summing two parameters to create a third. Don't overdo it. Configuration files should remain easy to read.

The tags shown below call the equivalent Python builtin or the operators defined in the `operator` Python module. Example:

```
any:      !any        [false, 0, true]    # == True
all:      !all        [true, 5, 0]        # == False

abs:      !abs        -1          # +1
int:      !int        1.23        # 1
```

(continues on next page)

(continued from previous page)

```yaml
round:     !round      9.87         # 10
sum:       !sum        [1, 2, 3]    # 6
prod:      !prod       [2, 3, 4]    # 24

min:       !min        [1, 2, 3]    # 1
max:       !max        [1, 2, 3]    # 3

sorted:    !sorted     [2, 1, 3]    # [1, 2, 3]
isorted:   !isorted    [2, 1, 3]    # [3, 2, 1]

# Operators
add:       !add        [1, 2]       # 1 + 2
sub:       !sub        [2, 1]       # 2 - 1
mul:       !mul        [3, 4]       # 3 * 4
mod:       !mod        [3, 2]       # 3 % 2
pow:       !pow        [2, 4]       # 2 ** 4
truediv:   !truediv    [3, 2]       # 3 / 2
floordiv:  !floordiv   [3, 2]       # 3 // 2
pow_mod:   !pow        [2, 4, 3]    # 2 ** 4 % 3

not:       !not        [true]
and:       !and        [true, false]
or:        !or         [true, false]
xor:       !xor        [true, true]

lt:        !lt         [1, 2]       # 1 <  2
le:        !le         [2, 2]       # 2 <= 2
eq:        !eq         [3, 3]       # 3 == 3
ne:        !ne         [3, 1]       # 3 != 1
ge:        !ge         [2, 2]       # 2 >= 2
gt:        !gt         [4, 3]       # 4 >  3

negate:    !negate     [1]              # -1
invert:    !invert     [true]           # ~true
contains:  !contains   [[1,2,3], 4]     # 4 in [1,2,3] == False

concat:    !concat     [[1,2,3], [4,5], [6,7,8]]  # [...]+[...]+[...]+...

# List generation
# ... using the paramspace.tools.create_indices function
list1:     !listgen    [0, 10, 2]    # [0, 2, 4, 6, 8]
list2:     !listgen
  from_range: [0, 10, 3]
  unique: true
  append: [100]
  remove: [0]
  sort: true

# ... using np.linspace, np.logspace, np.arange
lin:       !linspace   [-1, 1, 5]    # [-1., -.5, 0., .5, 1.]
log:       !logspace   [1, 4, 4]     # [10., 100., 1000., 10000.]
arange:    !arange     [0, 1, .2]    # [0., .2, .4, .6, .8]
```

(continues on next page)

(continued from previous page)

```
# String formatting
format1:  !format     ["{} is not {}", foo, bar]
format2:  !format
  fstr: "{some_key:}: {some_value:}"
  some_key: fish
  some_value: spam
format3:  !format
  fstr: "results: {stats[mean]:.2f} ± {stats[std]:.2f}"
  stats:
    mean: 1.632
    std:  0.026
```

1.3 Recursively updating maps

While YAML already provides the << operator to update a mapping, this operator does not work recursively. The `!rec-update` YAML tag supplies exactly that functionality using the *recursive_update()* function.

```
some_map: &some_map
  foo: bar
  spam: fish
some_other_map: &some_other_map
  foo:
    bar: baz
    baz: bar
  fish: spam

# Create a new map by recursively updating the first map with
# the second one (uses deep copies to avoid side effects)
merged: !rec-update [<<: *some_map, <<: *some_other_map]
# NOTE: Need to use  ^^-- inheritance here, otherwise this will
#       result in empty mappings (for some reason)
```

Warning: Always include via `<<: *my_ref`!

If supplying references to mappings (as shown in the example), the references **have** to be included using `<<: *my_ref`!

Otherwise, if using the simple `*my_ref` as argument, the YAML parser does not properly resolve the reference to the anchor but only returns an empty mapping.

CHAPTER

TWO

PARAMSPACE PACKAGE

This package provides classes to conveniently define hierarchically structured parameter spaces and iterate over them.

To that end, any dict-like object can be populated with `ParamDim` objects to create a parameter dimension at that key. When creating a `ParamSpace` from this dict, it becomes possible to iterate over all points in the space created by the parameter dimensions, i.e. the *parameter space*.

Furthermore, the `paramspace.yaml` module provides possibilities to define the parameter space fully from YAML configuration files, using custom YAML tags.

2.1 Submodules

2.1.1 paramspace.paramdim module

The ParamDim classes define parameter dimensions along which discrete values can be assumed. While they provide iteration abilities on their own, they make sense mostly to use as objects in a dict that is converted to a ParamSpace.

class paramspace.paramdim.**Masked**(*value*)

Bases: `object`

To indicate a masked value in a ParamDim

__init__(*value*)

Initialize a Masked object that is a placeholder for the given value

Parameters **value** – The value to mask

property value

classmethod to_yaml(*representer*, *node:* paramspace.paramdim.Masked)

Parameters

- **representer** (*ruamel.yaml.representer*) – The representer module
- **node** (`Masked`) – The node, i.e. an instance of this class

Returns the scalar value that this object masks

exception paramspace.paramdim.**MaskedValueError**

Bases: `ValueError`

Raised when trying to set the state of a ParamDim to a masked value

args

with_traceback()
Exception.with_traceback(tb) – set self.__traceback__ to tb and return self.

class paramspace.paramdim.**ParamDimBase**(*, *default*, *values: Optional[Iterable] = None*, *order: Optional[float] = None*, *name: Optional[str] = None*, *as_type: Optional[str] = None*, *assert_unique: bool = True*, ***kwargs*)

Bases: object

The ParamDim base class.

_OMIT_ATTR_IN_EQ = ()

_REPR_ATTRS = ()

_VKWARGS = ('values', 'range', 'linspace', 'logspace')

__init__(*, *default*, *values: Optional[Iterable] = None*, *order: Optional[float] = None*, *name: Optional[str] = None*, *as_type: Optional[str] = None*, *assert_unique: bool = True*, ***kwargs*) → None
Initialise a parameter dimension object.

Parameters

- **default** – default value of this parameter dimension
- **values** (*Iterable, optional*) – Which discrete values this parameter dimension can take. This argument takes precedence over any constructors given in the kwargs (like range, linspace, ...).
- **order** (*float, optional*) – If given, this allows to specify an order within a ParamSpace that includes this ParamDim object. If not, will use np.inf instead.
- **name** (*str, optional*) – If given, this is an *additional* name of this ParamDim object, and can be used by the ParamSpace to access this object.
- **as_type** (*str, optional*) – If given, casts the individual created values to a certain python type. The following string values are possible: str, int, bool, float
- **assert_unique** (*bool, optional*) – Whether to assert uniqueness of the values among them.
- ****kwargs** – Constructors for the *values* argument, valid keys are *range*, *linspace*, and *logspace*; corresponding values are expected to be iterables and are passed to *range(*args)*, *np.linspace(*args)*, or *np.logspace(*args)*, respectively.

Raises **TypeError** – For invalid arguments

_init_vals(*, *as_type: str*, *assert_unique: bool*, ***kwargs*)
Parses the arguments and invokes _set_vals

property name
The name value.

property order
The order value.

property default
The default value.

property values: tuple
The values that are iterated over.

Returns

the values this parameter dimension can take. If None, the values are not yet set.

Return type tuple

property coords: tuple

Returns the coordinates of this parameter dimension, i.e., the combined default value and the sequence of iteration values.

Returns coordinates associated with the indices of this dimension

Return type tuple

property pure_coords: tuple

Returns the pure coordinates of this parameter dimension, i.e., the combined default value and the sequence of iteration values, but with masked values resolved.

Returns coordinates associated with the indices of this dimension

Return type tuple

property num_values: int

The number of values available.

Returns The number of available values

Return type int

property num_states: int

The number of possible states, i.e., including the default state

Returns The number of possible states

Return type int

property state: int

The current iterator state

Returns

The state of the iterator; if it is None, the ParamDim is not inside an iteration.

Return type Union[int, None]

property current_value

If in an iteration, returns the value according to the current state. Otherwise, returns the default value.

__eq__(*other*) → bool

Check for equality between self and other

Parameters **other** – the object to compare to

Returns Whether the two objects are equivalent

Return type bool

abstract __len__() → int

Returns the effective length of the parameter dimension, i.e. the number of values that will be iterated over

Returns The number of values to be iterated over

Return type int

__str__() → str

Returns

Returns the string representation of the ParamDimBase-derived object

Return type str

`__repr__()` → str

Returns

Returns the string representation of the ParamDimBase-derived object

Return type str

`_parse_repr_attrs()` → dict

For the __repr__ method, collects some attributes into a dict

`__iter__()`

Iterate over available values

`__next__()`

Move to the next valid state and return the corresponding parameter value.

Returns The current value (inside an iteration)

`abstract enter_iteration()` → None

Sets the state to the first possible one, symbolising that an iteration has started.

Returns None

Raises `StopIteration` – If no iteration is possible

`abstract iterate_state()` → None

Iterates the state of the parameter dimension.

Returns None

Raises `StopIteration` – Upon end of iteration

`abstract reset()` → None

Called after the end of an iteration and should reset the object to a state where it is possible to start another iteration over it.

Returns None

`_parse_value(`*val*, *, *as_type: Optional[str] = None*`)`

Parses a single value and ensures it is of correct type.

`_set_values(`*values: Iterable*, *, *assert_unique: bool*, *as_type: Optional[str] = None*`)`

This function sets the values attribute; it is needed for the values setter function that is overwritten when changing the property in a derived class.

Parameters

- `values` (*Iterable*) – The iterable to set the values with
- `assert_unique` (*bool*) – Whether to assert uniqueness of the values
- `as_type` (*str, optional*) – The following values are possible: str, int, bool, float. If not given, will leave the values as they are.

Raises

- `AttributeError` – If the attribute is already set
- `ValueError` – If the iterator is invalid

Deleted Parameters:

as_float (bool, optional): If given, makes sure that values are of type float; this is needed for the numpy initializers

_rec_tuple_conv(*obj: list*)
Recursively converts a list-like object into a tuple, replacing all occurences of lists with tuples.

_YAML_UPDATE = {}

_YAML_REMOVE_IF = {'name': (None,), 'order': (None,)}

classmethod to_yaml(*representer*, *node*)

Parameters

- **representer** (*ruamel.yaml.representer*) – The representer module
- **node** (*type(self)*) – The node, i.e. an instance of this class

Returns a yaml mapping that is able to recreate this object

classmethod from_yaml(*constructor*, *node*)
The default constructor for ParamDim-derived objects

_abc_impl = <_abc_data object>

class paramspace.paramdim.**ParamDim**(**, mask: Union[bool, Tuple[bool]] = False, **kwargs*)
Bases: *paramspace.paramdim.ParamDimBase*

The ParamDim class.

_OMIT_ATTR_IN_EQ = ('_mask_cache', '_inside_iter', '_target_of')

_REPR_ATTRS = ('mask',)

yaml_tag = '!pdim'

_YAML_UPDATE = {'mask': 'mask'}

_YAML_REMOVE_IF = {'mask': (None, False), 'name': (None,), 'order': (None,)}

__init__(**, mask: Union[bool, Tuple[bool]] = False, **kwargs*)
Initialize a regular parameter dimension.

Parameters

- **mask** (*Union[bool, Tuple[bool]], optional*) – Which values of the dimension to mask, i.e., skip in iteration. Note that masked values still count to the length of the parameter dimension!
- ****kwargs** – Passed to `ParamDimBase.__init__`. Possible arguments:
  - default: default value of this parameter dimension
  - **values (Iterable, optional): Which discrete values this** parameter dimension can take. This argument takes precedence over any constructors given in the kwargs (like range, linspace, …).
  - **order (float, optional): If given, this allows to specify an** order within a ParamSpace that includes this ParamDim. If not given, np.inf will be used, i.e., dimension is last.
  - **name (str, optional): If given, this is an *additional* name** of this ParamDim object, and can be used by the ParamSpace to access this object.
  - ****kwargs: Constructors for the values argument, valid** keys are `range`, `linspace`, and `logspace`; corresponding values are expected to be iterables and are passed to `range(*args)`, `np.linspace(*args)`, or `np.logspace(*args)`, respectively.

property target_of
Returns the list that holds all the CoupledParamDim objects that point to this instance of ParamDim.

property state: int
The current iterator state

Returns

The state of the iterator; if it is None, the ParamDim is not inside an iteration.

Return type Union[int, None]

property mask_tuple: Tuple[bool]
Returns a tuple representation of the current mask

property mask: Union[bool, Tuple[bool]]
Returns False if no value is masked or a tuple of booleans that represents the mask

property num_masked: int
Returns the number of unmasked values

__len__() → int
Returns the effective length of the parameter dimension, i.e. the number of values that will be iterated over.

Returns The number of values to be iterated over

Return type int

enter_iteration() → None
Sets the state to the first possible one, symbolising that an iteration has started.

Raises **StopIteration** – If no iteration is possible because all values are masked.

iterate_state() → None
Iterates the state of the parameter dimension.

Raises **StopIteration** – Upon end of iteration

reset() → None
Called after the end of an iteration and should reset the object to a state where it is possible to start another iteration over it.

Returns None

_VKWARGS = ('values', 'range', 'linspace', 'logspace')

__eq__(*other*) → bool
Check for equality between self and other

Parameters **other** – the object to compare to

Returns Whether the two objects are equivalent

Return type bool

__iter__()
Iterate over available values

__next__()
Move to the next valid state and return the corresponding parameter value.

Returns The current value (inside an iteration)

__repr__() → str

Returns

Returns the string representation of the ParamDimBase-derived object

Return type str

`__str__()` → str

Returns

Returns the string representation of the ParamDimBase-derived object

Return type str

`_abc_impl = <_abc_data object>`

`_init_vals`(*, *as_type: str*, *assert_unique: bool*, ***kwargs*)
Parses the arguments and invokes `_set_vals`

`_parse_repr_attrs()` → dict
For the __repr__ method, collects some attributes into a dict

`_parse_value`(*val*, *, *as_type: Optional[str] = None*)
Parses a single value and ensures it is of correct type.

`_rec_tuple_conv`(*obj: list*)
Recursively converts a list-like object into a tuple, replacing all occurences of lists with tuples.

`_set_values`(*values: Iterable*, *, *assert_unique: bool*, *as_type: Optional[str] = None*)
This function sets the values attribute; it is needed for the values setter function that is overwritten when changing the property in a derived class.

Parameters

- **values** (*Iterable*) – The iterable to set the values with
- **assert_unique** (*bool*) – Whether to assert uniqueness of the values
- **as_type** (*str, optional*) – The following values are possible: str, int, bool, float. If not given, will leave the values as they are.

Raises

- **AttributeError** – If the attribute is already set
- **ValueError** – If the iterator is invalid

Deleted Parameters:

as_float (bool, optional): If given, makes sure that values are of type float; this is needed for the numpy initializers

`property coords: tuple`
Returns the coordinates of this parameter dimension, i.e., the combined default value and the sequence of iteration values.

Returns coordinates associated with the indices of this dimension

Return type tuple

`property current_value`
If in an iteration, returns the value according to the current state. Otherwise, returns the default value.

`property default`
The default value.

classmethod from_yaml(*constructor*, *node*)
The default constructor for ParamDim-derived objects

property name
The name value.

property num_states: int
The number of possible states, i.e., including the default state

Returns The number of possible states

Return type int

property num_values: int
The number of values available.

Returns The number of available values

Return type int

property order
The order value.

property pure_coords: tuple
Returns the pure coordinates of this parameter dimension, i.e., the combined default value and the sequence of iteration values, but with masked values resolved.

Returns coordinates associated with the indices of this dimension

Return type tuple

classmethod to_yaml(*representer*, *node*)

Parameters

- **representer** (*ruamel.yaml.representer*) – The representer module
- **node** (*type(self)*) – The node, i.e. an instance of this class

Returns a yaml mapping that is able to recreate this object

property values: tuple
The values that are iterated over.

Returns

the values this parameter dimension can take. If None, the values are not yet set.

Return type tuple

class paramspace.paramdim.**CoupledParamDim**(**, default=None, target_pdim: Optional[paramspace.paramdim.ParamDim] = None, target_name: Optional[Union[str, Sequence[str]]] = None, use_coupled_default: Optional[bool] = None, use_coupled_values: Optional[bool] = None, **kwargs*)

Bases: *paramspace.paramdim.ParamDimBase*

A CoupledParamDim object is recognized by the ParamSpace and its state moves alongside with another ParamDim's state.

_OMIT_ATTR_IN_EQ = ()

_REPR_ATTRS = ('target_pdim', 'target_name', '_use_coupled_default', '_use_coupled_values')

yaml_tag = '!coupled-pdim'

_YAML_UPDATE = {'target_name': '_target_name_as_list'}

_YAML_REMOVE_IF = {'assert_unique': (True, False), 'default': (None,), 'name': (None,), 'order': (None,), 'target_name': (None,), 'target_pdim': (None,), 'use_coupled_default': (None,), 'use_coupled_values': (None,), 'values': (None, [None])}

__init__(*, *default=None*, *target_pdim: Optional[*paramspace.paramdim.ParamDim*] = None*, *target_name: Optional[Union[str, Sequence[str]]] = None*, *use_coupled_default: Optional[bool] = None*, *use_coupled_values: Optional[bool] = None*, ***kwargs*)

Initialize a coupled parameter dimension.

If the *default* or any values-setting argument is set, those will be used. If that is not the case, the respective parts from the coupled dimension will be used.

Parameters

- **default** (*None, optional*) – The default value. If not given, will use the one from the coupled object.
- **target_pdim** (*ParamDim, optional*) – The ParamDim object to couple to
- **target_name** (*Union[str, Sequence[str]], optional*) – The *name* of the ParamDim object to couple to; needs to be within the same ParamSpace and the ParamSpace needs to be able to resolve it using this name.
- **use_coupled_default** (*bool, optional*) – DEPRECATED
- **use_coupled_values** (*bool, optional*) – DEPRECATED
- ****kwargs** – Passed to ParamDimBase.__init__

Raises **TypeError** – If neither target_pdim nor target_name were given or or both were given

__len__() → int

Returns the effective length of the parameter dimension, i.e. the number of values that will be iterated over; corresponds to that of the target ParamDim

Returns The number of values to be iterated over

Return type int

enter_iteration() → None

Does nothing, as state has no effect for CoupledParamDim

iterate_state() → None

Does nothing, as state has no effect for CoupledParamDim

reset() → None

Does nothing, as state has no effect for CoupledParamDim

property target_name: Union[str, Sequence[str]]

The ParamDim object this CoupledParamDim couples to.

property _target_name_as_list: Union[str, List[str]]

For the safe yaml representer, the target_name cannot be a tuple.

This property returns it as str or list of strings.

_VKWARGS = ('values', 'range', 'linspace', 'logspace')

__eq__(*other*) → bool

Check for equality between self and other

Parameters `other` – the object to compare to

Returns Whether the two objects are equivalent

Return type bool

`__iter__()`
Iterate over available values

`__next__()`
Move to the next valid state and return the corresponding parameter value.

Returns The current value (inside an iteration)

`__repr__()` → str

Returns

Returns the string representation of the ParamDimBase-derived object

Return type str

`__str__()` → str

Returns

Returns the string representation of the ParamDimBase-derived object

Return type str

`_abc_impl = <_abc_data object>`

`_init_vals`(*, *as_type: str*, *assert_unique: bool*, ***kwargs*)
Parses the arguments and invokes `_set_vals`

`_parse_repr_attrs()` → dict
For the __repr__ method, collects some attributes into a dict

`_parse_value`(*val*, *, *as_type: Optional[str] = None*)
Parses a single value and ensures it is of correct type.

`_rec_tuple_conv`(*obj: list*)
Recursively converts a list-like object into a tuple, replacing all occurences of lists with tuples.

`_set_values`(*values: Iterable*, *, *assert_unique: bool*, *as_type: Optional[str] = None*)
This function sets the values attribute; it is needed for the values setter function that is overwritten when changing the property in a derived class.

Parameters

- `values` (*Iterable*) – The iterable to set the values with
- `assert_unique` (*bool*) – Whether to assert uniqueness of the values
- `as_type` (*str, optional*) – The following values are possible: str, int, bool, float. If not given, will leave the values as they are.

Raises

- `AttributeError` – If the attribute is already set
- `ValueError` – If the iterator is invalid

Deleted Parameters:

as_float (bool, optional): If given, makes sure that values are of type float; this is needed for the numpy initializers

property coords: tuple

Returns the coordinates of this parameter dimension, i.e., the combined default value and the sequence of iteration values.

Returns coordinates associated with the indices of this dimension

Return type tuple

classmethod from_yaml(*constructor*, *node*)

The default constructor for ParamDim-derived objects

property name

The name value.

property num_states: int

The number of possible states, i.e., including the default state

Returns The number of possible states

Return type int

property num_values: int

The number of values available.

Returns The number of available values

Return type int

property order

The order value.

property pure_coords: tuple

Returns the pure coordinates of this parameter dimension, i.e., the combined default value and the sequence of iteration values, but with masked values resolved.

Returns coordinates associated with the indices of this dimension

Return type tuple

classmethod to_yaml(*representer*, *node*)

Parameters

- **representer** (*ruamel.yaml.representer*) – The representer module
- **node** (*type(self)*) – The node, i.e. an instance of this class

Returns a yaml mapping that is able to recreate this object

property target_pdim: paramspace.paramdim.ParamDim

The ParamDim object this CoupledParamDim couples to.

property default

The default value.

Returns the default value this parameter dimension can take.

Raises **RuntimeError** – If no ParamDim was associated yet

property values: tuple

The values that are iterated over.

If self._use_coupled_values is set, will be those of the coupled pdim.

Returns The values of this CoupledParamDim or the target ParamDim

Return type tuple

property state: int

The current iterator state of the target ParamDim

Returns

The state of the iterator; if it is None, the ParamDim is not inside an iteration.

Return type Union[int, None]

property current_value

If in an iteration, returns the value according to the current state. Otherwise, returns the default value.

property mask: Union[bool, Tuple[bool]]

Return the coupled object's mask value

2.1.2 paramspace.paramspace module

Implementation of the ParamSpace class

class paramspace.paramspace.**ParamSpace**(*d: dict*)

Bases: object

The ParamSpace class holds dict-like data in which some entries are ParamDim objects. These objects each define one parameter dimension.

The ParamSpace class then allows to iterate over the space that is created by the parameter dimensions: at each point of the space (created by the cartesian product of all dimensions), one manifestation of the underlying dict-like data is returned.

yaml_tag = '!pspace'

__init__(*d: dict*)

Initialize a ParamSpace object from a given mapping or sequence.

Parameters d (*Union[MutableMapping, MutableSequence]*) – The mapping or sequence that will form the parameter space. It is crucial that this object is mutable.

_gather_paramdims()

Gathers ParamDim objects by recursively going through the dict

static _unique_dim_names(*kv_pairs: Sequence[Tuple]*) → List[Tuple[str, *paramspace.paramdim.ParamDim*]]

Given a sequence of key-value pairs, tries to create a unique string representation of the entries, such that it can be used as a unique mapping from names to parameter dimension objects.

Parameters kv_pairs (*Sequence[Tuple]*) – Pairs of (path, ParamDim), where the path is a Tuple of strings.

Returns The now unique (name, ParamDim) pairs

Return type List[Tuple[str, *ParamDim*]]

Raises ValueError – For invalid names, i.e.: failure to find a unique representation.

_get_dim(*name: Union[str, Tuple[str]]*) → *paramspace.paramdim.ParamDimBase*

Get the ParamDim object with the given name or location.

Note that coupled parameter dimensions cannot be accessed via this method.

Parameters **name** (*Union[str, Tuple[str]]*) – If a string, will look it up by that name, which has to match completely. If it is a tuple of strings, the location is looked up instead.

Returns the parameter dimension object

Return type *ParamDimBase*

Raises

- **KeyError** – If the ParamDim could not be found
- **ValueError** – If the parameter dimension name was ambiguous

property default: dict

Returns the dictionary with all parameter dimensions resolved to their default values.

If an object is Masked, it will resolve it.

property current_point: dict

Returns the dictionary with all parameter dimensions resolved to the values, depending on the point in parameter space at which the iteration is.

Note that unlike .default, this does not resolve the value if it is Masked.

property dims: Dict[str, paramspace.paramdim.ParamDim]

Returns the ParamDim objects of this ParamSpace. The keys of this dictionary are the unique names of the dimensions, created during initialization.

property dims_by_loc: Dict[Tuple[str], paramspace.paramdim.ParamDim]

Returns the ParamDim objects of this ParamSpace, keys being the paths to the objects in the dictionary.

property coupled_dims: Dict[str, paramspace.paramdim.CoupledParamDim]

Returns the CoupledParamDim objects of this ParamSpace. The keys of this dictionary are the unique names of the dimensions, created during initialization.

property coupled_dims_by_loc: Dict[Tuple[str], paramspace.paramdim.CoupledParamDim]

Returns the CoupledParamDim objects found in this ParamSpace, keys being the paths to the objects in the dictionary.

property coords: Dict[str, tuple]

Returns the coordinates of all parameter dimensions as dict. This does not include the coupled dimensions!

As the coordinates are merely collected from the parameter dimensions, they may include Masked objects.

Note that the coordinates are converted to lists to make interfacing with xarray.DataArray easier.

property pure_coords: Dict[str, tuple]

Returns the pure coordinates of all parameter dimensions as dict. This does not include the coupled dimensions!

Unlike the .coords property, the pure coordinates are cleaned of any Masked values.

Note that the coordinates are converted to lists to make interfacing with xarray.DataArray easier.

property current_coords: collections.OrderedDict

Returns the current coordinates of all parameter dimensions.

This is a shortcut for the get_dim_values method without arguments.

property num_dims: int

Returns the number of parameter space dimensions. Coupled dimensions are not counted here!

property num_coupled_dims: int

Returns the number of coupled parameter space dimensions.

property volume: int
Returns the active volume of the parameter space, i.e. not counting coupled parameter dimensions or masked values

property full_volume: int
Returns the full volume, i.e. ignoring whether parameter dimensions are masked.

property shape: Tuple[int]
Returns the shape of the parameter space, not counting masked values of parameter dimensions. If a dimension is fully masked, it is still represented as of length 1, representing the default value being used.

Returns The iterator shape

Return type Tuple[int]

property full_shape: Tuple[int]
Returns the shape of the parameter space, ignoring masked values

Returns The shape of the fully unmasked iterator

Return type Tuple[int]

property states_shape: Tuple[int]
Returns the shape of the parameter space, including default states for each parameter dimension and ignoring masked ones.

Returns The shape tuple

Return type Tuple[int]

property max_state_no: int
Returns the highest possible state number

property state_vector: Tuple[int]
Returns a tuple of all current parameter dimension states

property state_no: Optional[int]
Returns the current state number by visiting the active parameter dimensions and querying their state numbers.

__eq__(*other*) → bool
Tests the equality of two ParamSpace objects.

__str__() → str
Returns a parsed, human-readable information string

__repr__() → str
Returns the raw string representation of the ParamSpace.

get_info_dict() → dict
Returns a dict with information about this ParamSpace object.

The returned dict contains similar information as *get_info_str()*. Furthermore, it uses only native data types (scalars, sequences, and mappings) such that it is easily serializable and usable in scenarios where the paramspace package is not available.

Note: This information is not meant to fully recreate the ParamSpace object, but merely to provide essential metadata like the volume or shape of the parameter space and the coordinates of each of its dimensions.

Raises **NotImplementedError** – If any of the parameter dimensions is masked.

get_info_str() → str
Returns a string that gives information about shape and size of this ParamSpace.

_parse_dims(*, *mode: str = 'names'*, *join_str: str = ' -> '*, *prefix: str = ' * '*) → str
Returns a multi-line string of dimension names or locations.

This function is intended mostly for internal representation, thus defaulting to the longer join strings.

classmethod to_yaml(*representer*, *node*)
In order to dump a ParamSpace as yaml, basically only the _dict attribute needs to be saved. It can be plugged into a constructor without any issues. However, to make the string representation a bit simpler, the OrderedDict is resolved to an unordered one.

Parameters

- **representer** (*ruamel.yaml.representer*) – The representer module
- **node** (*type(self)*) – The node, i.e. an instance of this class

Returns a yaml mapping that is able to recreate this object

classmethod from_yaml(*constructor*, *node*)
The default constructor for a ParamSpace object

get(*key*, *default=None*)
Returns a _copy_ of the item in the underlying dict

pop(*key*, *default=None*)
Pops an item from the underlying dict, if it is not a ParamDim

__iter__() → dict
Move to the next valid point in parameter space and return the corresponding dictionary.

Returns The current value of the iteration

Raises **StopIteration** – When the iteration has finished

iterator(*, *with_info: Optional[Union[str, Tuple[str]]] = None*, *omit_pt: bool = False*) → Generator[dict, None, None]
Returns an iterator (more precisely: a generator) yielding all unmasked points of the parameter space.

To control which information is returned at each point, the *with_info* and *omit_pt* arguments can be used. By default, the generator will return a single dictionary.

Note that an iteration is also possible for zero-volume parameter spaces, i.e. where no parameter dimensions were defined.

Parameters

- **with_info** (*Union[str, Tuple[str]], optional*) – Can pass strings here that are to be returned as the second value. Possible values are: 'state_no', 'state_vector', 'state_no_str', and 'current_coords'. To get multiple, add them to a tuple.
- **omit_pt** (*bool, optional*) – If true, the current value is omitted and only the information is returned.

Returns

yields point after point of the ParamSpace and the corresponding information

Return type Generator[dict, None, None]

reset() → None
Resets the paramter space and all of its dimensions to the initial state, i.e. where all states are None.

_next_state() → bool

Iterates the state of the parameter dimensions managed by this ParamSpace.

Important: this assumes that the parameter dimensions already have been prepared for an iteration and that self.state_no == 0.

Returns Returns False when iteration finishes

Return type bool

_gen_iter_rv(*pt*, *, *with_info: Sequence[str]*) → tuple

Is used during iteration to generate the iteration return value, adding additional information if specified.

Note that pt can also be None if iterate is a dry_run

property state_map: xarray.core.dataarray.DataArray

Returns an inverse mapping, i.e. an n-dimensional array where the indices along the dimensions relate to the states of the parameter dimensions and the content of the array relates to the state numbers.

Returns

A mapping of indices and coordinates to the state number. Note that it is not ensured that the coordinates are unique, so it _might_ not be possible to use location-based indexing.

Return type xr.DataArray

Raises RuntimeError – If – for an unknown reason – the iteration did not cover all of the state mapping. Should not occur.

property active_state_map: xarray.core.dataarray.DataArray

Returns a subset of the state map, where masked coordinates are removed and only the active coordinates are present.

Note that this array has to be re-calculated every time, as the mask status of the ParamDim objects is not controlled by the ParamSpace and can change without notice.

Also: the indices will no longer match the states of the dimensions! Values of the DataArray should only be accessed via the coordinates!

Returns

A reduced state map which only includes active, i.e.: unmasked coordinates.

Return type xr.DataArray

get_state_vector(*, *state_no: int*) → Tuple[int]

Returns the state vector that corresponds to a state number

Parameters **state_no** (*int*) – The state number to look for in the inverse mapping

Returns the state vector corresponding to the state number

Return type Tuple[int]

get_dim_values(*, *state_no: Optional[int] = None*, *state_vector: Optional[Tuple[int]] = None*) → collections.OrderedDict

Returns the current parameter dimension values or those of a certain state number or state vector.

_calc_state_no(*state_vector: Tuple[int]*) → int

set_mask(*name: Union[str, Tuple[str]]*, *mask: Union[bool, Tuple[bool]]*, *invert: bool = False*) → None

Set the mask value of the parameter dimension with the given name.

Parameters

- **name** (*Union[str, Tuple[str]]*) – the name of the dim, which can be a tuple of strings or a string. If name is a string, it will be converted to a tuple, regarding the '/' character as splitting string. The tuple is compared to the paths of the dimensions, starting from the back; thus, not the whole path needs to be given, it just needs to be enough to resolve the dimension names unambiguously. For names at the root level that could be ambiguous, a leading "/" in the string argument or an empty string in the tuple-form of the argument needs to be set to symbolise the dimension being at root level. Also, the ParamDim's custom name attribute can be used to identify it.
- **mask** (*Union[bool, Tuple[bool]]*) – The new mask values. Can also be a slice, the result of which defines the True values of the mask.
- **invert** (*bool, optional*) – If set, the mask will be inverted _after_ application.

set_masks(**mask_specs*) → None

Sets multiple mask specifications after another. Note that the order is maintained and that sequential specifications can apply to the same parameter dimensions.

Parameters ***mask_specs** – Can be tuples/lists or dicts which will be unpacked (in the given order) and passed to .set_mask

activate_subspace(**, *allow_default: bool = False*, *reset_all_others: bool = True*, ***selector*) → None

Selects a subspace of the parameter space and makes only that part active for iteration.

This is a wrapper around set_mask, implementing more arguments and also checking if any dimension is reduced to a default value, which might cause problems elsewhere.

Parameters

- **allow_default** (*bool, optional*) – If True, a ValueError is raised when any of the dimensions is completely masked or when the index 0 is used during selecting of a mask.
- **reset_all_others** (*bool, optional*) – If True, resets all masks before activating the subspace. If False, the previously applied masks are untouched.
- ****selector** – A dict specifying the *active* states. A key of the key-value pairs should be the name of the dimension, the value should be a dict with one of the following keys:
  - idx: to select by index
  - loc: to select by coordinate values
  - ****tol_kwargs: passed on to `np.isclose` when** comparing coordinate values.

  Non-sequence values will be put into lists. Alternatively, slices can be specified, which are applied on the list of all available indices or coordinates, respectively. As a shorthand, not specifying a dict but directly a list or a slice defaults to `loc`-behaviour.

Raises **ValueError** – Description

2.1.3 paramspace.tools module

This module provides general methods needed by the ParamSpan and ParamSpace classes.

`paramspace.tools.log`
The local logger instance

`paramspace.tools.SKIP`
A global *paramspace.tools.Skip* object to signify a Skip operation in the `recursive_*` functions. Not supported everywhere.

class `paramspace.tools.Skip`
Bases: `object`

A Skip object can be used to indiciate that no action should be taken.

It is used in the `recursive_**` functions like *paramspace.tools.recursive_update()* to indicate that a value is to be skipped.

`paramspace.tools.create_indices(*, from_range: Optional[list] = None, unique: bool = False, sort: bool = True, append: Optional[list] = None, remove: Optional[list] = None) → List[int]`
Generates a list of integer elements.

Parameters

- **from_range** (*list, optional*) – range arguments to use as the basis of the list
- **unique** (*bool, optional*) – Whether to ascertain uniqueness of elements
- **sort** (*bool, optional*) – Whether to sort the list before returning
- **append** (*list, optional*) – Additional elements to append to the list
- **remove** (*list, optional*) – Elements to remove all occurrences of

Returns The generated list

Return type List[int]

`paramspace.tools.recursive_contains(obj: Union[Mapping, Sequence], *, keys: Sequence) → bool`
Checks whether the given keysequence is reachable in the `obj`.

Parameters

- **obj** (*Union[Mapping, Sequence]*) – The object to check recursively
- **keys** (*Sequence*) – The sequence of keys to check for

Returns Whether the key sequence is reachable

Return type bool

`paramspace.tools.recursive_getitem(obj: Union[Mapping, Sequence], *, keys: Sequence)`
Go along the sequence of `keys` through `obj` and return the target item.

Parameters

- **obj** (*Union[Mapping, Sequence]*) – The object to get the item from
- **keys** (*Sequence*) – The sequence of keys to follow

Returns The target item from `obj`, specified by `keys`

Raises

- **IndexError** – If any index in the key sequence was not available

- **KeyError** – If any key in the key sequence was not available

`paramspace.tools.recursive_update(obj: Union[Mapping, List], upd: Union[Mapping, List], *, try_list_conversion: bool = False, no_convert: Sequence[type] = (<class 'str'>,)) → Union[Mapping, List]`

Recursively update items in `obj` with the values from `upd`.

Be aware that objects are not copied from `upd` to `obj`, but only assigned. This means:

- the given `obj` will be changed in place
- changing mutable elements in `obj` will also change them in *upd*

After the update, *obj* holds all entries of *upd* plus those that it did not have in common with *upd*.

If recursion is possible is determined by type; it is only done for types mappings (dicts) or lists.

To indicate that a value in a list should not be updated, an instance of the tools.Skip class, e.g. the tools.SKIP object, can be passed instead.

Parameters

- **obj** (*Union[Mapping, List]*) – The object to update.
- **upd** (*Union[Mapping, List]*) – The object to use for updating.
- **try_list_conversion** (*bool, optional*) – If true, it is tried to convert an entry in `obj` to a list if it is a list in `upd`
- **no_convert** (*Sequence[type], optional*) – For these types, conversion is skipped and an empty list is generated instead.

Returns The updated `obj`

Return type Union[Mapping, List]

`paramspace.tools.recursive_setitem(d: dict, *, keys: Tuple[str], val, create_key: bool = False)`

Recursively goes through dict-like `d` along the `keys` sequence in keys and sets the value to the child entry.

Parameters

- **d** (*dict*) – The dict-like object to invoke setitem on
- **keys** (*tuple*) – The key sequence pointing to the node to set the value of
- **val** – The value to set at `d[the][key][sequence]`
- **create_key** (*bool, optional*) – Whether to create the key if it does not already exist. Default: `False`.

Raises **KeyError** – On missing entry at `keys`.

`paramspace.tools.recursive_collect(obj: Union[Mapping, Sequence], *, select_func: Callable, prepend_info: Optional[Sequence] = None, info_func: Optional[Callable] = None, stop_recursion_types: Optional[Sequence[type]] = None, _parent_keys: Optional[tuple] = None) → list`

Go recursively through a mapping or sequence and collect selected elements.

The `select_func` is called on each value. If it returns `True`, that value will be collected to a list, which is returned at the end.

Additionally, some information can be gathered about these elements, controlled by `prepend_info`.

With `prepend_info`, information can be prepended to the return value. Then, not only the values but also these additional items can be gathered:

- `keys` : prepends the key
- `info_func` : prepends the return value of `info_func(val)`

The resulting return value is then a list of tuples (in that order).

Parameters

- **obj** (*Union[Mapping, Sequence]*) – The object to recursively search
- **select_func** (*Callable*) – Each element is passed to this function; if True is returned, the element is collected and search ends here.
- **prepend_info** (*Sequence, optional*) – If given, additional info about the selected elements can be gathered in two ways:
  1. By passing `keys`, the sequence of keys to get to this element is appended;
  2. by passing `info_func`, the `info_func` function is called on the argument and that value is added to the information tuple.
- **info_func** (*Callable, optional*) – The function used to prepend info
- **stop_recursion_types** (*Sequence[type], optional*) – Can specify types here that will not be further recursed through. NOTE that strings are never recursed through further.
- **_parent_keys** (*tuple, optional*) – Used to track the keys; not public!

Returns the collected elements, as selected by select_func(val) or – if `prepend_info` was set – tuples of `(info, element)`, where the requested information is in the first entries of the tuple

Return type list

Raises **ValueError** – Raised if invalid `prepend_info` entries were set

`paramspace.tools.`**`recursive_replace`**(*obj: Union[Mapping, Sequence]*, *, *select_func: Callable*, *replace_func: Callable*, *stop_recursion_types: Optional[Sequence[type]] = None*) → Union[Mapping, Sequence]

Go recursively through a mapping or sequence and call a replace function on each element that the select function returned true on.

For passing arguments to any of the two, use lambda functions.

Parameters

- **cont** (*Union[Mapping, Sequence]*) – The object to walk through recursively
- **select_func** (*Callable*) – The function that each value is passed to. If it returns `True`, the element will be replaced using the `replace_func`.
- **replace_func** (*Callable*) – Called if the `select_func` returned True. The return value replaces the existing object at the selected position inside `obj`.
- **stop_recursion_types** (*Sequence[type], optional*) – Can specify types here that will not be further recursed through. NOTE that strings are never recursed through further.

Returns The updated mapping where each element that was selected was replaced by the return value of the replacement function.

Return type Union[Mapping, Sequence]

`paramspace.tools.`**`is_iterable`**(*obj*) → bool

Whether the given object is iterable or not.

This is tested simply by invoking `iter(obj)` and returning `False` if this operation raises a TypeError.

Parameters **obj** – The object to test

Returns True if iterable, False else

Return type bool

`paramspace.tools.get_key_val_iter(obj: Union[Mapping, Sequence]) → Iterator`

Given an object – assumed dict- or sequence-like – returns a `(key, value)` iterator.

Parameters **obj** (*`Union[Mapping, Sequence]`*) – The object to generate the key-value iterator from

Returns An iterator that emits `(key, value)` tuples

Return type Iterator

2.1.4 paramspace.yaml module

This module registers various YAML constructors and representers, notably those for `ParamSpace` and `ParamDim`.

Furthermore, it defines a shared `ruamel.yaml.YAML` object that can be imported and used for loading and storing YAML files using the representers and constructors.

2.1.5 paramspace.yaml_constructors module

Defines the yaml constructors for the generation of `ParamSpace` and `ParamDim` during loading of YAML files.

Note that they are not registered in this module but in the `paramspace.yaml` module.

`paramspace.yaml_constructors.pspace(loader, node) → paramspace.paramspace.ParamSpace`

yaml constructor for creating a ParamSpace object from a mapping.

Suggested tag: `!pspace`

`paramspace.yaml_constructors.pspace_unsorted(loader, node) → paramspace.paramspace.ParamSpace`

yaml constructor for creating a ParamSpace object from a mapping.

Unlike the regular constructor, this one does NOT sort the input before instantiating ParamSpace.

Suggested tag: `!pspace-unsorted`

`paramspace.yaml_constructors.pdim(loader, node) → paramspace.paramdim.ParamDim`

constructor for creating a ParamDim object from a mapping

Suggested tag: `!pdim`

`paramspace.yaml_constructors.pdim_default(loader, node) → paramspace.paramdim.ParamDim`

constructor for creating a ParamDim object from a mapping, but only return the default value.

Suggested tag: `!pdim-default`

`paramspace.yaml_constructors.coupled_pdim(loader, node) → paramspace.paramdim.CoupledParamDim`

constructor for creating a CoupledParamDim object from a mapping

Suggested tag: `!coupled-pdim`

`paramspace.yaml_constructors.coupled_pdim_default(loader, node) → paramspace.paramdim.CoupledParamDim`

constructor for creating a CoupledParamDim object from a mapping, but only return the default value.

Suggested tag: `!coupled-pdim-default`

`paramspace.yaml_constructors._pspace_constructor(loader, node, sort_if_mapping: bool = True) → paramspace.paramspace.ParamSpace`

Constructor for instantiating ParamSpace from a mapping or a sequence

`paramspace.yaml_constructors.`**`_pdim_constructor`**(*loader*, *node*) → *paramspace.paramdim.ParamDim*
Constructor for creating a ParamDim object from a mapping

For it to be incorporated into a ParamSpace, one parent (or higher) of this node needs to be tagged such that the pspace_constructor is invoked.

`paramspace.yaml_constructors.`**`_coupled_pdim_constructor`**(*loader*, *node*) → *paramspace.paramdim.ParamDim*
Constructor for creating a ParamDim object from a mapping

For it to be incorporated into a ParamSpace, one parent (or higher) of this node needs to be tagged such that the pspace_constructor is invoked.

`paramspace.yaml_constructors.`**`_slice_constructor`**(*loader*, *node*)
Constructor for slices

`paramspace.yaml_constructors.`**`_range_constructor`**(*loader*, *node*)
Constructor for range

`paramspace.yaml_constructors.`**`_list_constructor`**(*loader*, *node*)
Constructor for lists, where node can be a mapping or sequence

`paramspace.yaml_constructors.`**`_func_constructor`**(*loader*, *node*, *, *func: Callable*, *unpack: bool = True*)
A constructor that constructs a scalar, mapping, or sequence from the given node and subsequently applies the given function on it.

Parameters

- **loader** – The selected YAML loader
- **node** – The node from which to construct a Python object
- **func** (*Callable*) – The callable to invoke on the resulting
- **unpack** (*bool, optional*) – Whether to unpack sequences or mappings into the `func` call

`paramspace.yaml_constructors.`**`recursively_sort_dict`**(*d: dict*) → collections.OrderedDict
Recursively sorts a dictionary by its keys, transforming it to an OrderedDict in the process.

From: http://stackoverflow.com/a/22721724/1827608

Parameters **d** (*dict*) – The dictionary to be sorted

Returns the recursively sorted dict

Return type OrderedDict

2.1.6 paramspace.yaml_representers module

This module implements custom YAML representer functions

`paramspace.yaml_representers.`**`_slice_representer`**(*representer*, *node: slice*)
Represents a Python slice object using the `!slice` YAML tag.

Parameters

- **representer** (*ruamel.yaml.representer*) – The representer module
- **node** (*slice*) – The node, i.e. a slice instance

Returns a yaml sequence that is able to recreate a slice

`paramspace.yaml_representers.`**`_range_representer`**(*representer*, *node: range*)
Represents a Python range object using the `!range` YAML tag.

Parameters

- **representer** (*ruamel.yaml.representer*) – The representer module
- **node** (*range*) – The node, i.e. a range instance

Returns a yaml sequence that is able to recreate a range

- genindex
- modindex

PYTHON MODULE INDEX

p

paramspace, 7
paramspace.paramdim, 7
paramspace.paramspace, 18
paramspace.tools, 24
paramspace.yaml, 27
paramspace.yaml_constructors, 27
paramspace.yaml_representers, 28

INDEX

Symbols

_OMIT_ATTR_IN_EQ (*paramspace.paramdim.CoupledParamDim attribute*), 14
_OMIT_ATTR_IN_EQ (*paramspace.paramdim.ParamDim attribute*), 11
_OMIT_ATTR_IN_EQ (*paramspace.paramdim.ParamDimBase attribute*), 8
_REPR_ATTRS (*paramspace.paramdim.CoupledParamDim attribute*), 14
_REPR_ATTRS (*paramspace.paramdim.ParamDim attribute*), 11
_REPR_ATTRS (*paramspace.paramdim.ParamDimBase attribute*), 8
_VKWARGS (*paramspace.paramdim.CoupledParamDim attribute*), 15
_VKWARGS (*paramspace.paramdim.ParamDim attribute*), 12
_VKWARGS (*paramspace.paramdim.ParamDimBase attribute*), 8
_YAML_REMOVE_IF (*paramspace.paramdim.CoupledParamDim attribute*), 15
_YAML_REMOVE_IF (*paramspace.paramdim.ParamDim attribute*), 11
_YAML_REMOVE_IF (*paramspace.paramdim.ParamDimBase attribute*), 11
_YAML_UPDATE (*paramspace.paramdim.CoupledParamDim attribute*), 15
_YAML_UPDATE (*paramspace.paramdim.ParamDim attribute*), 11
_YAML_UPDATE (*paramspace.paramdim.ParamDimBase attribute*), 11
__eq__() (*paramspace.paramdim.CoupledParamDim method*), 15
__eq__() (*paramspace.paramdim.ParamDim method*), 12
__eq__() (*paramspace.paramdim.ParamDimBase method*), 9
__eq__() (*paramspace.paramspace.ParamSpace method*), 20
__init__() (*paramspace.paramdim.CoupledParamDim method*), 15
__init__() (*paramspace.paramdim.Masked method*), 7
__init__() (*paramspace.paramdim.ParamDim method*), 11
__init__() (*paramspace.paramdim.ParamDimBase method*), 8
__init__() (*paramspace.paramspace.ParamSpace method*), 18
__iter__() (*paramspace.paramdim.CoupledParamDim method*), 16
__iter__() (*paramspace.paramdim.ParamDim method*), 12
__iter__() (*paramspace.paramdim.ParamDimBase method*), 10
__iter__() (*paramspace.paramspace.ParamSpace method*), 21
__len__() (*paramspace.paramdim.CoupledParamDim method*), 15
__len__() (*paramspace.paramdim.ParamDim method*), 12
__len__() (*paramspace.paramdim.ParamDimBase method*), 9
__next__() (*paramspace.paramdim.CoupledParamDim method*), 16
__next__() (*paramspace.paramdim.ParamDim method*), 12
__next__() (*paramspace.paramdim.ParamDimBase method*), 10
__repr__() (*paramspace.paramdim.CoupledParamDim method*), 16
__repr__() (*paramspace.paramdim.ParamDim method*), 12
__repr__() (*paramspace.paramdim.ParamDimBase method*), 9
__repr__() (*paramspace.paramspace.ParamSpace method*), 20
__str__() (*paramspace.paramdim.CoupledParamDim method*), 16
__str__() (*paramspace.paramdim.ParamDim method*), 13
__str__() (*paramspace.paramdim.ParamDimBase method*), 9
__str__() (*paramspace.paramspace.ParamSpace method*), 20

_abc_impl (*paramspace.paramdim.CoupledParamDim attribute*), 16
_abc_impl (*paramspace.paramdim.ParamDim attribute*), 13
_abc_impl (*paramspace.paramdim.ParamDimBase attribute*), 11
_calc_state_no() (*paramspace.paramspace.ParamSpace method*), 22
_coupled_pdim_constructor() (*in module paramspace.yaml_constructors*), 28
_func_constructor() (*in module paramspace.yaml_constructors*), 28
_gather_paramdims() (*paramspace.paramspace.ParamSpace method*), 18
_gen_iter_rv() (*paramspace.paramspace.ParamSpace method*), 22
_get_dim() (*paramspace.paramspace.ParamSpace method*), 18
_init_vals() (*paramspace.paramdim.CoupledParamDim method*), 16
_init_vals() (*paramspace.paramdim.ParamDim method*), 13
_init_vals() (*paramspace.paramdim.ParamDimBase method*), 8
_list_constructor() (*in module paramspace.yaml_constructors*), 28
_next_state() (*paramspace.paramspace.ParamSpace method*), 21
_parse_dims() (*paramspace.paramspace.ParamSpace method*), 21
_parse_repr_attrs() (*paramspace.paramdim.CoupledParamDim method*), 16
_parse_repr_attrs() (*paramspace.paramdim.ParamDim method*), 13
_parse_repr_attrs() (*paramspace.paramdim.ParamDimBase method*), 10
_parse_value() (*paramspace.paramdim.CoupledParamDim method*), 16
_parse_value() (*paramspace.paramdim.ParamDim method*), 13
_parse_value() (*paramspace.paramdim.ParamDimBase method*), 10
_pdim_constructor() (*in module paramspace.yaml_constructors*), 28
_pspace_constructor() (*in module paramspace.yaml_constructors*), 27
_range_constructor() (*in module paramspace.yaml_constructors*), 28
_range_representer() (*in module paramspace.yaml_representers*), 28
_rec_tuple_conv() (*paramspace.paramdim.CoupledParamDim method*), 16
_rec_tuple_conv() (*paramspace.paramdim.ParamDim method*), 13
_rec_tuple_conv() (*paramspace.paramdim.ParamDimBase method*), 10
_set_values() (*paramspace.paramdim.CoupledParamDim method*), 16
_set_values() (*paramspace.paramdim.ParamDim method*), 13
_set_values() (*paramspace.paramdim.ParamDimBase method*), 10
_slice_constructor() (*in module paramspace.yaml_constructors*), 28
_slice_representer() (*in module paramspace.yaml_representers*), 28
_target_name_as_list (*paramspace.paramdim.CoupledParamDim property*), 15
_unique_dim_names() (*paramspace.paramspace.ParamSpace static method*), 18

A

activate_subspace() (*paramspace.paramspace.ParamSpace method*), 23
active_state_map (*paramspace.paramspace.ParamSpace property*), 22
args (*paramspace.paramdim.MaskedValueError attribute*), 7

C

coords (*paramspace.paramdim.CoupledParamDim property*), 17
coords (*paramspace.paramdim.ParamDim property*), 13
coords (*paramspace.paramdim.ParamDimBase property*), 9
coords (*paramspace.paramspace.ParamSpace property*), 19
coupled_dims (*paramspace.paramspace.ParamSpace property*), 19
coupled_dims_by_loc (*paramspace.paramspace.ParamSpace property*), 19
coupled_pdim() (*in module paramspace.yaml_constructors*), 27
coupled_pdim_default() (*in module paramspace.yaml_constructors*), 27
CoupledParamDim (*class in paramspace.paramdim*), 14
create_indices() (*in module paramspace.tools*), 24
current_coords (*paramspace.paramspace.ParamSpace property*), 19

current_point (*paramspace.paramspace.ParamSpace property*), 19
current_value (*paramspace.paramdim.CoupledParamDim property*), 18
current_value (*paramspace.paramdim.ParamDim property*), 13
current_value (*paramspace.paramdim.ParamDimBase property*), 9

D

default (*paramspace.paramdim.CoupledParamDim property*), 17
default (*paramspace.paramdim.ParamDim property*), 13
default (*paramspace.paramdim.ParamDimBase property*), 8
default (*paramspace.paramspace.ParamSpace property*), 19
dims (*paramspace.paramspace.ParamSpace property*), 19
dims_by_loc (*paramspace.paramspace.ParamSpace property*), 19

E

enter_iteration() (*paramspace.paramdim.CoupledParamDim method*), 15
enter_iteration() (*paramspace.paramdim.ParamDim method*), 12
enter_iteration() (*paramspace.paramdim.ParamDimBase method*), 10

F

from_yaml() (*paramspace.paramdim.CoupledParamDim class method*), 17
from_yaml() (*paramspace.paramdim.ParamDim class method*), 13
from_yaml() (*paramspace.paramdim.ParamDimBase class method*), 11
from_yaml() (*paramspace.paramspace.ParamSpace class method*), 21
full_shape (*paramspace.paramspace.ParamSpace property*), 20
full_volume (*paramspace.paramspace.ParamSpace property*), 20

G

get() (*paramspace.paramspace.ParamSpace method*), 21
get_dim_values() (*paramspace.paramspace.ParamSpace method*), 22
get_info_dict() (*paramspace.paramspace.ParamSpace method*), 20
get_info_str() (*paramspace.paramspace.ParamSpace method*), 20
get_key_val_iter() (*in module paramspace.tools*), 27
get_state_vector() (*paramspace.paramspace.ParamSpace method*), 22

I

is_iterable() (*in module paramspace.tools*), 26
iterate_state() (*paramspace.paramdim.CoupledParamDim method*), 15
iterate_state() (*paramspace.paramdim.ParamDim method*), 12
iterate_state() (*paramspace.paramdim.ParamDimBase method*), 10
iterator() (*paramspace.paramspace.ParamSpace method*), 21

L

log (*in module paramspace.tools*), 24

M

mask (*paramspace.paramdim.CoupledParamDim property*), 18
mask (*paramspace.paramdim.ParamDim property*), 12
mask_tuple (*paramspace.paramdim.ParamDim property*), 12
Masked (*class in paramspace.paramdim*), 7
MaskedValueError, 7
max_state_no (*paramspace.paramspace.ParamSpace property*), 20
module
    paramspace, 7
    paramspace.paramdim, 7
    paramspace.paramspace, 18
    paramspace.tools, 24
    paramspace.yaml, 27
    paramspace.yaml_constructors, 27
    paramspace.yaml_representers, 28

N

name (*paramspace.paramdim.CoupledParamDim property*), 17
name (*paramspace.paramdim.ParamDim property*), 14
name (*paramspace.paramdim.ParamDimBase property*), 8
num_coupled_dims (*paramspace.paramspace.ParamSpace property*), 19
num_dims (*paramspace.paramspace.ParamSpace property*), 19
num_masked (*paramspace.paramdim.ParamDim property*), 12
num_states (*paramspace.paramdim.CoupledParamDim property*), 17
num_states (*paramspace.paramdim.ParamDim property*), 14

num_states (*paramspace.paramdim.ParamDimBase property*), 9
num_values (*paramspace.paramdim.CoupledParamDim property*), 17
num_values (*paramspace.paramdim.ParamDim property*), 14
num_values (*paramspace.paramdim.ParamDimBase property*), 9

O

order (*paramspace.paramdim.CoupledParamDim property*), 17
order (*paramspace.paramdim.ParamDim property*), 14
order (*paramspace.paramdim.ParamDimBase property*), 8

P

ParamDim (*class in paramspace.paramdim*), 11
ParamDimBase (*class in paramspace.paramdim*), 8
paramspace
    module, 7
ParamSpace (*class in paramspace.paramspace*), 18
paramspace.paramdim
    module, 7
paramspace.paramspace
    module, 18
paramspace.tools
    module, 24
paramspace.yaml
    module, 27
paramspace.yaml_constructors
    module, 27
paramspace.yaml_representers
    module, 28
pdim() (*in module paramspace.yaml_constructors*), 27
pdim_default() (*in module paramspace.yaml_constructors*), 27
pop() (*paramspace.paramspace.ParamSpace method*), 21
pspace() (*in module paramspace.yaml_constructors*), 27
pspace_unsorted() (*in module paramspace.yaml_constructors*), 27
pure_coords (*paramspace.paramdim.CoupledParamDim property*), 17
pure_coords (*paramspace.paramdim.ParamDim property*), 14
pure_coords (*paramspace.paramdim.ParamDimBase property*), 9
pure_coords (*paramspace.paramspace.ParamSpace property*), 19

R

recursive_collect() (*in module paramspace.tools*), 25
recursive_contains() (*in module paramspace.tools*), 24
recursive_getitem() (*in module paramspace.tools*), 24
recursive_replace() (*in module paramspace.tools*), 26
recursive_setitem() (*in module paramspace.tools*), 25
recursive_update() (*in module paramspace.tools*), 25
recursively_sort_dict() (*in module paramspace.yaml_constructors*), 28
reset() (*paramspace.paramdim.CoupledParamDim method*), 15
reset() (*paramspace.paramdim.ParamDim method*), 12
reset() (*paramspace.paramdim.ParamDimBase method*), 10
reset() (*paramspace.paramspace.ParamSpace method*), 21

S

set_mask() (*paramspace.paramspace.ParamSpace method*), 22
set_masks() (*paramspace.paramspace.ParamSpace method*), 23
shape (*paramspace.paramspace.ParamSpace property*), 20
Skip (*class in paramspace.tools*), 24
SKIP (*in module paramspace.tools*), 24
state (*paramspace.paramdim.CoupledParamDim property*), 18
state (*paramspace.paramdim.ParamDim property*), 12
state (*paramspace.paramdim.ParamDimBase property*), 9
state_map (*paramspace.paramspace.ParamSpace property*), 22
state_no (*paramspace.paramspace.ParamSpace property*), 20
state_vector (*paramspace.paramspace.ParamSpace property*), 20
states_shape (*paramspace.paramspace.ParamSpace property*), 20

T

target_name (*paramspace.paramdim.CoupledParamDim property*), 15
target_of (*paramspace.paramdim.ParamDim property*), 11
target_pdim (*paramspace.paramdim.CoupledParamDim property*), 17
to_yaml() (*paramspace.paramdim.CoupledParamDim class method*), 17
to_yaml() (*paramspace.paramdim.Masked class method*), 7

`to_yaml()` (*paramspace.paramdim.ParamDim class method*), 14
`to_yaml()` (*paramspace.paramdim.ParamDimBase class method*), 11
`to_yaml()` (*paramspace.paramspace.ParamSpace class method*), 21

V

`value` (*paramspace.paramdim.Masked property*), 7
`values` (*paramspace.paramdim.CoupledParamDim property*), 17
`values` (*paramspace.paramdim.ParamDim property*), 14
`values` (*paramspace.paramdim.ParamDimBase property*), 8
`volume` (*paramspace.paramspace.ParamSpace property*), 19

W

`with_traceback()` (*paramspace.paramdim.MaskedValueError method*), 7

Y

`yaml_tag` (*paramspace.paramdim.CoupledParamDim attribute*), 14
`yaml_tag` (*paramspace.paramdim.ParamDim attribute*), 11
`yaml_tag` (*paramspace.paramspace.ParamSpace attribute*), 18