OpTree

Release 0.11.1.dev6+g83dff71

OpTree Contributors

May 12, 2024

CONTENTS

CHAPTER

ONE

TREE OPERATIONS

1.1 Constants

MAX_RECURSION_DEPTH	Maximum recursion depth for pytree traversal.
NONE_IS_NODE	Literal constant that treats None as a pytree non-leaf node.
NONE_IS_LEAF	Literal constant that treats None as a pytree leaf node.

optree.**MAX_RECURSION_DEPTH**: int = 1000

Maximum recursion depth for pytree traversal. It is 1000.

This limit prevents infinite recursion from causing an overflow of the C stack and crashing Python.

optree.**NONE_IS_NODE**: bool = False

Literal constant that treats None as a pytree non-leaf node.

optree.**NONE_IS_LEAF**: bool = True

Literal constant that treats None as a pytree leaf node.

1.2 Tree Manipulation Functions

tree_flatten(tree[, is_leaf, none_is_leaf, ...])	Flatten a pytree.
tree_flatten_with_path(tree[, is_leaf, ...])	Flatten a pytree and additionally record the paths.
tree_flatten_with_accessor(tree[, is_leaf, ...])	Flatten a pytree and additionally record the accessors.
tree_unflatten(treespec, leaves)	Reconstruct a pytree from the treespec and the leaves.
tree_iter(tree[, is_leaf, none_is_leaf, ...])	Get an iterator over the leaves of a pytree.
tree_leaves(tree[, is_leaf, none_is_leaf, ...])	Get the leaves of a pytree.
tree_structure(tree[, is_leaf, ...])	Get the treespec for a pytree.
tree_paths(tree[, is_leaf, none_is_leaf, ...])	Get the path entries to the leaves of a pytree.
tree_accessors(tree[, is_leaf, ...])	Get the accessors to the leaves of a pytree.
tree_is_leaf(tree[, is_leaf, none_is_leaf, ...])	Test whether the given object is a leaf node.
all_leaves(iterable[, is_leaf, ...])	Test whether all elements in the given iterable are all leaves.
tree_map(func, tree, *rests[, is_leaf, ...])	Map a multi-input function over pytree args to produce a new pytree.

continues on next page

Table 1 – continued from previous page

*tree_map_*(func, tree, *rests[, is_leaf, ...])	Like *tree_map()*, but do an inplace call on each leaf and return the original tree.
tree_map_with_path(func, tree, *rests[, ...])	Map a multi-input function over pytree args as well as the tree paths to produce a new pytree.
*tree_map_with_path_*(func, tree, *rests[, ...])	Like *tree_map_with_path()*, but do an inplace call on each leaf and return the original tree.
tree_map_with_accessor(func, tree, *rests[, ...])	Map a multi-input function over pytree args as well as the tree accessors to produce a new pytree.
*tree_map_with_accessor_*(func, tree, *rests)	Like *tree_map_with_accessor()*, but do an inplace call on each leaf and return the original tree.
tree_replace_nones(sentinel, tree[, namespace])	Replace `None` in `tree` with `sentinel`.
tree_transpose(outer_treespec, ...[, is_leaf])	Transform a tree having tree structure (outer, inner) into one having structure (inner, outer).
tree_transpose_map(func, tree, *rests[, ...])	Map a multi-input function over pytree args to produce a new pytree with transposed structure.
tree_transpose_map_with_path(func, tree, *rests)	Map a multi-input function over pytree args as well as the tree paths to produce a new pytree with transposed structure.
tree_transpose_map_with_accessor(func, tree, ...)	Map a multi-input function over pytree args as well as the tree accessors to produce a new pytree with transposed structure.
tree_broadcast_prefix(prefix_tree, full_tree)	Return a pytree of same structure of `full_tree` with broadcasted subtrees in `prefix_tree`.
broadcast_prefix(prefix_tree, full_tree[, ...])	Return a list of broadcasted leaves in `prefix_tree` to match the number of leaves in `full_tree`.
tree_broadcast_common(tree, other_tree[, ...])	Return two pytrees of common suffix structure of `tree` and `other_tree` with broadcasted subtrees.
broadcast_common(tree, other_tree[, ...])	Return two lists of leaves in `tree` and `other_tree` broadcasted to match the number of leaves in the common suffix structure.
tree_broadcast_map(func, tree, *rests[, ...])	Map a multi-input function over pytree args to produce a new pytree.
tree_broadcast_map_with_path(func, tree, *rests)	Map a multi-input function over pytree args as well as the tree paths to produce a new pytree.
tree_broadcast_map_with_accessor(func, tree, ...)	Map a multi-input function over pytree args as well as the tree accessors to produce a new pytree.
tree_flatten_one_level(tree[, is_leaf, ...])	Flatten the pytree one level, returning a 4-tuple of children, auxiliary data, path entries, and an unflatten function.
prefix_errors(prefix_tree, full_tree[, ...])	Return a list of errors that would be raised by *broadcast_prefix()*.

optree.**tree_flatten**(*tree*, *is_leaf=None*, *, *none_is_leaf=False*, *namespace=''*)

Flatten a pytree.

See also *tree_flatten_with_path()* and *tree_unflatten()*.

The flattening order (i.e., the order of elements in the output list) is deterministic, corresponding to a left-to-right depth-first tree traversal.

```
>>> tree = {'b': (2, [3, 4]), 'a': 1, 'c': None, 'd': 5}
>>> tree_flatten(tree)
(
```

(continues on next page)

(continued from previous page)

```
    [1, 2, 3, 4, 5],
    PyTreeSpec({'a': *, 'b': (*, [*, *]), 'c': None, 'd': *})
)
>>> tree_flatten(tree, none_is_leaf=True)
(
    [1, 2, 3, 4, None, 5],
    PyTreeSpec({'a': *, 'b': (*, [*, *]), 'c': *, 'd': *}, NoneIsLeaf)
)
>>> tree_flatten(1)
([1], PyTreeSpec(*))
>>> tree_flatten(None)
([], PyTreeSpec(None))
>>> tree_flatten(None, none_is_leaf=True)
([None], PyTreeSpec(*, NoneIsLeaf))
```

For unordered dictionaries, `dict` and `collections.defaultdict`, the order is dependent on the **sorted** keys in the dictionary. Please use `collections.OrderedDict` if you want to keep the keys in the insertion order.

```
>>> from collections import OrderedDict
>>> tree = OrderedDict([('b', (2, [3, 4])), ('a', 1), ('c', None), ('d', 5)])
>>> tree_flatten(tree)
(
    [2, 3, 4, 1, 5],
    PyTreeSpec(OrderedDict({'b': (*, [*, *]), 'a': *, 'c': None, 'd': *}))
)
>>> tree_flatten(tree, none_is_leaf=True)
(
    [2, 3, 4, 1, None, 5],
    PyTreeSpec(OrderedDict({'b': (*, [*, *]), 'a': *, 'c': *, 'd': *}), NoneIsLeaf)
)
```

Parameters

- **tree** (*pytree*) – A pytree to flatten.
- **is_leaf** (*callable, optional*) – An optionally specified function that will be called at each flattening step. It should return a boolean, with `True` stopping the traversal and the whole subtree being treated as a leaf, and `False` indicating the flattening should traverse the current object.
- **none_is_leaf** (*bool, optional*) – Whether to treat `None` as a leaf. If `False`, `None` is a non-leaf node with arity 0. Thus `None` is contained in the treespec rather than in the leaves list. (default: `False`)
- **namespace** (*str, optional*) – The registry namespace used for custom pytree node types. (default: `''`, i.e., the global namespace)

Return type

`tuple`[`list`[`TypeVar`(T)], *PyTreeSpec*]

Returns

A pair `(leaves, treespec)` where the first element is a list of leaf values and the second element is a treespec representing the structure of the pytree.

optree.**tree_flatten_with_path**(*tree*, *is_leaf=None*, *, *none_is_leaf=False*, *namespace=''*)

Flatten a pytree and additionally record the paths.

See also *tree_flatten()*, *tree_paths()*, and *treespec_paths()*.

The flattening order (i.e., the order of elements in the output list) is deterministic, corresponding to a left-to-right depth-first tree traversal.

```
>>> tree = {'b': (2, [3, 4]), 'a': 1, 'c': None, 'd': 5}
>>> tree_flatten_with_path(tree)
(
    [('a',), ('b', 0), ('b', 1, 0), ('b', 1, 1), ('d',)],
    [1, 2, 3, 4, 5],
    PyTreeSpec({'a': *, 'b': (*, [*, *]), 'c': None, 'd': *})
)
>>> tree_flatten_with_path(tree, none_is_leaf=True)
(
    [('a',), ('b', 0), ('b', 1, 0), ('b', 1, 1), ('c',), ('d',)],
    [1, 2, 3, 4, None, 5],
    PyTreeSpec({'a': *, 'b': (*, [*, *]), 'c': *, 'd': *}, NoneIsLeaf)
)
>>> tree_flatten_with_path(1)
([()], [1], PyTreeSpec(*))
>>> tree_flatten_with_path(None)
([], [], PyTreeSpec(None))
>>> tree_flatten_with_path(None, none_is_leaf=True)
([()], [None], PyTreeSpec(*, NoneIsLeaf))
```

For unordered dictionaries, `dict` and `collections.defaultdict`, the order is dependent on the **sorted** keys in the dictionary. Please use `collections.OrderedDict` if you want to keep the keys in the insertion order.

```
>>> from collections import OrderedDict
>>> tree = OrderedDict([('b', (2, [3, 4])), ('a', 1), ('c', None), ('d', 5)])
>>> tree_flatten_with_path(tree)
(
    [('b', 0), ('b', 1, 0), ('b', 1, 1), ('a',), ('d',)],
    [2, 3, 4, 1, 5],
    PyTreeSpec(OrderedDict({'b': (*, [*, *]), 'a': *, 'c': None, 'd': *}))
)
>>> tree_flatten_with_path(tree, none_is_leaf=True)
(
    [('b', 0), ('b', 1, 0), ('b', 1, 1), ('a',), ('c',), ('d',)],
    [2, 3, 4, 1, None, 5],
    PyTreeSpec(OrderedDict({'b': (*, [*, *]), 'a': *, 'c': *, 'd': *}), NoneIsLeaf)
)
```

Parameters

- **tree** (*pytree*) – A pytree to flatten.
- **is_leaf** (*callable, optional*) – An optionally specified function that will be called at each flattening step. It should return a boolean, with `True` stopping the traversal and the whole subtree being treated as a leaf, and `False` indicating the flattening should traverse the current object.
- **none_is_leaf** (*bool, optional*) – Whether to treat `None` as a leaf. If `False`, `None` is a

non-leaf node with arity 0. Thus `None` is contained in the treespec rather than in the leaves list. (default: `False`)

- **namespace** (*str, optional*) – The registry namespace used for custom pytree node types. (default: `''`, i.e., the global namespace)

Return type
`tuple[list[tuple[Any, ...]], list[TypeVar(T)], PyTreeSpec]`

Returns
A triple `(paths, leaves, treespec)`. The first element is a list of the paths to the leaf values, while each path is a tuple of the index or keys. The second element is a list of leaf values and the last element is a treespec representing the structure of the pytree.

optree.**tree_flatten_with_accessor**(*tree*, *is_leaf=None*, *, *none_is_leaf=False*, *namespace=''*)

Flatten a pytree and additionally record the accessors.

See also *tree_flatten()*, *tree_accessors()*, and *treespec_accessors()*.

The flattening order (i.e., the order of elements in the output list) is deterministic, corresponding to a left-to-right depth-first tree traversal.

```
>>> tree = {'b': (2, [3, 4]), 'a': 1, 'c': None, 'd': 5}
>>> tree_flatten_with_accessor(tree)
(
    [
        PyTreeAccessor(*['a'], (MappingEntry(key='a', type=<class 'dict'>),)),
        PyTreeAccessor(*['b'][0], (MappingEntry(key='b', type=<class 'dict'>),␣
↪SequenceEntry(index=0, type=<class 'tuple'>))),
        PyTreeAccessor(*['b'][1][0], (MappingEntry(key='b', type=<class 'dict'>),␣
↪SequenceEntry(index=1, type=<class 'tuple'>), SequenceEntry(index=0, type=<class
↪'list'>))),
        PyTreeAccessor(*['b'][1][1], (MappingEntry(key='b', type=<class 'dict'>),␣
↪SequenceEntry(index=1, type=<class 'tuple'>), SequenceEntry(index=1, type=<class
↪'list'>))),
        PyTreeAccessor(*['d'], (MappingEntry(key='d', type=<class 'dict'>),))
    ],
    [1, 2, 3, 4, 5],
    PyTreeSpec({'a': *, 'b': (*, [*, *]), 'c': None, 'd': *})
)
>>> tree_flatten_with_accessor(tree, none_is_leaf=True)
(
    [
        PyTreeAccessor(*['a'], (MappingEntry(key='a', type=<class 'dict'>),)),
        PyTreeAccessor(*['b'][0], (MappingEntry(key='b', type=<class 'dict'>),␣
↪SequenceEntry(index=0, type=<class 'tuple'>))),
        PyTreeAccessor(*['b'][1][0], (MappingEntry(key='b', type=<class 'dict'>),␣
↪SequenceEntry(index=1, type=<class 'tuple'>), SequenceEntry(index=0, type=<class
↪'list'>))),
        PyTreeAccessor(*['b'][1][1], (MappingEntry(key='b', type=<class 'dict'>),␣
↪SequenceEntry(index=1, type=<class 'tuple'>), SequenceEntry(index=1, type=<class
↪'list'>))),
        PyTreeAccessor(*['c'], (MappingEntry(key='c', type=<class 'dict'>),)),
        PyTreeAccessor(*['d'], (MappingEntry(key='d', type=<class 'dict'>),))
    ],
    [1, 2, 3, 4, None, 5],
```

(continues on next page)

(continued from previous page)

```
    PyTreeSpec({'a': *, 'b': (*, [*, *]), 'c': *, 'd': *}, NoneIsLeaf)
)
>>> tree_flatten_with_accessor(1)
([PyTreeAccessor(*, ())], [1], PyTreeSpec(*))
>>> tree_flatten_with_accessor(None)
([], [], PyTreeSpec(None))
>>> tree_flatten_with_accessor(None, none_is_leaf=True)
([PyTreeAccessor(*, ())], [None], PyTreeSpec(*, NoneIsLeaf))
```

For unordered dictionaries, `dict` and `collections.defaultdict`, the order is dependent on the **sorted** keys in the dictionary. Please use `collections.OrderedDict` if you want to keep the keys in the insertion order.

```
>>> from collections import OrderedDict
>>> tree = OrderedDict([('b', (2, [3, 4])), ('a', 1), ('c', None), ('d', 5)])
>>> tree_flatten_with_accessor(tree)
(
    [
        PyTreeAccessor(*['b'][0], (MappingEntry(key='b', type=<class 'collections.
↪OrderedDict'>), SequenceEntry(index=0, type=<class 'tuple'>))),
        PyTreeAccessor(*['b'][1][0], (MappingEntry(key='b', type=<class
↪'collections.OrderedDict'>), SequenceEntry(index=1, type=<class 'tuple'>),␣
↪SequenceEntry(index=0, type=<class 'list'>))),
        PyTreeAccessor(*['b'][1][1], (MappingEntry(key='b', type=<class
↪'collections.OrderedDict'>), SequenceEntry(index=1, type=<class 'tuple'>),␣
↪SequenceEntry(index=1, type=<class 'list'>))),
        PyTreeAccessor(*['a'], (MappingEntry(key='a', type=<class 'collections.
↪OrderedDict'>),)),
        PyTreeAccessor(*['d'], (MappingEntry(key='d', type=<class 'collections.
↪OrderedDict'>),))
    ],
    [2, 3, 4, 1, 5],
    PyTreeSpec(OrderedDict({'b': (*, [*, *]), 'a': *, 'c': None, 'd': *}))
)
>>> tree_flatten_with_accessor(tree, none_is_leaf=True)
(
    [
        PyTreeAccessor(*['b'][0], (MappingEntry(key='b', type=<class 'collections.
↪OrderedDict'>), SequenceEntry(index=0, type=<class 'tuple'>))),
        PyTreeAccessor(*['b'][1][0], (MappingEntry(key='b', type=<class
↪'collections.OrderedDict'>), SequenceEntry(index=1, type=<class 'tuple'>),␣
↪SequenceEntry(index=0, type=<class 'list'>))),
        PyTreeAccessor(*['b'][1][1], (MappingEntry(key='b', type=<class
↪'collections.OrderedDict'>), SequenceEntry(index=1, type=<class 'tuple'>),␣
↪SequenceEntry(index=1, type=<class 'list'>))),
        PyTreeAccessor(*['a'], (MappingEntry(key='a', type=<class 'collections.
↪OrderedDict'>),)),
        PyTreeAccessor(*['c'], (MappingEntry(key='c', type=<class 'collections.
↪OrderedDict'>),)),
        PyTreeAccessor(*['d'], (MappingEntry(key='d', type=<class 'collections.
↪OrderedDict'>),))
    ],
    [2, 3, 4, 1, None, 5],
```

(continues on next page)

(continued from previous page)

```
    PyTreeSpec(OrderedDict({'b': (*, [*, *]), 'a': *, 'c': *, 'd': *}), NoneIsLeaf)
)
```

Parameters

- **tree** (*pytree*) – A pytree to flatten.
- **is_leaf** (*callable, optional*) – An optionally specified function that will be called at each flattening step. It should return a boolean, with `True` stopping the traversal and the whole subtree being treated as a leaf, and `False` indicating the flattening should traverse the current object.
- **none_is_leaf** (*bool, optional*) – Whether to treat `None` as a leaf. If `False`, `None` is a non-leaf node with arity 0. Thus `None` is contained in the treespec rather than in the leaves list. (default: `False`)
- **namespace** (*str, optional*) – The registry namespace used for custom pytree node types. (default: `''`, i.e., the global namespace)

Return type

`tuple`[`list`[PyTreeAccessor], `list`[`TypeVar`(T)], *PyTreeSpec*]

Returns

A triple `(accessors, leaves, treespec)`. The first element is a list of accessors to the leaf values. The second element is a list of leaf values and the last element is a treespec representing the structure of the pytree.

optree.**tree_unflatten**(*treespec*, *leaves*)

Reconstruct a pytree from the treespec and the leaves.

The inverse of *tree_flatten()*.

```
>>> tree = {'b': (2, [3, 4]), 'a': 1, 'c': None, 'd': 5}
>>> leaves, treespec = tree_flatten(tree)
>>> tree == tree_unflatten(treespec, leaves)
True
```

Parameters

- **treespec** (`PyTreeSpec`) – The treespec to reconstruct.
- **leaves** (*iterable*) – The list of leaves to use for reconstruction. The list must match the number of leaves of the treespec.

Return type

`Union`[`TypeVar`(T), `Tuple`[PyTree[T], `...`], `List`[PyTree[T]], `Dict`[`Any`, PyTree[T]], `Deque`[PyTree[T]], *CustomTreeNode*[PyTree[T]]]

Returns

The reconstructed pytree, containing the `leaves` placed in the structure described by `treespec`.

optree.**tree_iter**(*tree*, *is_leaf=None*, ***, *none_is_leaf=False*, *namespace=''*)

Get an iterator over the leaves of a pytree.

See also *tree_flatten()* and *tree_leaves()*.

```
>>> tree = {'b': (2, [3, 4]), 'a': 1, 'c': None, 'd': 5}
>>> list(tree_iter(tree))
[1, 2, 3, 4, 5]
>>> list(tree_iter(tree, none_is_leaf=True))
[1, 2, 3, 4, None, 5]
>>> list(tree_iter(1))
[1]
>>> list(tree_iter(None))
[]
>>> list(tree_iter(None, none_is_leaf=True))
[None]
```

Parameters

- **tree** (*pytree*) – A pytree to iterate over.
- **is_leaf** (*callable, optional*) – An optionally specified function that will be called at each flattening step. It should return a boolean, with `True` stopping the traversal and the whole subtree being treated as a leaf, and `False` indicating the flattening should traverse the current object.
- **none_is_leaf** (*bool, optional*) – Whether to treat `None` as a leaf. If `False`, `None` is a non-leaf node with arity 0. Thus `None` is contained in the treespec rather than in the leaves list. (default: `False`)
- **namespace** (*str, optional*) – The registry namespace used for custom pytree node types. (default: `''`, i.e., the global namespace)

Return type
`Iterable`[`TypeVar`(T)]

Returns
An iterator over the leaf values.

optree.**tree_leaves**(*tree*, *is_leaf=None*, ***, *none_is_leaf=False*, *namespace=''*)

Get the leaves of a pytree.

See also *tree_flatten()* and *tree_iter()*.

```
>>> tree = {'b': (2, [3, 4]), 'a': 1, 'c': None, 'd': 5}
>>> tree_leaves(tree)
[1, 2, 3, 4, 5]
>>> tree_leaves(tree, none_is_leaf=True)
[1, 2, 3, 4, None, 5]
>>> tree_leaves(1)
[1]
>>> tree_leaves(None)
[]
>>> tree_leaves(None, none_is_leaf=True)
[None]
```

Parameters

- **tree** (*pytree*) – A pytree to flatten.
- **is_leaf** (*callable, optional*) – An optionally specified function that will be called at each flattening step. It should return a boolean, with `True` stopping the traversal and the

whole subtree being treated as a leaf, and `False` indicating the flattening should traverse the current object.

- **none_is_leaf** (*bool, optional*) – Whether to treat `None` as a leaf. If `False`, `None` is a non-leaf node with arity 0. Thus `None` is contained in the treespec rather than in the leaves list. (default: `False`)
- **namespace** (*str, optional*) – The registry namespace used for custom pytree node types. (default: `''`, i.e., the global namespace)

Return type
`list`[`TypeVar`(`T`)]

Returns
A list of leaf values.

optree.**tree_structure**(*tree*, *is_leaf=None*, *, *none_is_leaf=False*, *namespace=''*)

Get the treespec for a pytree.

See also *tree_flatten()*.

```
>>> tree = {'b': (2, [3, 4]), 'a': 1, 'c': None, 'd': 5}
>>> tree_structure(tree)
PyTreeSpec({'a': *, 'b': (*, [*, *]), 'c': None, 'd': *})
>>> tree_structure(tree, none_is_leaf=True)
PyTreeSpec({'a': *, 'b': (*, [*, *]), 'c': *, 'd': *}, NoneIsLeaf)
>>> tree_structure(1)
PyTreeSpec(*)
>>> tree_structure(None)
PyTreeSpec(None)
>>> tree_structure(None, none_is_leaf=True)
PyTreeSpec(*, NoneIsLeaf)
```

Parameters

- **tree** (*pytree*) – A pytree to flatten.
- **is_leaf** (*callable, optional*) – An optionally specified function that will be called at each flattening step. It should return a boolean, with `True` stopping the traversal and the whole subtree being treated as a leaf, and `False` indicating the flattening should traverse the current object.
- **none_is_leaf** (*bool, optional*) – Whether to treat `None` as a leaf. If `False`, `None` is a non-leaf node with arity 0. Thus `None` is contained in the treespec rather than in the leaves list. (default: `False`)
- **namespace** (*str, optional*) – The registry namespace used for custom pytree node types. (default: `''`, i.e., the global namespace)

Return type
PyTreeSpec

Returns
A treespec object representing the structure of the pytree.

optree.**tree_paths**(*tree*, *is_leaf=None*, *, *none_is_leaf=False*, *namespace=''*)

Get the path entries to the leaves of a pytree.

See also *tree_flatten()*, *tree_flatten_with_path()*, and *treespec_paths()*.

```
>>> tree = {'b': (2, [3, 4]), 'a': 1, 'c': None, 'd': 5}
>>> tree_paths(tree)
[('a',), ('b', 0), ('b', 1, 0), ('b', 1, 1), ('d',)]
>>> tree_paths(tree, none_is_leaf=True)
[('a',), ('b', 0), ('b', 1, 0), ('b', 1, 1), ('c',), ('d',)]
>>> tree_paths(1)
[()]
>>> tree_paths(None)
[]
>>> tree_paths(None, none_is_leaf=True)
[()]
```

Parameters

- **tree** (*pytree*) – A pytree to flatten.
- **is_leaf** (*callable, optional*) – An optionally specified function that will be called at each flattening step. It should return a boolean, with `True` stopping the traversal and the whole subtree being treated as a leaf, and `False` indicating the flattening should traverse the current object.
- **none_is_leaf** (*bool, optional*) – Whether to treat `None` as a leaf. If `False`, `None` is a non-leaf node with arity 0. Thus `None` is contained in the treespec rather than in the leaves list. (default: `False`)
- **namespace** (*str, optional*) – The registry namespace used for custom pytree node types. (default: `''`, i.e., the global namespace)

Return type
`list[tuple[Any, ...]]`

Returns
A list of the paths to the leaf values, while each path is a tuple of the index or keys.

optree.**tree_accessors**(*tree*, *is_leaf=None*, ***, *none_is_leaf=False*, *namespace=''*)

Get the accessors to the leaves of a pytree.

See also *tree_flatten()*, *tree_flatten_with_accessor()*, and *treespec_accessors()*.

```
>>> tree = {'b': (2, [3, 4]), 'a': 1, 'c': None, 'd': 5}
>>> tree_accessors(tree)
[
    PyTreeAccessor(*['a'], (MappingEntry(key='a', type=<class 'dict'>),)),
    PyTreeAccessor(*['b'][0], (MappingEntry(key='b', type=<class 'dict'>),␣
↪SequenceEntry(index=0, type=<class 'tuple'>))),
    PyTreeAccessor(*['b'][1][0], (MappingEntry(key='b', type=<class 'dict'>),␣
↪SequenceEntry(index=1, type=<class 'tuple'>), SequenceEntry(index=0, type=<class
↪'list'>))),
    PyTreeAccessor(*['b'][1][1], (MappingEntry(key='b', type=<class 'dict'>),␣
↪SequenceEntry(index=1, type=<class 'tuple'>), SequenceEntry(index=1, type=<class
↪'list'>))),
    PyTreeAccessor(*['d'], (MappingEntry(key='d', type=<class 'dict'>),))
]
>>> tree_accessors(tree, none_is_leaf=True)
[
    PyTreeAccessor(*['a'], (MappingEntry(key='a', type=<class 'dict'>),)),
```

(continues on next page)

(continued from previous page)

```
    PyTreeAccessor(*['b'][0], (MappingEntry(key='b', type=<class 'dict'>),␣
↪SequenceEntry(index=0, type=<class 'tuple'>))),
    PyTreeAccessor(*['b'][1][0], (MappingEntry(key='b', type=<class 'dict'>),␣
↪SequenceEntry(index=1, type=<class 'tuple'>), SequenceEntry(index=0, type=<class
↪'list'>))),
    PyTreeAccessor(*['b'][1][1], (MappingEntry(key='b', type=<class 'dict'>),␣
↪SequenceEntry(index=1, type=<class 'tuple'>), SequenceEntry(index=1, type=<class
↪'list'>))),
    PyTreeAccessor(*['c'], (MappingEntry(key='c', type=<class 'dict'>),)),
    PyTreeAccessor(*['d'], (MappingEntry(key='d', type=<class 'dict'>),))
]
>>> tree_accessors(1)
[PyTreeAccessor(*, ())]
>>> tree_accessors(None)
[]
>>> tree_accessors(None, none_is_leaf=True)
[PyTreeAccessor(*, ())]
```

Parameters

- **tree** (*pytree*) – A pytree to flatten.
- **is_leaf** (*callable, optional*) – An optionally specified function that will be called at each flattening step. It should return a boolean, with `True` stopping the traversal and the whole subtree being treated as a leaf, and `False` indicating the flattening should traverse the current object.
- **none_is_leaf** (*bool, optional*) – Whether to treat `None` as a leaf. If `False`, `None` is a non-leaf node with arity 0. Thus `None` is contained in the treespec rather than in the leaves list. (default: `False`)
- **namespace** (*str, optional*) – The registry namespace used for custom pytree node types. (default: `''`, i.e., the global namespace)

Return type

`list`[PyTreeAccessor]

Returns

A list of accessors to the leaf values.

optree.**tree_is_leaf**(*tree*, *is_leaf=None*, *, *none_is_leaf=False*, *namespace=''*)

Test whether the given object is a leaf node.

See also *tree_flatten()*, *tree_leaves()*, and *all_leaves()*.

```
>>> tree_is_leaf(1)
True
>>> tree_is_leaf(None)
False
>>> tree_is_leaf(None, none_is_leaf=True)
True
>>> tree_is_leaf({'a': 1, 'b': (2, 3)})
False
```

Parameters

- **tree** (*pytree*) – A pytree to check if it is a leaf node.
- **is_leaf** (*callable, optional*) – An optionally specified function that will be called at each flattening step. It should return a boolean, with `True` stopping the traversal and the whole subtree being treated as a leaf, and `False` indicating the flattening should traverse the current object.
- **none_is_leaf** (*bool, optional*) – Whether to treat `None` as a leaf. If `False`, `None` is a non-leaf node with arity 0. Thus `None` is contained in the treespec rather than a leaf. (default: `False`)
- **namespace** (*str, optional*) – The registry namespace used for custom pytree node types. (default: `''`, i.e., the global namespace)

Return type
`bool`

Returns
A boolean indicating if all elements in the input iterable are leaves.

`optree.`**`all_leaves`**(*iterable*, *is_leaf=None*, *, *none_is_leaf=False*, *namespace=''*)

Test whether all elements in the given iterable are all leaves.

See also *tree_flatten()*, *tree_leaves()*, and *tree_is_leaf()*.

```
>>> tree = {'a': [1, 2, 3]}
>>> all_leaves(tree_leaves(tree))
True
>>> all_leaves([tree])
False
>>> all_leaves([1, 2, None, 3])
False
>>> all_leaves([1, 2, None, 3], none_is_leaf=True)
True
```

Note that this function iterates and checks the elements in the input iterable object, which uses the `iter()` function. For dictionaries, `iter(d)` for a dictionary `d` iterates the keys of the dictionary, not the values.

```
>>> list({'a': 1, 'b': (2, 3)})
['a', 'b']
>>> all_leaves({'a': 1, 'b': (2, 3)})
True
```

This function is useful in advanced cases. For example, if a library allows arbitrary map operations on a flat list of leaves it may want to check if the result is still a flat list of leaves.

Parameters

- **iterable** (*iterable*) – A iterable of leaves.
- **is_leaf** (*callable, optional*) – An optionally specified function that will be called at each flattening step. It should return a boolean, with `True` stopping the traversal and the whole subtree being treated as a leaf, and `False` indicating the flattening should traverse the current object.
- **none_is_leaf** (*bool, optional*) – Whether to treat `None` as a leaf. If `False`, `None` is a non-leaf node with arity 0. Thus `None` is contained in the treespec rather than a leaf. (default: `False`)

- **namespace** (*str, optional*) – The registry namespace used for custom pytree node types. (default: `''`, i.e., the global namespace)

Return type
bool

Returns
A boolean indicating if all elements in the input iterable are leaves.

optree.**tree_map**(*func*, *tree*, **rests*, *is_leaf=None*, *none_is_leaf=False*, *namespace=''*)

Map a multi-input function over pytree args to produce a new pytree.

See also *tree_map_()*, *tree_map_with_path()*, *tree_map_with_path_()*, and *tree_broadcast_map()*.

```
>>> tree_map(lambda x: x + 1, {'x': 7, 'y': (42, 64)})
{'x': 8, 'y': (43, 65)}
>>> tree_map(lambda x: x + 1, {'x': 7, 'y': (42, 64), 'z': None})
{'x': 8, 'y': (43, 65), 'z': None}
>>> tree_map(lambda x: x is None, {'x': 7, 'y': (42, 64), 'z': None})
{'x': False, 'y': (False, False), 'z': None}
>>> tree_map(lambda x: x is None, {'x': 7, 'y': (42, 64), 'z': None}, none_is_
↪leaf=True)
{'x': False, 'y': (False, False), 'z': True}
```

If multiple inputs are given, the structure of the tree is taken from the first input; subsequent inputs need only have `tree` as a prefix:

```
>>> tree_map(lambda x, y: [x] + y, [5, 6], [[7, 9], [1, 2]])
[[5, 7, 9], [6, 1, 2]]
```

Parameters

- **func** (*callable*) – A function that takes `1 + len(rests)` arguments, to be applied at the corresponding leaves of the pytrees.
- **tree** (*pytree*) – A pytree to be mapped over, with each leaf providing the first positional argument to function `func`.
- **rests** (*tuple of pytree*) – A tuple of pytrees, each of which has the same structure as `tree` or has `tree` as a prefix.
- **is_leaf** (*callable, optional*) – An optionally specified function that will be called at each flattening step. It should return a boolean, with `True` stopping the traversal and the whole subtree being treated as a leaf, and `False` indicating the flattening should traverse the current object.
- **none_is_leaf** (*bool, optional*) – Whether to treat `None` as a leaf. If `False`, `None` is a non-leaf node with arity 0. Thus `None` is contained in the treespec rather than in the leaves list and `None` will be remain in the result pytree. (default: `False`)
- **namespace** (*str, optional*) – The registry namespace used for custom pytree node types. (default: `''`, i.e., the global namespace)

Return type
Union[TypeVar(U), Tuple[PyTree[U], ...], List[PyTree[U]], Dict[Any, PyTree[U]], Deque[PyTree[U]], *CustomTreeNode*[PyTree[U]]]

Returns
A new pytree with the same structure as `tree` but with the value at each leaf given by `func(x,`

*xs) where x is the value at the corresponding leaf in tree and xs is the tuple of values at corresponding nodes in rests.

optree.**tree_map_**(*func*, *tree*, **rests*, *is_leaf=None*, *none_is_leaf=False*, *namespace=''*)

Like *tree_map()*, but do an inplace call on each leaf and return the original tree.

See also *tree_map()*, *tree_map_with_path()*, and *tree_map_with_path_()*.

Parameters

- **func** (*callable*) – A function that takes `1 + len(rests)` arguments, to be applied at the corresponding leaves of the pytrees.
- **tree** (*pytree*) – A pytree to be mapped over, with each leaf providing the first positional argument to function `func`.
- **rests** (*tuple of pytree*) – A tuple of pytrees, each of which has the same structure as `tree` or has `tree` as a prefix.
- **is_leaf** (*callable, optional*) – An optionally specified function that will be called at each flattening step. It should return a boolean, with `True` stopping the traversal and the whole subtree being treated as a leaf, and `False` indicating the flattening should traverse the current object.
- **none_is_leaf** (*bool, optional*) – Whether to treat `None` as a leaf. If `False`, `None` is a non-leaf node with arity 0. Thus `None` is contained in the treespec rather than in the leaves list and `None` will be remain in the result pytree. (default: `False`)
- **namespace** (*str, optional*) – The registry namespace used for custom pytree node types. (default: `''`, i.e., the global namespace)

Return type

`Union`[`TypeVar`(T), `Tuple`[PyTree[T], `...`], `List`[PyTree[T]], `Dict`[`Any`, PyTree[T]], `Deque`[PyTree[T]], *`CustomTreeNode`*[PyTree[T]]]

Returns

The original `tree` with the value at each leaf is given by the side-effect of function `func(x, *xs)` (not the return value) where `x` is the value at the corresponding leaf in `tree` and `xs` is the tuple of values at values at corresponding nodes in `rests`.

optree.**tree_map_with_path**(*func*, *tree*, **rests*, *is_leaf=None*, *none_is_leaf=False*, *namespace=''*)

Map a multi-input function over pytree args as well as the tree paths to produce a new pytree.

See also *tree_map()*, *tree_map_()*, and *tree_map_with_path_()*.

```
>>> tree_map_with_path(lambda p, x: (len(p), x), {'x': 7, 'y': (42, 64)})
{'x': (1, 7), 'y': ((2, 42), (2, 64))}
>>> tree_map_with_path(lambda p, x: x + len(p), {'x': 7, 'y': (42, 64), 'z': None})
{'x': 8, 'y': (44, 66), 'z': None}
>>> tree_map_with_path(lambda p, x: p, {'x': 7, 'y': (42, 64), 'z': {1.5: None}})
{'x': ('x',), 'y': (('y', 0), ('y', 1)), 'z': {1.5: None}}
>>> tree_map_with_path(lambda p, x: p, {'x': 7, 'y': (42, 64), 'z': {1.5: None}},␣
↪none_is_leaf=True)
{'x': ('x',), 'y': (('y', 0), ('y', 1)), 'z': {1.5: ('z', 1.5)}}
```

Parameters

- **func** (*callable*) – A function that takes `2 + len(rests)` arguments, to be applied at the corresponding leaves of the pytrees with extra paths.

- **tree** (*pytree*) – A pytree to be mapped over, with each leaf providing the second positional argument and the corresponding path providing the first positional argument to function `func`.
- **rests** (*tuple of pytree*) – A tuple of pytrees, each of which has the same structure as `tree` or has `tree` as a prefix.
- **is_leaf** (*callable, optional*) – An optionally specified function that will be called at each flattening step. It should return a boolean, with `True` stopping the traversal and the whole subtree being treated as a leaf, and `False` indicating the flattening should traverse the current object.
- **none_is_leaf** (*bool, optional*) – Whether to treat `None` as a leaf. If `False`, `None` is a non-leaf node with arity 0. Thus `None` is contained in the treespec rather than in the leaves list and `None` will be remain in the result pytree. (default: `False`)
- **namespace** (*str, optional*) – The registry namespace used for custom pytree node types. (default: `''`, i.e., the global namespace)

Return type
`Union`[`TypeVar`(U), `Tuple`[PyTree[U], `...`], `List`[PyTree[U]], `Dict`[`Any`, PyTree[U]], `Deque`[PyTree[U]], *CustomTreeNode*[PyTree[U]]]

Returns
A new pytree with the same structure as `tree` but with the value at each leaf given by `func(p, x, *xs)` where `(p, x)` are the path and value at the corresponding leaf in `tree` and `xs` is the tuple of values at corresponding nodes in `rests`.

optree.**tree_map_with_path_**(*func*, *tree*, **rests*, *is_leaf=None*, *none_is_leaf=False*, *namespace=''*)

Like *tree_map_with_path()*, but do an inplace call on each leaf and return the original tree.

See also *tree_map()*, *tree_map_()*, and *tree_map_with_path()*.

Parameters

- **func** (*callable*) – A function that takes `2 + len(rests)` arguments, to be applied at the corresponding leaves of the pytrees with extra paths.
- **tree** (*pytree*) – A pytree to be mapped over, with each leaf providing the second positional argument and the corresponding path providing the first positional argument to function `func`.
- **rests** (*tuple of pytree*) – A tuple of pytrees, each of which has the same structure as `tree` or has `tree` as a prefix.
- **is_leaf** (*callable, optional*) – An optionally specified function that will be called at each flattening step. It should return a boolean, with `True` stopping the traversal and the whole subtree being treated as a leaf, and `False` indicating the flattening should traverse the current object.
- **none_is_leaf** (*bool, optional*) – Whether to treat `None` as a leaf. If `False`, `None` is a non-leaf node with arity 0. Thus `None` is contained in the treespec rather than in the leaves list and `None` will be remain in the result pytree. (default: `False`)
- **namespace** (*str, optional*) – The registry namespace used for custom pytree node types. (default: `''`, i.e., the global namespace)

Return type
`Union`[`TypeVar`(T), `Tuple`[PyTree[T], `...`], `List`[PyTree[T]], `Dict`[`Any`, PyTree[T]], `Deque`[PyTree[T]], *CustomTreeNode*[PyTree[T]]]

Returns

The original `tree` with the value at each leaf is given by the side-effect of function `func(p, x, *xs)` (not the return value) where `(p, x)` are the path and value at the corresponding leaf in `tree` and `xs` is the tuple of values at values at corresponding nodes in `rests`.

optree.**tree_map_with_accessor**(*func*, *tree*, **rests*, *is_leaf=None*, *none_is_leaf=False*, *namespace=''*)

Map a multi-input function over pytree args as well as the tree accessors to produce a new pytree.

See also *tree_map()*, *tree_map_()*, and *tree_map_with_accessor_()*.

```
>>> tree_map_with_accessor(lambda a, x: f'{a.codegen("tree")} = {x!r}', {'x': 7, 'y
↪': (42, 64)})
{'x': "tree['x'] = 7", 'y': ("tree['y'][0] = 42", "tree['y'][1] = 64")}
>>> tree_map_with_accessor(lambda a, x: x + len(a), {'x': 7, 'y': (42, 64), 'z':␣
↪None})
{'x': 8, 'y': (44, 66), 'z': None}
>>> tree_map_with_accessor(
...     lambda a, x: a,
...     {'x': 7, 'y': (42, 64), 'z': {1.5: None}},
... )
{
    'x': PyTreeAccessor(*['x'], (MappingEntry(key='x', type=<class 'dict'>),)),
    'y': (
        PyTreeAccessor(*['y'][0], (MappingEntry(key='y', type=<class 'dict'>),␣
↪SequenceEntry(index=0, type=<class 'tuple'>))),
        PyTreeAccessor(*['y'][1], (MappingEntry(key='y', type=<class 'dict'>),␣
↪SequenceEntry(index=1, type=<class 'tuple'>)))
    ),
    'z': {1.5: None}
}
>>> tree_map_with_accessor(
...     lambda a, x: a,
...     {'x': 7, 'y': (42, 64), 'z': {1.5: None}},
...     none_is_leaf=True,
... )
{
    'x': PyTreeAccessor(*['x'], (MappingEntry(key='x', type=<class 'dict'>),)),
    'y': (
        PyTreeAccessor(*['y'][0], (MappingEntry(key='y', type=<class 'dict'>),␣
↪SequenceEntry(index=0, type=<class 'tuple'>))),
        PyTreeAccessor(*['y'][1], (MappingEntry(key='y', type=<class 'dict'>),␣
↪SequenceEntry(index=1, type=<class 'tuple'>)))
    ),
    'z': {
        1.5: PyTreeAccessor(*['z'][1.5], (MappingEntry(key='z', type=<class 'dict'>
↪), MappingEntry(key=1.5, type=<class 'dict'>)))
    }
}
```

Parameters

- **func** (*callable*) – A function that takes `2 + len(rests)` arguments, to be applied at the corresponding leaves of the pytrees with extra accessors.
- **tree** (*pytree*) – A pytree to be mapped over, with each leaf providing the second posi-

tional argument and the corresponding path providing the first positional argument to function `func`.

- **rests** (*tuple of pytree*) – A tuple of pytrees, each of which has the same structure as `tree` or has `tree` as a prefix.
- **is_leaf** (*callable, optional*) – An optionally specified function that will be called at each flattening step. It should return a boolean, with `True` stopping the traversal and the whole subtree being treated as a leaf, and `False` indicating the flattening should traverse the current object.
- **none_is_leaf** (*bool, optional*) – Whether to treat `None` as a leaf. If `False`, `None` is a non-leaf node with arity 0. Thus `None` is contained in the treespec rather than in the leaves list and `None` will be remain in the result pytree. (default: `False`)
- **namespace** (*str, optional*) – The registry namespace used for custom pytree node types. (default: `''`, i.e., the global namespace)

Return type
`Union`[`TypeVar`(U), `Tuple`[PyTree[U], `...`], `List`[PyTree[U]], `Dict`[`Any`, PyTree[U]], `Deque`[PyTree[U]], *CustomTreeNode*[PyTree[U]]]

Returns
A new pytree with the same structure as `tree` but with the value at each leaf given by `func(a, x, *xs)` where `(a, x)` are the accessor and value at the corresponding leaf in `tree` and `xs` is the tuple of values at corresponding nodes in `rests`.

optree.**tree_map_with_accessor_**(*func*, *tree*, **rests*, *is_leaf=None*, *none_is_leaf=False*, *namespace=''*)

Like *tree_map_with_accessor()*, but do an inplace call on each leaf and return the original tree.

See also *tree_map()*, *tree_map_()*, and *tree_map_with_accessor()*.

Parameters

- **func** (*callable*) – A function that takes `2 + len(rests)` arguments, to be applied at the corresponding leaves of the pytrees with extra accessors.
- **tree** (*pytree*) – A pytree to be mapped over, with each leaf providing the second positional argument and the corresponding path providing the first positional argument to function `func`.
- **rests** (*tuple of pytree*) – A tuple of pytrees, each of which has the same structure as `tree` or has `tree` as a prefix.
- **is_leaf** (*callable, optional*) – An optionally specified function that will be called at each flattening step. It should return a boolean, with `True` stopping the traversal and the whole subtree being treated as a leaf, and `False` indicating the flattening should traverse the current object.
- **none_is_leaf** (*bool, optional*) – Whether to treat `None` as a leaf. If `False`, `None` is a non-leaf node with arity 0. Thus `None` is contained in the treespec rather than in the leaves list and `None` will be remain in the result pytree. (default: `False`)
- **namespace** (*str, optional*) – The registry namespace used for custom pytree node types. (default: `''`, i.e., the global namespace)

Return type
`Union`[`TypeVar`(T), `Tuple`[PyTree[T], `...`], `List`[PyTree[T]], `Dict`[`Any`, PyTree[T]], `Deque`[PyTree[T]], *CustomTreeNode*[PyTree[T]]]

Returns
The original `tree` with the value at each leaf is given by the side-effect of function `func(a, x,`

*xs) (not the return value) where (a, x) are the accessor and value at the corresponding leaf in tree and xs is the tuple of values at values at corresponding nodes in rests.

optree.**tree_replace_nones**(*sentinel*, *tree*, *namespace=''*)

Replace None in tree with sentinel.

See also *tree_flatten()* and *tree_map()*.

```
>>> tree_replace_nones(0, {'a': 1, 'b': None, 'c': (2, None)})
{'a': 1, 'b': 0, 'c': (2, 0)}
>>> tree_replace_nones(0, None)
0
```

Parameters

- **sentinel** (*object*) – The value to replace None with.
- **tree** (*pytree*) – A pytree to be transformed.
- **namespace** (*str, optional*) – The registry namespace used for custom pytree node types. (default: '', i.e., the global namespace)

Return type

Union[TypeVar(T), Tuple[PyTree[T], ...], List[PyTree[T]], Dict[Any, PyTree[T]], Deque[PyTree[T]], *CustomTreeNode*[PyTree[T]]]

Returns

A new pytree with the same structure as tree but with None replaced.

optree.**tree_transpose**(*outer_treespec*, *inner_treespec*, *tree*, *is_leaf=None*)

Transform a tree having tree structure (outer, inner) into one having structure (inner, outer).

See also *tree_flatten()*, *tree_structure()*, and *tree_transpose_map()*.

```
>>> outer_treespec = tree_structure({'a': 1, 'b': 2, 'c': (3, 4)})
>>> outer_treespec
PyTreeSpec({'a': *, 'b': *, 'c': (*, *)})
>>> inner_treespec = tree_structure((1, 2))
>>> inner_treespec
PyTreeSpec((*, *))
>>> tree = {'a': (1, 2), 'b': (3, 4), 'c': ((5, 6), (7, 8))}
>>> tree_transpose(outer_treespec, inner_treespec, tree)
({'a': 1, 'b': 3, 'c': (5, 7)}, {'a': 2, 'b': 4, 'c': (6, 8)})
```

For performance reasons, this function is only checks for the number of leaves in the input pytree, not the structure. The result is only enumerated up to the original order of leaves in tree, then transpose depends on the number of leaves in structure (inner, outer). The caller is responsible for ensuring that the input pytree has a prefix structure of outer_treespec followed by a prefix structure of inner_treespec. Otherwise, the result may be incorrect.

```
>>> tree_transpose(outer_treespec, inner_treespec, list(range(1, 9)))
({'a': 1, 'b': 3, 'c': (5, 7)}, {'a': 2, 'b': 4, 'c': (6, 8)})
```

Parameters

- **outer_treespec** (PyTreeSpec) – A treespec object representing the outer structure of the pytree.

- **inner_treespec** (PyTreeSpec) – A treespec object representing the inner structure of the pytree.
- **tree** (*pytree*) – A pytree to be transposed.
- **is_leaf** (*callable, optional*) – An optionally specified function that will be called at each flattening step. It should return a boolean, with True stopping the traversal and the whole subtree being treated as a leaf, and False indicating the flattening should traverse the current object.

Return type
Union[TypeVar(T), Tuple[PyTree[T], ...], List[PyTree[T]], Dict[Any, PyTree[T]], Deque[PyTree[T]], *CustomTreeNode*[PyTree[T]]]

Returns
A new pytree with the same structure as `inner_treespec` but with the value at each leaf has the same structure as `outer_treespec`.

optree.**tree_transpose_map**(*func*, *tree*, **rests*, *inner_treespec=None*, *is_leaf=None*, *none_is_leaf=False*, *namespace=''*)

Map a multi-input function over pytree args to produce a new pytree with transposed structure.

See also *tree_map()*, *tree_map_with_path()*, and *tree_transpose()*.

```
>>> tree = {'b': (2, [3, 4]), 'a': 1, 'c': (5, 6)}
>>> tree_transpose_map(
...     lambda x: {'identity': x, 'double': 2 * x},
...     tree,
... )
{
    'identity': {'b': (2, [3, 4]), 'a': 1, 'c': (5, 6)},
    'double': {'b': (4, [6, 8]), 'a': 2, 'c': (10, 12)}
}
>>> tree_transpose_map(
...     lambda x: {'identity': x, 'double': (x, x)},
...     tree,
... )
{
    'identity': {'b': (2, [3, 4]), 'a': 1, 'c': (5, 6)},
    'double': (
        {'b': (2, [3, 4]), 'a': 1, 'c': (5, 6)},
        {'b': (2, [3, 4]), 'a': 1, 'c': (5, 6)}
    )
}
>>> tree_transpose_map(
...     lambda x: {'identity': x, 'double': (x, x)},
...     tree,
...     inner_treespec=tree_structure({'identity': 0, 'double': 0}),
... )
{
    'identity': {'b': (2, [3, 4]), 'a': 1, 'c': (5, 6)},
    'double': {'b': ((2, 2), [(3, 3), (4, 4)]), 'a': (1, 1), 'c': ((5, 5), (6, 6))}
}
```

Parameters

- **func** (*callable*) – A function that takes `1 + len(rests)` arguments, to be applied at the corresponding leaves of the pytrees.
- **tree** (*pytree*) – A pytree to be mapped over, with each leaf providing the first positional argument to function `func`.
- **rests** (*tuple of pytree*) – A tuple of pytrees, each of which has the same structure as `tree` or has `tree` as a prefix.
- **inner_treespec** (*PyTreeSpec, optional*) – The treespec object representing the inner structure of the result pytree. If not specified, the inner structure is inferred from the result of the function `func` on the first leaf. (default: `None`)
- **is_leaf** (*callable, optional*) – An optionally specified function that will be called at each flattening step. It should return a boolean, with `True` stopping the traversal and the whole subtree being treated as a leaf, and `False` indicating the flattening should traverse the current object.
- **none_is_leaf** (*bool, optional*) – Whether to treat `None` as a leaf. If `False`, `None` is a non-leaf node with arity 0. Thus `None` is contained in the treespec rather than in the leaves list and `None` will be remain in the result pytree. (default: `False`)
- **namespace** (*str, optional*) – The registry namespace used for custom pytree node types. (default: `''`, i.e., the global namespace)

Return type

`Union`[`TypeVar`(U), `Tuple`[PyTree[U], `...`], `List`[PyTree[U]], `Dict`[`Any`, PyTree[U]], `Deque`[PyTree[U]], *`CustomTreeNode`*[PyTree[U]]]

Returns

A new nested pytree with the same structure as `inner_treespec` but with the value at each leaf has the same structure as `tree`. The subtree at each leaf is given by the result of function `func(x, *xs)` where `x` is the value at the corresponding leaf in `tree` and `xs` is the tuple of values at corresponding nodes in `rests`.

optree.**tree_transpose_map_with_path**(*func*, *tree*, **rests*, *inner_treespec=None*, *is_leaf=None*, *none_is_leaf=False*, *namespace=''*)

Map a multi-input function over pytree args as well as the tree paths to produce a new pytree with transposed structure.

See also *tree_map_with_path()*, *tree_transpose_map()*, and *tree_transpose()*.

```
>>> tree = {'b': (2, [3, 4]), 'a': 1, 'c': (5, 6)}
>>> tree_transpose_map_with_path(
...     lambda p, x: {'depth': len(p), 'value': x},
...     tree,
... )
{
    'depth': {'b': (2, [3, 3]), 'a': 1, 'c': (2, 2)},
    'value': {'b': (2, [3, 4]), 'a': 1, 'c': (5, 6)}
}
>>> tree_transpose_map_with_path(
...     lambda p, x: {'path': p, 'value': x},
...     tree,
...     inner_treespec=tree_structure({'path': 0, 'value': 0}),
... )
{
    'path': {
```

(continues on next page)

(continued from previous page)

```
        'b': (('b', 0), [('b', 1, 0), ('b', 1, 1)]),
        'a': ('a',),
        'c': (('c', 0), ('c', 1))
    },
    'value': {'b': (2, [3, 4]), 'a': 1, 'c': (5, 6)}
}
```

Parameters

- **func** (*callable*) – A function that takes `2 + len(rests)` arguments, to be applied at the corresponding leaves of the pytrees with extra paths.
- **tree** (*pytree*) – A pytree to be mapped over, with each leaf providing the second positional argument and the corresponding path providing the first positional argument to function `func`.
- **rests** (*tuple of pytree*) – A tuple of pytrees, each of which has the same structure as `tree` or has `tree` as a prefix.
- **inner_treespec** (`PyTreeSpec`, *optional*) – The treespec object representing the inner structure of the result pytree. If not specified, the inner structure is inferred from the result of the function `func` on the first leaf. (default: `None`)
- **is_leaf** (*callable*, *optional*) – An optionally specified function that will be called at each flattening step. It should return a boolean, with `True` stopping the traversal and the whole subtree being treated as a leaf, and `False` indicating the flattening should traverse the current object.
- **none_is_leaf** (*bool*, *optional*) – Whether to treat `None` as a leaf. If `False`, `None` is a non-leaf node with arity 0. Thus `None` is contained in the treespec rather than in the leaves list and `None` will be remain in the result pytree. (default: `False`)
- **namespace** (*str*, *optional*) – The registry namespace used for custom pytree node types. (default: `''`, i.e., the global namespace)

Return type

`Union`[`TypeVar`(U), `Tuple`[PyTree[U], `...`], `List`[PyTree[U]], `Dict`[`Any`, PyTree[U]], `Deque`[PyTree[U]], *`CustomTreeNode`*[PyTree[U]]]

Returns

A new nested pytree with the same structure as `inner_treespec` but with the value at each leaf has the same structure as `tree`. The subtree at each leaf is given by the result of function `func(p, x, *xs)` where `(p, x)` are the path and value at the corresponding leaf in `tree` and `xs` is the tuple of values at corresponding nodes in `rests`.

optree.**tree_transpose_map_with_accessor**(*func*, *tree*, **rests*, *inner_treespec=None*, *is_leaf=None*, *none_is_leaf=False*, *namespace=''*)

Map a multi-input function over pytree args as well as the tree accessors to produce a new pytree with transposed structure.

See also *tree_map_with_accessor()*, *tree_transpose_map()*, and *tree_transpose()*.

```
>>> tree = {'b': (2, [3, 4]), 'a': 1, 'c': (5, 6)}
>>> tree_transpose_map_with_accessor(
...     lambda a, x: {'depth': len(a), 'code': a.codegen('tree'), 'value': x},
...     tree,
```

(continues on next page)

(continued from previous page)

```
... )
{
    'depth': {
        'b': (2, [3, 3]),
        'a': 1,
        'c': (2, 2)
    },
    'code': {
        'b': ("tree['b'][0]", ["tree['b'][1][0]", "tree['b'][1][1]"]),
        'a': "tree['a']",
        'c': ("tree['c'][0]", "tree['c'][1]")
    },
    'value': {
        'b': (2, [3, 4]),
        'a': 1,
        'c': (5, 6)
    }
}
>>> tree_transpose_map_with_accessor(
...     lambda a, x: {'path': a.path, 'accessor': a, 'value': x},
...     tree,
...     inner_treespec=tree_structure({'path': 0, 'accessor': 0, 'value': 0}),
... )
{
    'path': {
        'b': (('b', 0), [('b', 1, 0), ('b', 1, 1)]),
        'a': ('a',),
        'c': (('c', 0), ('c', 1))
    },
    'accessor': {
        'b': (
            PyTreeAccessor(*['b'][0], (MappingEntry(key='b', type=<class 'dict'>),␣
↪SequenceEntry(index=0, type=<class 'tuple'>))),
            [
                PyTreeAccessor(*['b'][1][0], (MappingEntry(key='b', type=<class
↪'dict'>), SequenceEntry(index=1, type=<class 'tuple'>), SequenceEntry(index=0,␣
↪type=<class 'list'>))),
                PyTreeAccessor(*['b'][1][1], (MappingEntry(key='b', type=<class
↪'dict'>), SequenceEntry(index=1, type=<class 'tuple'>), SequenceEntry(index=1,␣
↪type=<class 'list'>)))
            ]
        ),
        'a': PyTreeAccessor(*['a'], (MappingEntry(key='a', type=<class 'dict'>),)),
        'c': (
            PyTreeAccessor(*['c'][0], (MappingEntry(key='c', type=<class 'dict'>),␣
↪SequenceEntry(index=0, type=<class 'tuple'>))),
            PyTreeAccessor(*['c'][1], (MappingEntry(key='c', type=<class 'dict'>),␣
↪SequenceEntry(index=1, type=<class 'tuple'>)))
        )
    },
    'value': {'b': (2, [3, 4]), 'a': 1, 'c': (5, 6)}
}
```

Parameters

- **func** (*callable*) – A function that takes `2 + len(rests)` arguments, to be applied at the corresponding leaves of the pytrees with extra accessors.
- **tree** (*pytree*) – A pytree to be mapped over, with each leaf providing the second positional argument and the corresponding path providing the first positional argument to function `func`.
- **rests** (*tuple of pytree*) – A tuple of pytrees, each of which has the same structure as `tree` or has `tree` as a prefix.
- **inner_treespec** (`PyTreeSpec`, *optional*) – The treespec object representing the inner structure of the result pytree. If not specified, the inner structure is inferred from the result of the function `func` on the first leaf. (default: `None`)
- **is_leaf** (*callable*, *optional*) – An optionally specified function that will be called at each flattening step. It should return a boolean, with `True` stopping the traversal and the whole subtree being treated as a leaf, and `False` indicating the flattening should traverse the current object.
- **none_is_leaf** (*bool*, *optional*) – Whether to treat `None` as a leaf. If `False`, `None` is a non-leaf node with arity 0. Thus `None` is contained in the treespec rather than in the leaves list and `None` will be remain in the result pytree. (default: `False`)
- **namespace** (*str*, *optional*) – The registry namespace used for custom pytree node types. (default: `''`, i.e., the global namespace)

Return type

`Union`[`TypeVar`(U), `Tuple`[PyTree[U], `...`], `List`[PyTree[U]], `Dict`[`Any`, PyTree[U]], `Deque`[PyTree[U]], *`CustomTreeNode`*[PyTree[U]]]

Returns

A new nested pytree with the same structure as `inner_treespec` but with the value at each leaf has the same structure as `tree`. The subtree at each leaf is given by the result of function `func(a, x, *xs)` where `(a, x)` are the accessor and value at the corresponding leaf in `tree` and `xs` is the tuple of values at corresponding nodes in `rests`.

optree.**tree_broadcast_prefix**(*prefix_tree*, *full_tree*, *is_leaf=None*, ***, *none_is_leaf=False*, *namespace=''*)

Return a pytree of same structure of `full_tree` with broadcasted subtrees in `prefix_tree`.

See also *broadcast_prefix()*, *tree_broadcast_common()*, and *treespec_is_prefix()*.

If a `prefix_tree` is a prefix of a `full_tree`, this means the `full_tree` can be constructed by replacing the leaves of `prefix_tree` with appropriate **subtrees**.

This function returns a pytree with the same size as `full_tree`. The leaves are replicated from `prefix_tree`. The number of replicas is determined by the corresponding subtree in `full_tree`.

```
>>> tree_broadcast_prefix(1, [2, 3, 4])
[1, 1, 1]
>>> tree_broadcast_prefix([1, 2, 3], [4, 5, 6])
[1, 2, 3]
>>> tree_broadcast_prefix([1, 2, 3], [4, 5, 6, 7])
Traceback (most recent call last):
    ...
ValueError: list arity mismatch; expected: 3, got: 4; list: [4, 5, 6, 7].
>>> tree_broadcast_prefix([1, 2, 3], [4, 5, (6, 7)])
[1, 2, (3, 3)]
```

(continues on next page)

(continued from previous page)

```
>>> tree_broadcast_prefix([1, 2, 3], [4, 5, {'a': 6, 'b': 7, 'c': (None, 8)}])
[1, 2, {'a': 3, 'b': 3, 'c': (None, 3)}]
>>> tree_broadcast_prefix([1, 2, 3], [4, 5, {'a': 6, 'b': 7, 'c': (None, 8)}], none_
↪is_leaf=True)
[1, 2, {'a': 3, 'b': 3, 'c': (3, 3)}]
```

Parameters

- **prefix_tree** (*pytree*) – A pytree with the prefix structure of `full_tree`.
- **full_tree** (*pytree*) – A pytree with the suffix structure of `prefix_tree`.
- **is_leaf** (*callable, optional*) – An optionally specified function that will be called at each flattening step. It should return a boolean, with `True` stopping the traversal and the whole subtree being treated as a leaf, and `False` indicating the flattening should traverse the current object.
- **none_is_leaf** (*bool, optional*) – Whether to treat `None` as a leaf. If `False`, `None` is a non-leaf node with arity 0. Thus `None` is contained in the treespec rather than in the leaves list and `None` will be remain in the result pytree. (default: `False`)
- **namespace** (*str, optional*) – The registry namespace used for custom pytree node types. (default: `''`, i.e., the global namespace)

Return type

`Union`[`TypeVar`(T), `Tuple`[PyTree[T], `...`], `List`[PyTree[T]], `Dict`[`Any`, PyTree[T]], `Deque`[PyTree[T]], *`CustomTreeNode`*[PyTree[T]]]

Returns

A pytree of same structure of `full_tree` with broadcasted subtrees in `prefix_tree`.

optree.**broadcast_prefix**(*prefix_tree*, *full_tree*, *is_leaf=None*, ***, *none_is_leaf=False*, *namespace=''*)

Return a list of broadcasted leaves in `prefix_tree` to match the number of leaves in `full_tree`.

See also *`tree_broadcast_prefix()`*, *`broadcast_common()`*, and *`treespec_is_prefix()`*.

If a `prefix_tree` is a prefix of a `full_tree`, this means the `full_tree` can be constructed by replacing the leaves of `prefix_tree` with appropriate **subtrees**.

This function returns a list of leaves with the same size as `full_tree`. The leaves are replicated from `prefix_tree`. The number of replicas is determined by the corresponding subtree in `full_tree`.

```
>>> broadcast_prefix(1, [2, 3, 4])
[1, 1, 1]
>>> broadcast_prefix([1, 2, 3], [4, 5, 6])
[1, 2, 3]
>>> broadcast_prefix([1, 2, 3], [4, 5, 6, 7])
Traceback (most recent call last):
    ...
ValueError: list arity mismatch; expected: 3, got: 4; list: [4, 5, 6, 7].
>>> broadcast_prefix([1, 2, 3], [4, 5, (6, 7)])
[1, 2, 3, 3]
>>> broadcast_prefix([1, 2, 3], [4, 5, {'a': 6, 'b': 7, 'c': (None, 8)}])
[1, 2, 3, 3, 3]
>>> broadcast_prefix([1, 2, 3], [4, 5, {'a': 6, 'b': 7, 'c': (None, 8)}], none_is_
↪leaf=True)
[1, 2, 3, 3, 3, 3]
```

Parameters

- **prefix_tree** (*pytree*) – A pytree with the prefix structure of `full_tree`.
- **full_tree** (*pytree*) – A pytree with the suffix structure of `prefix_tree`.
- **is_leaf** (*callable, optional*) – An optionally specified function that will be called at each flattening step. It should return a boolean, with `True` stopping the traversal and the whole subtree being treated as a leaf, and `False` indicating the flattening should traverse the current object.
- **none_is_leaf** (*bool, optional*) – Whether to treat `None` as a leaf. If `False`, `None` is a non-leaf node with arity 0. Thus `None` is contained in the treespec rather than in the leaves list and `None` will be remain in the result pytree. (default: `False`)
- **namespace** (*str, optional*) – The registry namespace used for custom pytree node types. (default: `''`, i.e., the global namespace)

Return type
`list`[`TypeVar`(T)]

Returns
A list of leaves in `prefix_tree` broadcasted to match the number of leaves in `full_tree`.

optree.**tree_broadcast_common**(*tree*, *other_tree*, *is_leaf=None*, ***, *none_is_leaf=False*, *namespace=''*)

Return two pytrees of common suffix structure of `tree` and `other_tree` with broadcasted subtrees.

See also *broadcast_common()*, *tree_broadcast_prefix()*, and *treespec_is_prefix()*.

If a `suffix_tree` is a suffix of a `tree`, this means the `suffix_tree` can be constructed by replacing the leaves of `tree` with appropriate **subtrees**.

This function returns two pytrees with the same structure. The tree structure is the common suffix structure of `tree` and `other_tree`. The leaves are replicated from `tree` and `other_tree`. The number of replicas is determined by the corresponding subtree in the suffix structure.

```
>>> tree_broadcast_common(1, [2, 3, 4])
([1, 1, 1], [2, 3, 4])
>>> tree_broadcast_common([1, 2, 3], [4, 5, 6])
([1, 2, 3], [4, 5, 6])
>>> tree_broadcast_common([1, 2, 3], [4, 5, 6, 7])
Traceback (most recent call last):
    ...
ValueError: list arity mismatch; expected: 3, got: 4.
>>> tree_broadcast_common([1, (2, 3), 4], [5, 6, (7, 8)])
([1, (2, 3), (4, 4)], [5, (6, 6), (7, 8)])
>>> tree_broadcast_common([1, {'a': (2, 3)}, 4], [5, 6, {'a': 7, 'b': 8, 'c': (None,
↪ 9)}])
([1, {'a': (2, 3)}, {'a': 4, 'b': 4, 'c': (None, 4)}],
 [5, {'a': (6, 6)}, {'a': 7, 'b': 8, 'c': (None, 9)}])
>>> tree_broadcast_common([1, {'a': (2, 3)}, 4], [5, 6, {'a': 7, 'b': 8, 'c': (None,
↪ 9)}], none_is_leaf=True)
([1, {'a': (2, 3)}, {'a': 4, 'b': 4, 'c': (4, 4)}],
 [5, {'a': (6, 6)}, {'a': 7, 'b': 8, 'c': (None, 9)}])
>>> tree_broadcast_common([1, None], [None, 2])
([None, None], [None, None])
>>> tree_broadcast_common([1, None], [None, 2], none_is_leaf=True)
([1, None], [None, 2])
```

Parameters

- **tree** (*pytree*) – A pytree has a common suffix structure of `other_tree`.
- **other_tree** (*pytree*) – A pytree has a common suffix structure of `tree`.
- **is_leaf** (*callable, optional*) – An optionally specified function that will be called at each flattening step. It should return a boolean, with `True` stopping the traversal and the whole subtree being treated as a leaf, and `False` indicating the flattening should traverse the current object.
- **none_is_leaf** (*bool, optional*) – Whether to treat `None` as a leaf. If `False`, `None` is a non-leaf node with arity 0. Thus `None` is contained in the treespec rather than in the leaves list and `None` will be remain in the result pytree. (default: `False`)
- **namespace** (*str, optional*) – The registry namespace used for custom pytree node types. (default: `''`, i.e., the global namespace)

Return type

tuple[Union[TypeVar(T), Tuple[Union[TypeVar(T), Tuple[PyTree[T], ...], List[PyTree[T]], Dict[Any, PyTree[T]], Deque[PyTree[T]], *CustomTreeNode*[PyTree[T]]], ...], List[Union[TypeVar(T), Tuple[PyTree[T], ...], List[PyTree[T]], Dict[Any, PyTree[T]], Deque[PyTree[T]], *CustomTreeNode*[PyTree[T]]]], Dict[Any, Union[TypeVar(T), Tuple[PyTree[T], ...], List[PyTree[T]], Dict[Any, PyTree[T]], Deque[PyTree[T]], *CustomTreeNode*[PyTree[T]]]], Deque[Union[TypeVar(T), Tuple[PyTree[T], ...], List[PyTree[T]], Dict[Any, PyTree[T]], Deque[PyTree[T]], *CustomTreeNode*[PyTree[T]]]], *CustomTreeNode*[Union[TypeVar(T), Tuple[PyTree[T], ...], List[PyTree[T]], Dict[Any, PyTree[T]], Deque[PyTree[T]], *CustomTreeNode*[PyTree[T]]]]], Union[TypeVar(T), Tuple[Union[TypeVar(T), Tuple[PyTree[T], ...], List[PyTree[T]], Dict[Any, PyTree[T]], Deque[PyTree[T]], *CustomTreeNode*[PyTree[T]]], ...], List[Union[TypeVar(T), Tuple[PyTree[T], ...], List[PyTree[T]], Dict[Any, PyTree[T]], Deque[PyTree[T]], *CustomTreeNode*[PyTree[T]]]], Dict[Any, Union[TypeVar(T), Tuple[PyTree[T], ...], List[PyTree[T]], Dict[Any, PyTree[T]], Deque[PyTree[T]], *CustomTreeNode*[PyTree[T]]]], Deque[Union[TypeVar(T), Tuple[PyTree[T], ...], List[PyTree[T]], Dict[Any, PyTree[T]], Deque[PyTree[T]], *CustomTreeNode*[PyTree[T]]]], *CustomTreeNode*[Union[TypeVar(T), Tuple[PyTree[T], ...], List[PyTree[T]], Dict[Any, PyTree[T]], Deque[PyTree[T]], *CustomTreeNode*[PyTree[T]]]]]]

Returns

Two pytrees of common suffix structure of `tree` and `other_tree` with broadcasted subtrees.

optree.**broadcast_common**(*tree*, *other_tree*, *is_leaf=None*, ***, *none_is_leaf=False*, *namespace=''*)

Return two lists of leaves in `tree` and `other_tree` broadcasted to match the number of leaves in the common suffix structure.

See also *tree_broadcast_common()*, *broadcast_prefix()*, and *treespec_is_prefix()*.

If a `suffix_tree` is a suffix of a `tree`, this means the `suffix_tree` can be constructed by replacing the leaves of `tree` with appropriate **subtrees**.

This function returns two pytrees with the same structure. The tree structure is the common suffix structure of `tree` and `other_tree`. The leaves are replicated from `tree` and `other_tree`. The number of replicas is determined by the corresponding subtree in the suffix structure.

```
>>> broadcast_common(1, [2, 3, 4])
([1, 1, 1], [2, 3, 4])
>>> broadcast_common([1, 2, 3], [4, 5, 6])
```

(continues on next page)

(continued from previous page)

```
([1, 2, 3], [4, 5, 6])
>>> broadcast_common([1, 2, 3], [4, 5, 6, 7])
Traceback (most recent call last):
    ...
ValueError: list arity mismatch; expected: 3, got: 4.
>>> broadcast_common([1, (2, 3), 4], [5, 6, (7, 8)])
([1, 2, 3, 4, 4], [5, 6, 6, 7, 8])
>>> broadcast_common([1, {'a': (2, 3)}, 4], [5, 6, {'a': 7, 'b': 8, 'c': (None, 9)}
↪])
([1, 2, 3, 4, 4, 4], [5, 6, 6, 7, 8, 9])
>>> broadcast_common([1, {'a': (2, 3)}, 4], [5, 6, {'a': 7, 'b': 8, 'c': (None, 9)}
↪], none_is_leaf=True)
([1, 2, 3, 4, 4, 4, 4], [5, 6, 6, 7, 8, None, 9])
>>> broadcast_common([1, None], [None, 2])
([], [])
>>> broadcast_common([1, None], [None, 2], none_is_leaf=True)
([1, None], [None, 2])
```

Parameters

- **tree** (*pytree*) – A pytree has a common suffix structure of `other_tree`.
- **other_tree** (*pytree*) – A pytree has a common suffix structure of `tree`.
- **is_leaf** (*callable, optional*) – An optionally specified function that will be called at each flattening step. It should return a boolean, with `True` stopping the traversal and the whole subtree being treated as a leaf, and `False` indicating the flattening should traverse the current object.
- **none_is_leaf** (*bool, optional*) – Whether to treat `None` as a leaf. If `False`, `None` is a non-leaf node with arity 0. Thus `None` is contained in the treespec rather than in the leaves list and `None` will be remain in the result pytree. (default: `False`)
- **namespace** (*str, optional*) – The registry namespace used for custom pytree node types. (default: `''`, i.e., the global namespace)

Return type

`tuple`[`list`[`TypeVar`(T)], `list`[`TypeVar`(T)]]

Returns

Two lists of leaves in `tree` and `other_tree` broadcasted to match the number of leaves in the common suffix structure.

optree.**tree_broadcast_map**(*func*, *tree*, **rests*, *is_leaf=None*, *none_is_leaf=False*, *namespace=''*)

Map a multi-input function over pytree args to produce a new pytree.

See also *tree_broadcast_map_with_path()*, *tree_map()*, *tree_map_()*, and *tree_map_with_path()*.

If only one input is provided, this function is the same as *tree_map()*:

```
>>> tree_broadcast_map(lambda x: x + 1, {'x': 7, 'y': (42, 64)})
{'x': 8, 'y': (43, 65)}
>>> tree_broadcast_map(lambda x: x + 1, {'x': 7, 'y': (42, 64), 'z': None})
{'x': 8, 'y': (43, 65), 'z': None}
>>> tree_broadcast_map(lambda x: x is None, {'x': 7, 'y': (42, 64), 'z': None})
{'x': False, 'y': (False, False), 'z': None}
```

(continues on next page)

(continued from previous page)

```
>>> tree_broadcast_map(lambda x: x is None, {'x': 7, 'y': (42, 64), 'z': None},␣
↪none_is_leaf=True)
{'x': False, 'y': (False, False), 'z': True}
```

If multiple inputs are given, all input trees will be broadcasted to the common suffix structure of all inputs:

```
>>> tree_broadcast_map(lambda x, y: x * y, [5, 6, (3, 4)], [{'a': 7, 'b': 9}, [1,␣
↪2], 8])
[{'a': 35, 'b': 45}, [6, 12], (24, 32)]
```

Parameters

- **func** (*callable*) – A function that takes `1 + len(rests)` arguments, to be applied at the corresponding leaves of the pytrees.
- **tree** (*pytree*) – A pytree to be mapped over, with each leaf providing the first positional argument to function `func`.
- **rests** (*tuple of pytree*) – A tuple of pytrees, they should have a common suffix structure with each other and with `tree`.
- **is_leaf** (*callable, optional*) – An optionally specified function that will be called at each flattening step. It should return a boolean, with `True` stopping the traversal and the whole subtree being treated as a leaf, and `False` indicating the flattening should traverse the current object.
- **none_is_leaf** (*bool, optional*) – Whether to treat `None` as a leaf. If `False`, `None` is a non-leaf node with arity 0. Thus `None` is contained in the treespec rather than in the leaves list and `None` will be remain in the result pytree. (default: `False`)
- **namespace** (*str, optional*) – The registry namespace used for custom pytree node types. (default: `''`, i.e., the global namespace)

Return type

`Union`[`TypeVar`(U), `Tuple`[PyTree[U], `...`], `List`[PyTree[U]], `Dict`[`Any`, PyTree[U]], `Deque`[PyTree[U]], *`CustomTreeNode`*[PyTree[U]]]

Returns

A new pytree with the structure as the common suffix structure of `tree` and `rests` but with the value at each leaf given by `func(x, *xs)` where `x` is the value at the corresponding leaf (may be broadcasted) in `tree` and `xs` is the tuple of values at corresponding leaves (may be broadcasted) in `rests`.

optree.**tree_broadcast_map_with_path**(*func*, *tree*, **rests*, *is_leaf=None*, *none_is_leaf=False*, *namespace=''*)

Map a multi-input function over pytree args as well as the tree paths to produce a new pytree.

See also *tree_broadcast_map()*, *tree_map()*, *tree_map_()*, and *tree_map_with_path()*.

If only one input is provided, this function is the same as *tree_map()*:

```
>>> tree_broadcast_map_with_path(lambda p, x: (len(p), x), {'x': 7, 'y': (42, 64)})
{'x': (1, 7), 'y': ((2, 42), (2, 64))}
>>> tree_broadcast_map_with_path(lambda p, x: x + len(p), {'x': 7, 'y': (42, 64), 'z
↪': None})
{'x': 8, 'y': (44, 66), 'z': None}
>>> tree_broadcast_map_with_path(lambda p, x: p, {'x': 7, 'y': (42, 64), 'z': {1.5:␣
↪None}})
```

(continues on next page)

(continued from previous page)

```
{'x': ('x',), 'y': (('y', 0), ('y', 1)), 'z': {1.5: None}}
>>> tree_broadcast_map_with_path(lambda p, x: p, {'x': 7, 'y': (42, 64), 'z': {1.5:␣
↪None}}, none_is_leaf=True)
{'x': ('x',), 'y': (('y', 0), ('y', 1)), 'z': {1.5: ('z', 1.5)}}
```

If multiple inputs are given, all input trees will be broadcasted to the common suffix structure of all inputs:

```
>>> tree_broadcast_map_with_path(
...     lambda p, x, y: (p, x * y),
...     [5, 6, (3, 4)],
...     [{'a': 7, 'b': 9}, [1, 2], 8],
... )
[
    {'a': ((0, 'a'), 35), 'b': ((0, 'b'), 45)},
    [((1, 0), 6), ((1, 1), 12)],
    (((2, 0), 24), ((2, 1), 32))
]
```

Parameters

- **func** (*callable*) – A function that takes `2 + len(rests)` arguments, to be applied at the corresponding leaves of the pytrees with extra paths.
- **tree** (*pytree*) – A pytree to be mapped over, with each leaf providing the first positional argument to function `func`.
- **rests** (*tuple of pytree*) – A tuple of pytrees, they should have a common suffix structure with each other and with `tree`.
- **is_leaf** (*callable, optional*) – An optionally specified function that will be called at each flattening step. It should return a boolean, with `True` stopping the traversal and the whole subtree being treated as a leaf, and `False` indicating the flattening should traverse the current object.
- **none_is_leaf** (*bool, optional*) – Whether to treat `None` as a leaf. If `False`, `None` is a non-leaf node with arity 0. Thus `None` is contained in the treespec rather than in the leaves list and `None` will be remain in the result pytree. (default: `False`)
- **namespace** (*str, optional*) – The registry namespace used for custom pytree node types. (default: `''`, i.e., the global namespace)

Return type

`Union`[`TypeVar`(U), `Tuple`[PyTree[U], `...`], `List`[PyTree[U]], `Dict`[`Any`, PyTree[U]], `Deque`[PyTree[U]], *`CustomTreeNode`*[PyTree[U]]]

Returns

A new pytree with the structure as the common suffix structure of `tree` and `rests` but with the value at each leaf given by `func(p, x, *xs)` where `(p, x)` are the path and value at the corresponding leaf (may be broadcasted) in and `xs` is the tuple of values at corresponding leaves (may be broadcasted) in `rests`.

optree.**tree_broadcast_map_with_accessor**(*func*, *tree*, **rests*, *is_leaf=None*, *none_is_leaf=False*, *namespace=''*)

Map a multi-input function over pytree args as well as the tree accessors to produce a new pytree.

See also *tree_broadcast_map()*, *tree_map()*, *tree_map_()*, and *tree_map_with_accessor()*.

If only one input is provided, this function is the same as *tree_map()*:

```
>>> tree_broadcast_map_with_accessor(lambda a, x: (len(a), x), {'x': 7, 'y': (42,␣
↪64)})
{'x': (1, 7), 'y': ((2, 42), (2, 64))}
>>> tree_broadcast_map_with_accessor(lambda a, x: x + len(a), {'x': 7, 'y': (42,␣
↪64), 'z': None})
{'x': 8, 'y': (44, 66), 'z': None}
>>> tree_broadcast_map_with_accessor(
...     lambda a, x: a.codegen('tree'),
...     {'x': 7, 'y': (42, 64), 'z': {1.5: None}},
... )
{
    'x': "tree['x']",
    'y': ("tree['y'][0]", "tree['y'][1]"),
    'z': {1.5: None}
}
>>> tree_broadcast_map_with_accessor(
...     lambda a, x: a.codegen('tree'),
...     {'x': 7, 'y': (42, 64), 'z': {1.5: None}},
...     none_is_leaf=True,
... )
{
    'x': "tree['x']",
    'y': ("tree['y'][0]", "tree['y'][1]"),
    'z': {1.5: "tree['z'][1.5]"}
}
```

If multiple inputs are given, all input trees will be broadcasted to the common suffix structure of all inputs:

```
>>> tree_broadcast_map_with_accessor(
...     lambda a, x, y: f'{a.codegen("tree")} = {x * y}',
...     [5, 6, (3, 4)],
...     [{'a': 7, 'b': 9}, [1, 2], 8],
... )
[
    {'a': "tree[0]['a'] = 35", 'b': "tree[0]['b'] = 45"},
    ['tree[1][0] = 6', 'tree[1][1] = 12'],
    ('tree[2][0] = 24', 'tree[2][1] = 32')
]
```

Parameters

- **func** (*callable*) – A function that takes `2 + len(rests)` arguments, to be applied at the corresponding leaves of the pytrees with extra accessors.
- **tree** (*pytree*) – A pytree to be mapped over, with each leaf providing the first positional argument to function `func`.
- **rests** (*tuple of pytree*) – A tuple of pytrees, they should have a common suffix structure with each other and with `tree`.
- **is_leaf** (*callable, optional*) – An optionally specified function that will be called at each flattening step. It should return a boolean, with `True` stopping the traversal and the whole subtree being treated as a leaf, and `False` indicating the flattening should traverse the current object.

- **none_is_leaf** (*bool, optional*) – Whether to treat `None` as a leaf. If `False`, `None` is a non-leaf node with arity 0. Thus `None` is contained in the treespec rather than in the leaves list and `None` will be remain in the result pytree. (default: `False`)
- **namespace** (*str, optional*) – The registry namespace used for custom pytree node types. (default: `''`, i.e., the global namespace)

Return type

`Union`[`TypeVar`(U), `Tuple`[PyTree[U], `...`], `List`[PyTree[U]], `Dict`[`Any`, PyTree[U]], `Deque`[PyTree[U]], *`CustomTreeNode`*[PyTree[U]]]

Returns

A new pytree with the structure as the common suffix structure of `tree` and `rests` but with the value at each leaf given by `func(a, x, *xs)` where `(a, x)` are the accessor and value at the corresponding leaf (may be broadcasted) in and `xs` is the tuple of values at corresponding leaves (may be broadcasted) in `rests`.

optree.**tree_flatten_one_level**(*tree*, *is_leaf=None*, ***, *none_is_leaf=False*, *namespace=''*)

Flatten the pytree one level, returning a 4-tuple of children, auxiliary data, path entries, and an unflatten function.

See also *tree_flatten()*, *tree_flatten_with_path()*.

```
>>> children, metadata, entries, unflatten_func = tree_flatten_one_level({'b': (2,␣
↪[3, 4]), 'a': 1, 'c': None, 'd': 5})
>>> children, metadata, entries
([1, (2, [3, 4]), None, 5], ['a', 'b', 'c', 'd'], ('a', 'b', 'c', 'd'))
>>> unflatten_func(metadata, children)
{'a': 1, 'b': (2, [3, 4]), 'c': None, 'd': 5}
>>> children, metadata, entries, unflatten_func = tree_flatten_one_level([{'a': 1,
↪'b': (2, 3)}, (4, 5)])
>>> children, metadata, entries
([{'a': 1, 'b': (2, 3)}, (4, 5)], None, (0, 1))
>>> unflatten_func(metadata, children)
[{'a': 1, 'b': (2, 3)}, (4, 5)]
```

Parameters

- **tree** (*pytree*) – A pytree to be traversed.
- **is_leaf** (*callable, optional*) – An optionally specified function that will be called at each flattening step. It should return a boolean, with `True` stopping the traversal and the whole subtree being treated as a leaf, and `False` indicating the flattening should traverse the current object.
- **none_is_leaf** (*bool, optional*) – Whether to treat `None` as a leaf. If `False`, `None` is a non-leaf node with arity 0. Thus `None` is contained in the treespec rather than in the leaves list and `None` will be remain in the result pytree. (default: `False`)
- **namespace** (*str, optional*) – The registry namespace used for custom pytree node types. (default: `''`, i.e., the global namespace)

Return type

`tuple`[`list`[`Union`[`TypeVar`(T), `Tuple`[`Union`[`TypeVar`(T), `Tuple`[PyTree[T], `...`], `List`[PyTree[T]], `Dict`[`Any`, PyTree[T]], `Deque`[PyTree[T]], *`CustomTreeNode`*[PyTree[T]]], `...`], `List`[`Union`[`TypeVar`(T), `Tuple`[PyTree[T], `...`], `List`[PyTree[T]], `Dict`[`Any`, PyTree[T]], `Deque`[PyTree[T]], *`CustomTreeNode`*[PyTree[T]]]], `Dict`[`Any`, `Union`[`TypeVar`(T), `Tuple`[PyTree[T], `...`], `List`[PyTree[T]], `Dict`[`Any`, PyTree[T]], `Deque`[PyTree[T]], *`CustomTreeNode`*[PyTree[T]]]],

Deque[Union[TypeVar(T), Tuple[PyTree[T], ...], List[PyTree[T]], Dict[Any, PyTree[T]], Deque[PyTree[T]], CustomTreeNode[PyTree[T]]]], CustomTreeNode[Union[TypeVar(T), Tuple[PyTree[T], ...], List[PyTree[T]], Dict[Any, PyTree[T]], Deque[PyTree[T]], CustomTreeNode[PyTree[T]]]]]], Optional[TypeVar(_MetaData, bound= Hashable)], tuple[Any, ...], Callable[[Optional[TypeVar(_MetaData, bound= Hashable)], list[Union[TypeVar(T), Tuple[Union[TypeVar(T), Tuple[PyTree[T], ...], List[PyTree[T]], Dict[Any, PyTree[T]], Deque[PyTree[T]], CustomTreeNode[PyTree[T]]], ...], List[Union[TypeVar(T), Tuple[PyTree[T], ...], List[PyTree[T]], Dict[Any, PyTree[T]], Deque[PyTree[T]], CustomTreeNode[PyTree[T]]]], Dict[Any, Union[TypeVar(T), Tuple[PyTree[T], ...], List[PyTree[T]], Dict[Any, PyTree[T]], Deque[PyTree[T]], CustomTreeNode[PyTree[T]]]], Deque[Union[TypeVar(T), Tuple[PyTree[T], ...], List[PyTree[T]], Dict[Any, PyTree[T]], Deque[PyTree[T]], CustomTreeNode[PyTree[T]]]], CustomTreeNode[Union[TypeVar(T), Tuple[PyTree[T], ...], List[PyTree[T]], Dict[Any, PyTree[T]], Deque[PyTree[T]], CustomTreeNode[PyTree[T]]]]]], Union[TypeVar(T), Tuple[Union[TypeVar(T), Tuple[PyTree[T], ...], List[PyTree[T]], Dict[Any, PyTree[T]], Deque[PyTree[T]], CustomTreeNode[PyTree[T]]], ...], List[Union[TypeVar(T), Tuple[PyTree[T], ...], List[PyTree[T]], Dict[Any, PyTree[T]], Deque[PyTree[T]], CustomTreeNode[PyTree[T]]]], Dict[Any, Union[TypeVar(T), Tuple[PyTree[T], ...], List[PyTree[T]], Dict[Any, PyTree[T]], Deque[PyTree[T]], CustomTreeNode[PyTree[T]]]], Deque[Union[TypeVar(T), Tuple[PyTree[T], ...], List[PyTree[T]], Dict[Any, PyTree[T]], Deque[PyTree[T]], CustomTreeNode[PyTree[T]]]], CustomTreeNode[Union[TypeVar(T), Tuple[PyTree[T], ...], List[PyTree[T]], Dict[Any, PyTree[T]], Deque[PyTree[T]], CustomTreeNode[PyTree[T]]]]]]]

Returns

A 4-tuple `(children, metadata, entries, unflatten_func)`. The first element is a list of one-level children of the pytree node. The second element is the auxiliary data used to reconstruct the pytree node. The third element is a tuple of path entries to the children. The fourth element is a function that can be used to unflatten the auxiliary data and children back to the pytree node.

optree.**prefix_errors**(*prefix_tree*, *full_tree*, *is_leaf=None*, ***, *none_is_leaf=False*, *namespace=''*)

Return a list of errors that would be raised by *broadcast_prefix()*.

Return type

list[Callable[[str], ValueError]]

1.3 Tree Reduce Functions

tree_reduce(func, tree[, initial, is_leaf, ...])	Traversal through a pytree and reduce the leaves in left-to-right depth-first order.
tree_sum(tree[, start, is_leaf, ...])	Sum `start` and leaf values in `tree` in left-to-right depth-first order and return the total.
tree_max(tree, *[, default, key, is_leaf, ...])	Return the maximum leaf value in `tree`.
tree_min(tree, *[, default, key, is_leaf, ...])	Return the minimum leaf value in `tree`.
tree_all(tree, *[, is_leaf, none_is_leaf, ...])	Test whether all leaves in `tree` are true (or if `tree` is empty).
tree_any(tree, *[, is_leaf, none_is_leaf, ...])	Test whether all leaves in `tree` are true (or `False` if `tree` is empty).

optree.**tree_reduce**(*func*, *tree*, *initial=<MISSING>*, *, *is_leaf=None*, *none_is_leaf=False*, *namespace=''*)

Traversal through a pytree and reduce the leaves in left-to-right depth-first order.

See also *tree_leaves()* and *tree_sum()*.

```
>>> tree_reduce(lambda x, y: x + y, {'x': 1, 'y': (2, 3)})
6
>>> tree_reduce(lambda x, y: x + y, {'x': 1, 'y': (2, None), 'z': 3})  # `None` is a␣
↪non-leaf node with arity 0 by default
6
>>> tree_reduce(lambda x, y: x and y, {'x': 1, 'y': (2, None), 'z': 3})
3
>>> tree_reduce(lambda x, y: x and y, {'x': 1, 'y': (2, None), 'z': 3}, none_is_
↪leaf=True)
None
```

Parameters

- **func** (*callable*) – A function that takes two arguments and returns a value of the same type.
- **tree** (*pytree*) – A pytree to be traversed.
- **initial** (*object, optional*) – An initial value to be used for the reduction. If not provided, the first leaf value is used as the initial value.
- **is_leaf** (*callable, optional*) – An optionally specified function that will be called at each flattening step. It should return a boolean, with `True` stopping the traversal and the whole subtree being treated as a leaf, and `False` indicating the flattening should traverse the current object.
- **none_is_leaf** (*bool, optional*) – Whether to treat `None` as a leaf. If `False`, `None` is a non-leaf node with arity 0. Thus `None` is contained in the treespec rather than in the leaves list and `None` will be remain in the result pytree. (default: `False`)
- **namespace** (*str, optional*) – The registry namespace used for custom pytree node types. (default: `''`, i.e., the global namespace)

Return type

`TypeVar`(T)

Returns

The result of reducing the leaves of the pytree using `func`.

optree.**tree_sum**(*tree*, *start=0*, *, *is_leaf=None*, *none_is_leaf=False*, *namespace=''*)

Sum `start` and leaf values in `tree` in left-to-right depth-first order and return the total.

See also *tree_leaves()* and *tree_reduce()*.

```
>>> tree_sum({'x': 1, 'y': (2, 3)})
6
>>> tree_sum({'x': 1, 'y': (2, None), 'z': 3})  # `None` is a non-leaf node with␣
↪arity 0 by default
6
>>> tree_sum({'x': 1, 'y': (2, None), 'z': 3}, none_is_leaf=True)
Traceback (most recent call last):
    ...
TypeError: unsupported operand type(s) for +: 'int' and 'NoneType'
```

(continues on next page)

(continued from previous page)

```
>>> tree_sum({'x': 'a', 'y': ('b', None), 'z': 'c'}, start='')
'abc'
>>> tree_sum({'x': [1], 'y': ([2], [None]), 'z': [3]}, start=[], is_leaf=lambda x:␣
↪isinstance(x, list))
[1, 2, None, 3]
```

Parameters

- **tree** (*pytree*) – A pytree to be traversed.
- **start** (*object, optional*) – An initial value to be used for the sum. (default: `0`)
- **is_leaf** (*callable, optional*) – An optionally specified function that will be called at each flattening step. It should return a boolean, with `True` stopping the traversal and the whole subtree being treated as a leaf, and `False` indicating the flattening should traverse the current object.
- **none_is_leaf** (*bool, optional*) – Whether to treat `None` as a leaf. If `False`, `None` is a non-leaf node with arity 0. Thus `None` is contained in the treespec rather than in the leaves list and `None` will be remain in the result pytree. (default: `False`)
- **namespace** (*str, optional*) – The registry namespace used for custom pytree node types. (default: `''`, i.e., the global namespace)

Return type
`TypeVar`(T)

Returns
The total sum of `start` and leaf values in `tree`.

optree.**tree_max**(*tree*, *, *default=<MISSING>*, *key=None*, *is_leaf=None*, *none_is_leaf=False*, *namespace=''*)

Return the maximum leaf value in `tree`.

See also *tree_leaves()* and *tree_min()*.

```
>>> tree_max({})
Traceback (most recent call last):
    ...
ValueError: max() iterable argument is empty
>>> tree_max({}, default=0)
0
>>> tree_max({'x': 0, 'y': (2, 1)})
2
>>> tree_max({'x': 0, 'y': (2, 1)}, key=lambda x: -x)
0
>>> tree_max({'a': None})  # `None` is a non-leaf node with arity 0 by default
Traceback (most recent call last):
    ...
ValueError: max() iterable argument is empty
>>> tree_max({'a': None}, default=0)  # `None` is a non-leaf node with arity 0 by␣
↪default
0
>>> tree_max({'a': None}, none_is_leaf=True)
None
>>> tree_max(None)  # `None` is a non-leaf node with arity 0 by default
```

(continues on next page)

(continued from previous page)

```
Traceback (most recent call last):
    ...
ValueError: max() iterable argument is empty
>>> tree_max(None, default=0)
0
>>> tree_max(None, none_is_leaf=True)
None
```

Parameters

- **tree** (*pytree*) – A pytree to be traversed.
- **default** (*object, optional*) – The default value to return if `tree` is empty. If the `tree` is empty and `default` is not specified, raise a `ValueError`.
- **key** (*callable or None, optional*) – An one argument ordering function like that used for `list.sort()`.
- **is_leaf** (*callable, optional*) – An optionally specified function that will be called at each flattening step. It should return a boolean, with `True` stopping the traversal and the whole subtree being treated as a leaf, and `False` indicating the flattening should traverse the current object.
- **none_is_leaf** (*bool, optional*) – Whether to treat `None` as a leaf. If `False`, `None` is a non-leaf node with arity 0. Thus `None` is contained in the treespec rather than in the leaves list and `None` will be remain in the result pytree. (default: `False`)
- **namespace** (*str, optional*) – The registry namespace used for custom pytree node types. (default: `''`, i.e., the global namespace)

Return type

`TypeVar`(T)

Returns

The maximum leaf value in `tree`.

optree.**tree_min**(*tree*, *, *default=<MISSING>*, *key=None*, *is_leaf=None*, *none_is_leaf=False*, *namespace=''*)

Return the minimum leaf value in `tree`.

See also *tree_leaves()* and *tree_max()*.

```
>>> tree_min({})
Traceback (most recent call last):
    ...
ValueError: min() iterable argument is empty
>>> tree_min({}, default=0)
0
>>> tree_min({'x': 0, 'y': (2, 1)})
0
>>> tree_min({'x': 0, 'y': (2, 1)}, key=lambda x: -x)
2
>>> tree_min({'a': None})  # `None` is a non-leaf node with arity 0 by default
Traceback (most recent call last):
    ...
ValueError: min() iterable argument is empty
>>> tree_min({'a': None}, default=0)  # `None` is a non-leaf node with arity 0 by␣
```

(continues on next page)

(continued from previous page)

```
↪default
0
>>> tree_min({'a': None}, none_is_leaf=True)
None
>>> tree_min(None)  # `None` is a non-leaf node with arity 0 by default
Traceback (most recent call last):
    ...
ValueError: min() iterable argument is empty
>>> tree_min(None, default=0)
0
>>> tree_min(None, none_is_leaf=True)
None
```

Parameters

- **tree** (*pytree*) – A pytree to be traversed.
- **default** (*object, optional*) – The default value to return if `tree` is empty. If the `tree` is empty and `default` is not specified, raise a `ValueError`.
- **key** (*callable or None, optional*) – An one argument ordering function like that used for `list.sort()`.
- **is_leaf** (*callable, optional*) – An optionally specified function that will be called at each flattening step. It should return a boolean, with `True` stopping the traversal and the whole subtree being treated as a leaf, and `False` indicating the flattening should traverse the current object.
- **none_is_leaf** (*bool, optional*) – Whether to treat `None` as a leaf. If `False`, `None` is a non-leaf node with arity 0. Thus `None` is contained in the treespec rather than in the leaves list and `None` will be remain in the result pytree. (default: `False`)
- **namespace** (*str, optional*) – The registry namespace used for custom pytree node types. (default: `''`, i.e., the global namespace)

Return type

`TypeVar`(T)

Returns

The minimum leaf value in `tree`.

optree.**tree_all**(*tree*, *, *is_leaf=None*, *none_is_leaf=False*, *namespace=''*)

Test whether all leaves in `tree` are true (or if `tree` is empty).

See also *tree_leaves()* and *tree_any()*.

```
>>> tree_all({})
True
>>> tree_all({'x': 1, 'y': (2, 3)})
True
>>> tree_all({'x': 1, 'y': (2, None), 'z': 3})  # `None` is a non-leaf node by␣
↪default
True
>>> tree_all({'x': 1, 'y': (2, None), 'z': 3}, none_is_leaf=True)
False
>>> tree_all(None)  # `None` is a non-leaf node by default
```

(continues on next page)

(continued from previous page)

```
True
>>> tree_all(None, none_is_leaf=True)
False
```

Parameters

- **tree** (*pytree*) – A pytree to be traversed.
- **is_leaf** (*callable, optional*) – An optionally specified function that will be called at each flattening step. It should return a boolean, with `True` stopping the traversal and the whole subtree being treated as a leaf, and `False` indicating the flattening should traverse the current object.
- **none_is_leaf** (*bool, optional*) – Whether to treat `None` as a leaf. If `False`, `None` is a non-leaf node with arity 0. Thus `None` is contained in the treespec rather than in the leaves list and `None` will be remain in the result pytree. (default: `False`)
- **namespace** (*str, optional*) – The registry namespace used for custom pytree node types. (default: `''`, i.e., the global namespace)

Return type
`bool`

Returns
`True` if all leaves in `tree` are true, or if `tree` is empty. Otherwise, `False`.

optree.**tree_any**(*tree*, *, *is_leaf=None*, *none_is_leaf=False*, *namespace=''*)

Test whether all leaves in `tree` are true (or `False` if `tree` is empty).

See also *tree_leaves()* and *tree_all()*.

```
>>> tree_any({})
False
>>> tree_any({'x': 0, 'y': (2, 0)})
True
>>> tree_any({'a': None})  # `None` is a non-leaf node with arity 0 by default
False
>>> tree_any({'a': None}, none_is_leaf=True)  # `None` is evaluated as false
False
>>> tree_any(None)  # `None` is a non-leaf node with arity 0 by default
False
>>> tree_any(None, none_is_leaf=True)  # `None` is evaluated as false
False
```

Parameters

- **tree** (*pytree*) – A pytree to be traversed.
- **is_leaf** (*callable, optional*) – An optionally specified function that will be called at each flattening step. It should return a boolean, with `True` stopping the traversal and the whole subtree being treated as a leaf, and `False` indicating the flattening should traverse the current object.
- **none_is_leaf** (*bool, optional*) – Whether to treat `None` as a leaf. If `False`, `None` is a non-leaf node with arity 0. Thus `None` is contained in the treespec rather than in the leaves list and `None` will be remain in the result pytree. (default: `False`)

- **namespace** (*str, optional*) – The registry namespace used for custom pytree node types. (default: `''`, i.e., the global namespace)

Return type
`bool`

Returns
`True` if any leaves in `tree` are true, otherwise, `False`. If `tree` is empty, return `False`.

1.4 PyTreeSpec Functions

treespec_paths(treespec)	Return a list of paths to the leaves of a treespec.
treespec_accessors(treespec)	Return a list of accessors to the leaves of a treespec.
treespec_entries(treespec)	Return a list of one-level entries of a treespec to its children.
treespec_entry(treespec, index)	Return the entry of a treespec at the given index.
treespec_children(treespec)	Return a list of treespecs for the children of a treespec.
treespec_child(treespec, index)	Return the treespec of the child of a treespec at the given index.
treespec_is_leaf(treespec[, strict])	Return whether the treespec is a leaf that has no children.
treespec_is_strict_leaf(treespec)	Return whether the treespec is a strict leaf.
treespec_is_prefix(treespec, other_treespec)	Return whether `treespec` is a prefix of `other_treespec`.
treespec_is_suffix(treespec, other_treespec)	Return whether `treespec` is a suffix of `other_treespec`.
treespec_leaf(*[, none_is_leaf, namespace])	Make a treespec representing a leaf node.
treespec_none(*[, none_is_leaf, namespace])	Make a treespec representing a `None` node.
treespec_tuple([iterable, none_is_leaf, ...])	Make a tuple treespec from a list of child treespecs.
treespec_list([iterable, none_is_leaf, ...])	Make a list treespec from a list of child treespecs.
treespec_dict([mapping, none_is_leaf, namespace])	Make a dict treespec from a dict of child treespecs.
treespec_namedtuple(namedtuple, *[, ...])	Make a namedtuple treespec from a namedtuple of child treespecs.
treespec_ordereddict([mapping, ...])	Make an OrderedDict treespec from an OrderedDict of child treespecs.
treespec_defaultdict([default_factory, ...])	Make a defaultdict treespec from a defaultdict of child treespecs.
treespec_deque([iterable, maxlen, ...])	Make a deque treespec from a deque of child treespecs.
treespec_structseq(structseq, *[, ...])	Make a PyStructSequence treespec from a PyStructSequence of child treespecs.
treespec_from_collection(collection, *[, ...])	Make a treespec from a collection of child treespecs.

optree.**treespec_paths**(*treespec*)

Return a list of paths to the leaves of a treespec.

See also *tree_flatten_with_path()*, *tree_paths()*, and *PyTreeSpec.paths()*.

Return type
`list[tuple[Any, ...]]`

optree.**treespec_accessors**(*treespec*)

Return a list of accessors to the leaves of a treespec.

See also *tree_flatten_with_accessor()*, *tree_accessors()* and *PyTreeSpec.accessors()*.

Return type
list[PyTreeAccessor]

optree.**treespec_entries**(*treespec*)

Return a list of one-level entries of a treespec to its children.

See also *treespec_entry()*, *treespec_paths()*, *treespec_children()*, and *PyTreeSpec.entries()*.

Return type
list[Any]

optree.**treespec_entry**(*treespec*, *index*)

Return the entry of a treespec at the given index.

See also *treespec_entries()*, *treespec_children()*, and *PyTreeSpec.entry()*.

Return type
Any

optree.**treespec_children**(*treespec*)

Return a list of treespecs for the children of a treespec.

See also *treespec_child()*, *treespec_paths()*, *treespec_entries()*, and *PyTreeSpec.children()*.

Return type
list[*PyTreeSpec*]

optree.**treespec_child**(*treespec*, *index*)

Return the treespec of the child of a treespec at the given index.

See also *treespec_children()*, *treespec_entries()*, and *PyTreeSpec.child()*.

Return type
PyTreeSpec

optree.**treespec_is_leaf**(*treespec*, *strict=True*)

Return whether the treespec is a leaf that has no children.

See also *treespec_is_strict_leaf()* and *PyTreeSpec.is_leaf()*.

This function is equivalent to `treespec.is_leaf(strict=strict)`. If `strict=False`, it will return `True` if and only if the treespec represents a strict leaf. If `strict=False`, it will return `True` if the treespec represents a strict leaf or `None` or an empty container (e.g., an empty tuple).

```
>>> treespec_is_leaf(tree_structure(1))
True
>>> treespec_is_leaf(tree_structure((1, 2)))
False
>>> treespec_is_leaf(tree_structure(None))
False
>>> treespec_is_leaf(tree_structure(None), strict=False)
True
>>> treespec_is_leaf(tree_structure(None, none_is_leaf=False))
False
>>> treespec_is_leaf(tree_structure(None, none_is_leaf=True))
```

(continues on next page)

(continued from previous page)

```
True
>>> treespec_is_leaf(tree_structure(()))
False
>>> treespec_is_leaf(tree_structure(()), strict=False)
True
>>> treespec_is_leaf(tree_structure([]))
False
>>> treespec_is_leaf(tree_structure([]), strict=False)
True
```

Parameters

- **treespec** (PyTreeSpec) – A treespec.
- **strict** (*bool, optional*) – Whether not to treat None or an empty container (e.g., an empty tuple) as a leaf. (default: True)

Return type
bool

Returns
True if the treespec represents a leaf that has no children, otherwise, False.

optree.**treespec_is_strict_leaf**(*treespec*)

Return whether the treespec is a strict leaf.

See also *treespec_is_leaf()* and *PyTreeSpec.is_leaf()*.

This function respects the none_is_leaf setting in the treespec. It is equivalent to treespec.is_leaf(strict=True). It will return True if and only if the treespec represents a strict leaf.

```
>>> treespec_is_strict_leaf(tree_structure(1))
True
>>> treespec_is_strict_leaf(tree_structure((1, 2)))
False
>>> treespec_is_strict_leaf(tree_structure(None))
False
>>> treespec_is_strict_leaf(tree_structure(None, none_is_leaf=False))
False
>>> treespec_is_strict_leaf(tree_structure(None, none_is_leaf=True))
True
>>> treespec_is_strict_leaf(tree_structure(()))
False
>>> treespec_is_strict_leaf(tree_structure([]))
False
```

Parameters
treespec (PyTreeSpec) – A treespec.

Return type
bool

Returns
True if the treespec represents a strict leaf, otherwise, False.

optree.**treespec_is_prefix**(*treespec*, *other_treespec*, *strict=False*)

Return whether `treespec` is a prefix of `other_treespec`.

See also *treespec_is_prefix()* and *PyTreeSpec.is_prefix()*.

Return type
`bool`

optree.**treespec_is_suffix**(*treespec*, *other_treespec*, *strict=False*)

Return whether `treespec` is a suffix of `other_treespec`.

See also *treespec_is_suffix()* *PyTreeSpec.is_suffix()*.

Return type
`bool`

optree.**treespec_leaf**(*, *none_is_leaf=False*, *namespace=''*)

Make a treespec representing a leaf node.

See also *tree_structure()*, *treespec_none()*, and *treespec_tuple()*.

```
>>> treespec_leaf()
PyTreeSpec(*)
>>> treespec_leaf(none_is_leaf=True)
PyTreeSpec(*, NoneIsLeaf)
>>> treespec_leaf(none_is_leaf=False) == treespec_leaf(none_is_leaf=True)
False
>>> treespec_leaf() == tree_structure(1)
True
>>> treespec_leaf(none_is_leaf=True) == tree_structure(1, none_is_leaf=True)
True
>>> treespec_leaf(none_is_leaf=True) == tree_structure(None, none_is_leaf=True)
True
>>> treespec_leaf(none_is_leaf=True) == tree_structure(None, none_is_leaf=False)
False
>>> treespec_leaf(none_is_leaf=True) == treespec_none(none_is_leaf=True)
True
>>> treespec_leaf(none_is_leaf=True) == treespec_none(none_is_leaf=False)
False
>>> treespec_leaf(none_is_leaf=False) == treespec_none(none_is_leaf=True)
False
>>> treespec_leaf(none_is_leaf=False) == treespec_none(none_is_leaf=False)
False
```

Parameters

- **none_is_leaf** (*bool, optional*) – Whether to treat `None` as a leaf. If `False`, `None` is a non-leaf node with arity 0. Thus `None` is contained in the treespec rather than in the leaves list and `None` will be remain in the result pytree. (default: `False`)
- **namespace** (*str, optional*) – The registry namespace used for custom pytree node types. (default: `''`, i.e., the global namespace)

Return type
PyTreeSpec

Returns
A treespec representing a leaf node.

optree.**treespec_none**(*, *none_is_leaf=False*, *namespace=''*)

Make a treespec representing a `None` node.

See also *tree_structure()*, *treespec_leaf()*, and *treespec_tuple()*.

```
>>> treespec_none()
PyTreeSpec(None)
>>> treespec_none(none_is_leaf=True)
PyTreeSpec(*, NoneIsLeaf)
>>> treespec_none(none_is_leaf=False) == treespec_none(none_is_leaf=True)
False
>>> treespec_none() == tree_structure(None)
True
>>> treespec_none() == tree_structure(1)
False
>>> treespec_none(none_is_leaf=True) == tree_structure(1, none_is_leaf=True)
True
>>> treespec_none(none_is_leaf=True) == tree_structure(None, none_is_leaf=True)
True
>>> treespec_none(none_is_leaf=True) == tree_structure(None, none_is_leaf=False)
False
>>> treespec_none(none_is_leaf=True) == treespec_leaf(none_is_leaf=True)
True
>>> treespec_none(none_is_leaf=False) == treespec_leaf(none_is_leaf=True)
False
>>> treespec_none(none_is_leaf=True) == treespec_leaf(none_is_leaf=False)
False
>>> treespec_none(none_is_leaf=False) == treespec_leaf(none_is_leaf=False)
False
```

Parameters

- **none_is_leaf** (*bool, optional*) – Whether to treat `None` as a leaf. If `False`, `None` is a non-leaf node with arity 0. Thus `None` is contained in the treespec rather than in the leaves list and `None` will be remain in the result pytree. (default: `False`)
- **namespace** (*str, optional*) – The registry namespace used for custom pytree node types. (default: `''`, i.e., the global namespace)

Return type

PyTreeSpec

Returns

A treespec representing a `None` node.

optree.**treespec_tuple**(*iterable=()*, *, *none_is_leaf=False*, *namespace=''*)

Make a tuple treespec from a list of child treespecs.

See also *tree_structure()*, *treespec_leaf()*, and *treespec_none()*.

```
>>> treespec_tuple([treespec_leaf(), treespec_leaf()])
PyTreeSpec((*, *))
>>> treespec_tuple([treespec_leaf(), treespec_leaf(), treespec_none()])
PyTreeSpec((*, *, None))
>>> treespec_tuple()
PyTreeSpec(())
```

(continues on next page)

(continued from previous page)

```
>>> treespec_tuple([treespec_leaf(), treespec_tuple([treespec_leaf(), treespec_
↪leaf()])])
PyTreeSpec((*, (*, *)))
>>> treespec_tuple([treespec_leaf(), tree_structure({'a': 1, 'b': 2})])
PyTreeSpec((*, {'a': *, 'b': *}))
>>> treespec_tuple([treespec_leaf(), tree_structure({'a': 1, 'b': 2}, none_is_
↪leaf=True)])
Traceback (most recent call last):
    ...
ValueError: Expected treespec(s) with `node_is_leaf=False`.
```

Parameters

- **iterable** (*iterable of PyTreeSpec, optional*) – A iterable of child treespecs. They must have the same `node_is_leaf` and `namespace` values.
- **none_is_leaf** (*bool, optional*) – Whether to treat `None` as a leaf. If `False`, `None` is a non-leaf node with arity 0. Thus `None` is contained in the treespec rather than in the leaves list and `None` will be remain in the result pytree. (default: `False`)
- **namespace** (*str, optional*) – The registry namespace used for custom pytree node types. (default: `''`, i.e., the global namespace)

Return type

PyTreeSpec

Returns

A treespec representing a tuple node with the given children.

optree.**treespec_list**(*iterable=()*, *, *none_is_leaf=False*, *namespace=''*)

Make a list treespec from a list of child treespecs.

See also *tree_structure()*, *treespec_leaf()*, and *treespec_none()*.

```
>>> treespec_list([treespec_leaf(), treespec_leaf()])
PyTreeSpec([*, *])
>>> treespec_list([treespec_leaf(), treespec_leaf(), treespec_none()])
PyTreeSpec([*, *, None])
>>> treespec_list()
PyTreeSpec([])
>>> treespec_list([treespec_leaf(), treespec_tuple([treespec_leaf(), treespec_
↪leaf()])])
PyTreeSpec([*, (*, *)])
>>> treespec_list([treespec_leaf(), tree_structure({'a': 1, 'b': 2})])
PyTreeSpec([*, {'a': *, 'b': *}])
>>> treespec_list([treespec_leaf(), tree_structure({'a': 1, 'b': 2}, none_is_
↪leaf=True)])
Traceback (most recent call last):
    ...
ValueError: Expected treespec(s) with `node_is_leaf=False`.
```

Parameters

- **iterable** (*iterable of PyTreeSpec, optional*) – A iterable of child treespecs. They must have the same `node_is_leaf` and `namespace` values.

- **none_is_leaf** (*bool, optional*) – Whether to treat `None` as a leaf. If `False`, `None` is a non-leaf node with arity 0. Thus `None` is contained in the treespec rather than in the leaves list and `None` will be remain in the result pytree. (default: `False`)
- **namespace** (*str, optional*) – The registry namespace used for custom pytree node types. (default: `''`, i.e., the global namespace)

Return type
PyTreeSpec

Returns
A treespec representing a list node with the given children.

optree.**treespec_dict**(*mapping=()*, *, *none_is_leaf=False*, *namespace=''*, ***kwargs*)

Make a dict treespec from a dict of child treespecs.

See also *tree_structure()*, *treespec_leaf()*, and *treespec_none()*.

```
>>> treespec_dict({'a': treespec_leaf(), 'b': treespec_leaf()})
PyTreeSpec({'a': *, 'b': *})
>>> treespec_dict([('b', treespec_leaf()), ('c', treespec_leaf()), ('a', treespec_
↪none())])
PyTreeSpec({'a': None, 'b': *, 'c': *})
>>> treespec_dict()
PyTreeSpec({})
>>> treespec_dict(a=treespec_leaf(), b=treespec_tuple([treespec_leaf(), treespec_
↪leaf()]))
PyTreeSpec({'a': *, 'b': (*, *)})
>>> treespec_dict({'a': treespec_leaf(), 'b': tree_structure([1, 2])})
PyTreeSpec({'a': *, 'b': [*, *]})
>>> treespec_dict({'a': treespec_leaf(), 'b': tree_structure([1, 2], none_is_
↪leaf=True)})
Traceback (most recent call last):
    ...
ValueError: Expected treespec(s) with `node_is_leaf=False`.
```

Parameters

- **mapping** (*mapping of PyTreeSpec, optional*) – A mapping of child treespecs. They must have the same `node_is_leaf` and `namespace` values.
- **none_is_leaf** (*bool, optional*) – Whether to treat `None` as a leaf. If `False`, `None` is a non-leaf node with arity 0. Thus `None` is contained in the treespec rather than in the leaves list and `None` will be remain in the result pytree. (default: `False`)
- **namespace** (*str, optional*) – The registry namespace used for custom pytree node types. (default: `''`, i.e., the global namespace)

Return type
PyTreeSpec

Returns
A treespec representing a dict node with the given children.

optree.**treespec_namedtuple**(*namedtuple*, *, *none_is_leaf=False*, *namespace=''*)

Make a namedtuple treespec from a namedtuple of child treespecs.

See also *tree_structure()*, *treespec_leaf()*, and *treespec_none()*.

```
>>> from collections import namedtuple
>>> Point = namedtuple('Point', ['x', 'y'])
>>> treespec_namedtuple(Point(x=treespec_leaf(), y=treespec_leaf()))
PyTreeSpec(Point(x=*, y=*))
>>> treespec_namedtuple(Point(x=treespec_leaf(), y=treespec_tuple([treespec_leaf(),␣
↪treespec_leaf()])))
PyTreeSpec(Point(x=*, y=(*, *)))
>>> treespec_namedtuple(Point(x=treespec_leaf(), y=tree_structure([1, 2])))
PyTreeSpec(Point(x=*, y=[*, *]))
>>> treespec_namedtuple(Point(x=treespec_leaf(), y=tree_structure([1, 2], none_is_
↪leaf=True)))
Traceback (most recent call last):
    ...
ValueError: Expected treespec(s) with `node_is_leaf=False`.
```

Parameters

- **namedtuple** (*namedtuple of* PyTreeSpec) – A namedtuple of child treespecs. They must have the same `node_is_leaf` and `namespace` values.
- **none_is_leaf** (*bool, optional*) – Whether to treat `None` as a leaf. If `False`, `None` is a non-leaf node with arity 0. Thus `None` is contained in the treespec rather than in the leaves list and `None` will be remain in the result pytree. (default: `False`)
- **namespace** (*str, optional*) – The registry namespace used for custom pytree node types. (default: `''`, i.e., the global namespace)

Returns

A treespec representing a dict node with the given children.

optree.**treespec_ordereddict**(*mapping=()*, *, *none_is_leaf=False*, *namespace=''*, ***kwargs*)

Make an OrderedDict treespec from an OrderedDict of child treespecs.

See also *tree_structure()*, *treespec_leaf()*, and *treespec_none()*.

```
>>> treespec_ordereddict({'a': treespec_leaf(), 'b': treespec_leaf()})
PyTreeSpec(OrderedDict({'a': *, 'b': *}))
>>> treespec_ordereddict([('b', treespec_leaf()), ('c', treespec_leaf()), ('a',␣
↪treespec_none())])
PyTreeSpec(OrderedDict({'b': *, 'c': *, 'a': None}))
>>> treespec_ordereddict()
PyTreeSpec(OrderedDict())
>>> treespec_ordereddict(a=treespec_leaf(), b=treespec_tuple([treespec_leaf(),␣
↪treespec_leaf()]))
PyTreeSpec(OrderedDict({'a': *, 'b': (*, *)}))
>>> treespec_ordereddict({'a': treespec_leaf(), 'b': tree_structure([1, 2])})
PyTreeSpec(OrderedDict({'a': *, 'b': [*, *]}))
>>> treespec_ordereddict({'a': treespec_leaf(), 'b': tree_structure([1, 2], none_is_
↪leaf=True)})
Traceback (most recent call last):
    ...
ValueError: Expected treespec(s) with `node_is_leaf=False`.
```

Parameters

- **mapping** (*mapping of* PyTreeSpec*, optional*) – A mapping of child treespecs. They must have the same `node_is_leaf` and `namespace` values.
- **none_is_leaf** (*bool, optional*) – Whether to treat `None` as a leaf. If `False`, `None` is a non-leaf node with arity 0. Thus `None` is contained in the treespec rather than in the leaves list and `None` will be remain in the result pytree. (default: `False`)
- **namespace** (*str, optional*) – The registry namespace used for custom pytree node types. (default: `''`, i.e., the global namespace)

Return type
PyTreeSpec

Returns
A treespec representing an OrderedDict node with the given children.

optree.**treespec_defaultdict**(*default_factory=None*, *mapping=()*, *, *none_is_leaf=False*, *namespace=''*, ***kwargs*)

Make a defaultdict treespec from a defaultdict of child treespecs.

See also *tree_structure()*, *treespec_leaf()*, and *treespec_none()*.

```
>>> treespec_defaultdict(int, {'a': treespec_leaf(), 'b': treespec_leaf()})
PyTreeSpec(defaultdict(<class 'int'>, {'a': *, 'b': *}))
>>> treespec_defaultdict(int, [('b', treespec_leaf()), ('c', treespec_leaf()), ('a',
↪ treespec_none())])
PyTreeSpec(defaultdict(<class 'int'>, {'a': None, 'b': *, 'c': *}))
>>> treespec_defaultdict()
PyTreeSpec(defaultdict(None, {}))
>>> treespec_defaultdict(int)
PyTreeSpec(defaultdict(<class 'int'>, {}))
>>> treespec_defaultdict(int, a=treespec_leaf(), b=treespec_tuple([treespec_leaf(),␣
↪treespec_leaf()]))
PyTreeSpec(defaultdict(<class 'int'>, {'a': *, 'b': (*, *)}))
>>> treespec_defaultdict(int, {'a': treespec_leaf(), 'b': tree_structure([1, 2])})
PyTreeSpec(defaultdict(<class 'int'>, {'a': *, 'b': [*, *]}))
>>> treespec_defaultdict(int, {'a': treespec_leaf(), 'b': tree_structure([1, 2],␣
↪none_is_leaf=True)})
Traceback (most recent call last):
    ...
ValueError: Expected treespec(s) with `node_is_leaf=False`.
```

Parameters

- **default_factory** (*callable or None, optional*) – A factory function that will be used to create a missing value. (default: `None`)
- **mapping** (*mapping of* PyTreeSpec*, optional*) – A mapping of child treespecs. They must have the same `node_is_leaf` and `namespace` values.
- **none_is_leaf** (*bool, optional*) – Whether to treat `None` as a leaf. If `False`, `None` is a non-leaf node with arity 0. Thus `None` is contained in the treespec rather than in the leaves list and `None` will be remain in the result pytree. (default: `False`)
- **namespace** (*str, optional*) – The registry namespace used for custom pytree node types. (default: `''`, i.e., the global namespace)

Return type
PyTreeSpec

Returns
A treespec representing a defaultdict node with the given children.

optree.**treespec_deque**(*iterable=()*, *maxlen=None*, ***, *none_is_leaf=False*, *namespace=''*)

Make a deque treespec from a deque of child treespecs.

See also *tree_structure()*, *treespec_leaf()*, and *treespec_none()*.

```
>>> treespec_deque([treespec_leaf(), treespec_leaf()])
PyTreeSpec(deque([*, *]))
>>> treespec_deque([treespec_leaf(), treespec_leaf(), treespec_none()], maxlen=5)
PyTreeSpec(deque([*, *, None], maxlen=5))
>>> treespec_deque()
PyTreeSpec(deque([]))
>>> treespec_deque([treespec_leaf(), treespec_tuple([treespec_leaf(), treespec_
↪leaf()])])
PyTreeSpec(deque([*, (*, *)]))
>>> treespec_deque([treespec_leaf(), tree_structure({'a': 1, 'b': 2})], maxlen=5)
PyTreeSpec(deque([*, {'a': *, 'b': *}], maxlen=5))
>>> treespec_deque([treespec_leaf(), tree_structure({'a': 1, 'b': 2}, none_is_
↪leaf=True)], maxlen=5)
Traceback (most recent call last):
    ...
ValueError: Expected treespec(s) with `node_is_leaf=False`.
```

Parameters

- **iterable** (*iterable of PyTreeSpec, optional*) – A iterable of child treespecs. They must have the same `node_is_leaf` and `namespace` values.
- **maxlen** (*int or None, optional*) – The maximum size of a deque or `None` if unbounded. (default: `None`)
- **none_is_leaf** (*bool, optional*) – Whether to treat `None` as a leaf. If `False`, `None` is a non-leaf node with arity 0. Thus `None` is contained in the treespec rather than in the leaves list and `None` will be remain in the result pytree. (default: `False`)
- **namespace** (*str, optional*) – The registry namespace used for custom pytree node types. (default: `''`, i.e., the global namespace)

Return type
PyTreeSpec

Returns
A treespec representing a deque node with the given children.

optree.**treespec_structseq**(*structseq*, ***, *none_is_leaf=False*, *namespace=''*)

Make a PyStructSequence treespec from a PyStructSequence of child treespecs.

See also *tree_structure()*, *treespec_leaf()*, and *treespec_none()*.

Parameters

- **structseq** (*PyStructSequence of PyTreeSpec*) – A PyStructSequence of child treespecs. They must have the same `node_is_leaf` and `namespace` values.

- **none_is_leaf** (*bool, optional*) – Whether to treat `None` as a leaf. If `False`, `None` is a non-leaf node with arity 0. Thus `None` is contained in the treespec rather than in the leaves list and `None` will be remain in the result pytree. (default: `False`)
- **namespace** (*str, optional*) – The registry namespace used for custom pytree node types. (default: `''`, i.e., the global namespace)

Return type
PyTreeSpec

Returns
A treespec representing a PyStructSequence node with the given children.

optree.**treespec_from_collection**(*collection*, *, *none_is_leaf=False*, *namespace=''*)

Make a treespec from a collection of child treespecs.

See also *tree_structure()*, *treespec_leaf()*, and *treespec_none()*.

```
>>> treespec_from_collection(None)
PyTreeSpec(None)
>>> treespec_from_collection(None, none_is_leaf=True)
PyTreeSpec(*, NoneIsLeaf)
>>> treespec_from_collection(object())
PyTreeSpec(*)
>>> treespec_from_collection([treespec_leaf(), treespec_none()])
PyTreeSpec([*, None])
>>> treespec_from_collection({'a': treespec_leaf(), 'b': treespec_none()})
PyTreeSpec({'a': *, 'b': None})
>>> treespec_from_collection(deque([treespec_leaf(), tree_structure({'a': 1, 'b': 2}
↪)], maxlen=5))
PyTreeSpec(deque([*, {'a': *, 'b': *}], maxlen=5))
>>> treespec_from_collection({'a': treespec_leaf(), 'b': (treespec_leaf(), treespec_
↪none())})
Traceback (most recent call last):
    ...
ValueError: Expected a(n) dict of PyTreeSpec(s), got {'a': PyTreeSpec(*), 'b':␣
↪(PyTreeSpec(*), PyTreeSpec(None))}.
>>> treespec_from_collection([treespec_leaf(), tree_structure({'a': 1, 'b': 2},␣
↪none_is_leaf=True)])
Traceback (most recent call last):
    ...
ValueError: Expected treespec(s) with `node_is_leaf=False`.
```

Parameters

- **collection** (*collection of* `PyTreeSpec`) – A collection of child treespecs. They must have the same `node_is_leaf` and `namespace` values.
- **none_is_leaf** (*bool, optional*) – Whether to treat `None` as a leaf. If `False`, `None` is a non-leaf node with arity 0. Thus `None` is contained in the treespec rather than in the leaves list and `None` will be remain in the result pytree. (default: `False`)
- **namespace** (*str, optional*) – The registry namespace used for custom pytree node types. (default: `''`, i.e., the global namespace)

Return type
PyTreeSpec

Returns

A treespec representing the same structure of the collection with the given children.

CHAPTER

TWO

PYTREE NODE REGISTRY

register_pytree_node(cls, flatten_func, ...)	Extend the set of types that are considered internal nodes in pytrees.
register_pytree_node_class([cls, ...])	Extend the set of types that are considered internal nodes in pytrees.
unregister_pytree_node(cls, *, namespace)	Remove a type from the pytree node registry.

optree.**register_pytree_node**(*cls*, *flatten_func*, *unflatten_func*, *, *path_entry_type=<class 'optree.accessor.AutoEntry'>*, *namespace*)

Extend the set of types that are considered internal nodes in pytrees.

See also *register_pytree_node_class()* and *unregister_pytree_node()*.

The `namespace` argument is used to avoid collisions that occur when different libraries register the same Python type with different behaviors. It is recommended to add a unique prefix to the namespace to avoid conflicts with other libraries. Namespaces can also be used to specify the same class in different namespaces for different use cases.

Warning: For safety reasons, a `namespace` must be specified while registering a custom type. It is used to isolate the behavior of flattening and unflattening a pytree node type. This is to prevent accidental collisions between different libraries that may register the same type.

Parameters

- **cls** (*type*) – A Python type to treat as an internal pytree node.
- **flatten_func** (*callable*) – A function to be used during flattening, taking an instance of `cls` and returning a triple or optionally a pair, with (1) an iterable for the children to be flattened recursively, and (2) some hashable auxiliary data to be stored in the treespec and to be passed to the `unflatten_func`, and (3) (optional) an iterable for the tree path entries to the corresponding children. If the entries are not provided or given by `None`, then *range(len(children))* will be used.
- **unflatten_func** (*callable*) – A function taking two arguments: the auxiliary data that was returned by `flatten_func` and stored in the treespec, and the unflattened children. The function should return an instance of `cls`.
- **path_entry_type** (*type, optional*) – The type of the path entry to be used in the treespec. (default: `AutoEntry`)

- **namespace** (*str*) – A non-empty string that uniquely identifies the namespace of the type registry. This is used to isolate the registry from other modules that might register a different custom behavior for the same type.

Return type
Type[*CustomTreeNode*[TypeVar(T)]]

Returns
The same type as the input `cls`.

Raises

- **TypeError** – If the input type is not a class.
- **TypeError** – If the path entry class is not a subclass of `PyTreeEntry`.
- **TypeError** – If the namespace is not a string.
- **ValueError** – If the namespace is an empty string.
- **ValueError** – If the type is already registered in the registry.

Examples

```
>>> # Registry a Python type with lambda functions
>>> register_pytree_node(
...     set,
...     lambda s: (sorted(s), None, None),
...     lambda _, children: set(children),
...     namespace='set',
... )
<class 'set'>
```

```
>>> # Register a Python type into a namespace
>>> import torch
>>> register_pytree_node(
...     torch.Tensor,
...     flatten_func=lambda tensor: (
...         (tensor.cpu().detach().numpy(),),
...         {'dtype': tensor.dtype, 'device': tensor.device, 'requires_grad':␣
↪tensor.requires_grad},
...     ),
...     unflatten_func=lambda metadata, children: torch.tensor(children[0],␣
↪**metadata),
...     namespace='torch2numpy',
... )
<class 'torch.Tensor'>
```

```
>>>
>>> tree = {'weight': torch.ones(size=(1, 2)).cuda(), 'bias': torch.zeros(size=(2,
↪))}
>>> tree
{'weight': tensor([[1., 1.]], device='cuda:0'), 'bias': tensor([0., 0.])}
```

```
>>> # Flatten without specifying the namespace
>>> tree_flatten(tree)  # `torch.Tensor`s are leaf nodes
([tensor([0., 0.]), tensor([[1., 1.]], device='cuda:0')], PyTreeSpec({'bias': *,
↪'weight': *}))
```

```
>>> # Flatten with the namespace
>>> tree_flatten(tree, namespace='torch2numpy')
(
    [array([0., 0.], dtype=float32), array([[1., 1.]], dtype=float32)],
    PyTreeSpec(
        {
            'bias': CustomTreeNode(Tensor[{'dtype': torch.float32, 'device':␣
↪device(type='cpu'), 'requires_grad': False}], [*]),
            'weight': CustomTreeNode(Tensor[{'dtype': torch.float32, 'device':␣
↪device(type='cuda', index=0), 'requires_grad': False}], [*])
        },
        namespace='torch2numpy'
    )
)
```

```
>>> # Register the same type with a different namespace for different behaviors
>>> def tensor2flatparam(tensor):
...     return [torch.nn.Parameter(tensor.reshape(-1))], tensor.shape, None
...
... def flatparam2tensor(metadata, children):
...     return children[0].reshape(metadata)
...
... register_pytree_node(
...     torch.Tensor,
...     flatten_func=tensor2flatparam,
...     unflatten_func=flatparam2tensor,
...     namespace='tensor2flatparam',
... )
<class 'torch.Tensor'>
```

```
>>> # Flatten with the new namespace
>>> tree_flatten(tree, namespace='tensor2flatparam')
(
    [
        Parameter containing: tensor([0., 0.], requires_grad=True),
        Parameter containing: tensor([1., 1.], device='cuda:0', requires_grad=True)
    ],
    PyTreeSpec(
        {
            'bias': CustomTreeNode(Tensor[torch.Size([2])], [*]),
            'weight': CustomTreeNode(Tensor[torch.Size([1, 2])], [*])
        },
        namespace='tensor2flatparam'
    )
)
```

optree.**register_pytree_node_class**(*cls=None*, ***, *path_entry_type=None*, *namespace=None*)

Extend the set of types that are considered internal nodes in pytrees.

See also *register_pytree_node()* and *unregister_pytree_node()*.

The `namespace` argument is used to avoid collisions that occur when different libraries register the same Python type with different behaviors. It is recommended to add a unique prefix to the namespace to avoid conflicts with other libraries. Namespaces can also be used to specify the same class in different namespaces for different use cases.

Warning: For safety reasons, a `namespace` must be specified while registering a custom type. It is used to isolate the behavior of flattening and unflattening a pytree node type. This is to prevent accidental collisions between different libraries that may register the same type.

Parameters

- **cls** (*type*, *optional*) – A Python type to treat as an internal pytree node.
- **path_entry_type** (*type*, *optional*) – The type of the path entry to be used in the treespec. (default: `AutoEntry`)
- **namespace** (*str*, *optional*) – A non-empty string that uniquely identifies the namespace of the type registry. This is used to isolate the registry from other modules that might register a different custom behavior for the same type.

Return type

Union[Type[*CustomTreeNode*[TypeVar(T)]], Callable[[Type[*CustomTreeNode*[TypeVar(T)]]], Type[*CustomTreeNode*[TypeVar(T)]]]]

Returns

The same type as the input `cls` if the argument presents. Otherwise, return a decorator function that registers the class as a pytree node.

Raises

- **TypeError** – If the path entry class is not a subclass of `PyTreeEntry`.
- **TypeError** – If the namespace is not a string.
- **ValueError** – If the namespace is an empty string.
- **ValueError** – If the type is already registered in the registry.

This function is a thin wrapper around *register_pytree_node()*, and provides a class-oriented interface:

```
@register_pytree_node_class(namespace='foo')
class Special:
    TREE_PATH_ENTRY_TYPE = GetAttrEntry

    def __init__(self, x, y):
        self.x = x
        self.y = y

    def tree_flatten(self):
        return ((self.x, self.y), None, ('x', 'y'))

    @classmethod
    def tree_unflatten(cls, metadata, children):
        return cls(*children)
```

(continues on next page)

(continued from previous page)

```
@register_pytree_node_class('mylist')
class MyList(UserList):
    TREE_PATH_ENTRY_TYPE = SequenceEntry

    def tree_flatten(self):
        return self.data, None, None

    @classmethod
    def tree_unflatten(cls, metadata, children):
        return cls(*children)
```

optree.**unregister_pytree_node**(*cls*, *, *namespace*)

Remove a type from the pytree node registry.

See also *register_pytree_node()* and *register_pytree_node_class()*.

This function is the inverse operation of function *register_pytree_node()*.

Parameters

- **cls** (*type*) – A Python type to remove from the pytree node registry.
- **namespace** (*str*) – The namespace of the pytree node registry to remove the type from.

Return type

PyTreeNodeRegistryEntry

Returns

The removed registry entry.

Raises

- **TypeError** – If the input type is not a class.
- **TypeError** – If the namespace is not a string.
- **ValueError** – If the namespace is an empty string.
- **ValueError** – If the type is a built-in type that cannot be unregistered.
- **ValueError** – If the type is not found in the registry.

Examples

```
>>> # Register a Python type with lambda functions
>>> register_pytree_node(
...     set,
...     lambda s: (sorted(s), None, None),
...     lambda _, children: set(children),
...     namespace='temp',
... )
<class 'set'>
```

```
>>> # Unregister the Python type
>>> unregister_pytree_node(set, namespace='temp')
```

CHAPTER

THREE

INTEGRATION WITH FUNCTOOLS

partial(func, *args, **keywords)	A version of functools.partial() that works in pytrees.
reduce(func, tree[, initial, is_leaf, ...])	Traversal through a pytree and reduce the leaves in left-to-right depth-first order.

class optree.functools.**partial**(*func: Callable[[...], Any]*, **args: T*, ***keywords: T*)

Bases: partial, *CustomTreeNode*[T]

A version of functools.partial() that works in pytrees.

Use it for partial function evaluation in a way that is compatible with transformations, e.g., `partial(func, *args, **kwargs)`.

(You need to explicitly opt-in to this behavior because we did not want to give functools.partial() different semantics than normal function closures.)

For example, here is a basic usage of *partial* in a manner similar to functools.partial():

```
>>> import operator
>>> import torch
>>> add_one = partial(operator.add, torch.ones(()))
>>> add_one(torch.tensor([[1, 2], [3, 4]]))
tensor([[2., 3.],
        [4., 5.]])
```

Pytree compatibility means that the resulting partial function can be passed as an argument within tree-map functions, which is not possible with a standard functools.partial() function:

```
>>> def call_func_on_cuda(f, *args, **kwargs):
...     f, args, kwargs = tree_map(lambda t: t.cuda(), (f, args, kwargs))
...     return f(*args, **kwargs)
...
>>>
>>> tree_map(lambda t: t.cuda(), add_one)
optree.functools.partial(<built-in function add>, tensor(1., device='cuda:0'))
>>> call_func_on_cuda(add_one, torch.tensor([[1, 2], [3, 4]]))
tensor([[2., 3.],
        [4., 5.]], device='cuda:0')
```

Passing zero arguments to *partial* effectively wraps the original function, making it a valid argument in tree-map functions:

```
>>>
>>> call_func_on_cuda(partial(torch.add), torch.tensor(1), torch.tensor(2))
tensor(3, device='cuda:0')
```

Had we passed `operator.add()` to `call_func_on_cuda` directly, it would have resulted in a `TypeError` or `AttributeError`.

Create a new *partial* instance.

optree.functools.**reduce**(*func*, *tree*, *initial=<MISSING>*, *, *is_leaf=None*, *none_is_leaf=False*, *namespace=''*)

Traversal through a pytree and reduce the leaves in left-to-right depth-first order.

See also `tree_leaves()` and `tree_sum()`.

```
>>> tree_reduce(lambda x, y: x + y, {'x': 1, 'y': (2, 3)})
6
>>> tree_reduce(lambda x, y: x + y, {'x': 1, 'y': (2, None), 'z': 3})  # `None` is a␣
↪non-leaf node with arity 0 by default
6
>>> tree_reduce(lambda x, y: x and y, {'x': 1, 'y': (2, None), 'z': 3})
3
>>> tree_reduce(lambda x, y: x and y, {'x': 1, 'y': (2, None), 'z': 3}, none_is_␣
↪leaf=True)
None
```

Parameters

- **func** (*callable*) – A function that takes two arguments and returns a value of the same type.
- **tree** (*pytree*) – A pytree to be traversed.
- **initial** (*object, optional*) – An initial value to be used for the reduction. If not provided, the first leaf value is used as the initial value.
- **is_leaf** (*callable, optional*) – An optionally specified function that will be called at each flattening step. It should return a boolean, with `True` stopping the traversal and the whole subtree being treated as a leaf, and `False` indicating the flattening should traverse the current object.
- **none_is_leaf** (*bool, optional*) – Whether to treat `None` as a leaf. If `False`, `None` is a non-leaf node with arity 0. Thus `None` is contained in the treespec rather than in the leaves list and `None` will be remain in the result pytree. (default: `False`)
- **namespace** (*str, optional*) – The registry namespace used for custom pytree node types. (default: `''`, i.e., the global namespace)

Return type

`TypeVar`(T)

Returns

The result of reducing the leaves of the pytree using `func`.

CHAPTER

FOUR

TYPING SUPPORT

PyTreeSpec	Representing the structure of the pytree.
PyTreeDef	alias of *PyTreeSpec*
PyTreeKind(self, value)	The kind of a pytree node.
PyTree()	Generic PyTree type.
PyTreeTypeVar(name, param)	Type variable for PyTree.
CustomTreeNode(*args, **kwargs)	The abstract base class for custom pytree nodes.
is_namedtuple(obj)	Return whether the object is an instance of namedtuple or a subclass of namedtuple.
is_namedtuple_instance(obj)	Return whether the object is an instance of namedtuple.
is_namedtuple_class(cls)	Return whether the class is a subclass of namedtuple.
namedtuple_fields(obj)	Return the field names of a namedtuple.
is_structseq(obj)	Return whether the object is an instance of PyStructSequence or a class of PyStructSequence.
is_structseq_instance(obj)	Return whether the object is an instance of PyStructSequence.
is_structseq_class(cls)	Return whether the object is a class of PyStructSequence.
structseq_fields(obj)	Return the field names of a PyStructSequence.

class optree.**PyTreeSpec**

Bases: pybind11_object

Representing the structure of the pytree.

__annotations__ = {}

__delattr__(*name*, /)

Implement delattr(self, name).

__eq__(*self:* optree.PyTreeSpec, *other:* optree.PyTreeSpec) → bool

Test for equality to another object.

__ge__(*self:* optree.PyTreeSpec, *other:* optree.PyTreeSpec) → bool

Test for this treespec is a suffix of another object.

__getattribute__(*name*, /)

Return getattr(self, name).

__getstate__(*self:* optree.PyTreeSpec) → object

__gt__(*self:* optree.PyTreeSpec, *other:* optree.PyTreeSpec) → bool

Test for this treespec is a strict suffix of another object.

__hash__(*self:* optree.PyTreeSpec) → int

Return the hash of the treespec.

__init__(**args*, ***kwargs*)

__le__(*self:* optree.PyTreeSpec, *other:* optree.PyTreeSpec) → bool

Test for this treespec is a prefix of another object.

__len__(*self:* optree.PyTreeSpec) → int

Number of leaves in the tree.

__lt__(*self:* optree.PyTreeSpec, *other:* optree.PyTreeSpec) → bool

Test for this treespec is a strict prefix of another object.

__ne__(*self:* optree.PyTreeSpec, *other:* optree.PyTreeSpec) → bool

Test for inequality to another object.

__new__(***kwargs*)

__setattr__(*name*, *value*, */*)

Implement setattr(self, name, value).

__setstate__(*self:* optree.PyTreeSpec, *state: object*) → None

Serialization support for PyTreeSpec.

accessors(*self:* optree.PyTreeSpec) → list[object]

Return a list of accessors to the leaves in the treespec.

broadcast_to_common_suffix(*self:* optree.PyTreeSpec, *other:* optree.PyTreeSpec) → *optree.PyTreeSpec*

Broadcast to the common suffix of this treespec and other treespec.

child(*self:* optree.PyTreeSpec, *index: int*) → *optree.PyTreeSpec*

Return the treespec for the child at the given index.

children(*self:* optree.PyTreeSpec) → list[*optree.PyTreeSpec*]

Return a list of treespecs for the children.

compose(*self:* optree.PyTreeSpec, *inner_treespec:* optree.PyTreeSpec) → *optree.PyTreeSpec*

Compose two treespecs. Constructs the inner treespec as a subtree at each leaf node.

entries(*self:* optree.PyTreeSpec) → list

Return a list of one-level entries to the children.

entry(*self:* optree.PyTreeSpec, *index: int*) → object

Return the entry at the given index.

flatten_up_to(*self:* optree.PyTreeSpec, *full_tree: object*) → list

Flatten the subtrees in `full_tree` up to the structure of this treespec and return a list of subtrees.

is_leaf(*self:* optree.PyTreeSpec, *strict: bool = True*) → bool

Test whether the current node is a leaf.

is_prefix(*self:* optree.PyTreeSpec, *other:* optree.PyTreeSpec, *strict: bool = False*) → bool

Test whether this treespec is a prefix of the given treespec.

is_suffix(*self:* optree.PyTreeSpec, *other:* optree.PyTreeSpec, *strict: bool = False*) → bool

Test whether this treespec is a suffix of the given treespec.

property kind

The kind of the current node.

property namespace

The registry namespace used to resolve the custom pytree node types.

property none_is_leaf

Whether to treat None as a leaf. If false, None is a non-leaf node with arity 0. Thus None is contained in the treespec rather than in the leaves list.

property num_children

Number of children in the current node. Note that a leaf is also a node but has no children.

property num_leaves

Number of leaves in the tree.

property num_nodes

Number of nodes in the tree. Note that a leaf is also a node but has no children.

paths(*self:* optree.PyTreeSpec) → list[tuple]

Return a list of paths to the leaves of the treespec.

property type

The type of the current node. Return None if the current node is a leaf.

unflatten(*self:* optree.PyTreeSpec, *leaves: Iterable*) → object

Reconstruct a pytree from the leaves.

walk(*self:* optree.PyTreeSpec, *f_node: Callable*, *f_leaf: object*, *leaves: Iterable*) → object

Walk over the pytree structure, calling `f_node(children, node_data)` at nodes, and `f_leaf(leaf)` at leaves.

optree.**PyTreeDef**

alias of *PyTreeSpec*

class optree.**PyTreeKind**(*self: optree._C.PyTreeKind*, *value: int*)

Bases: `pybind11_object`

The kind of a pytree node.

Members:

CUSTOM : A custom type.

LEAF : An opaque leaf node.

NONE : None.

TUPLE : A tuple.

LIST : A list.

DICT : A dict.

NAMEDTUPLE : A collections.namedtuple.

ORDEREDDICT : A collections.OrderedDict.

DEFAULTDICT : A collections.defaultdict.

DEQUE : A collections.deque.

STRUCTSEQUENCE : A PyStructSequence.

CUSTOM = <PyTreeKind.CUSTOM: 0>

DEFAULTDICT = <PyTreeKind.DEFAULTDICT: 8>

DEQUE = <PyTreeKind.DEQUE: 9>

DICT = <PyTreeKind.DICT: 5>

LEAF = <PyTreeKind.LEAF: 1>

LIST = <PyTreeKind.LIST: 4>

NAMEDTUPLE = <PyTreeKind.NAMEDTUPLE: 6>

NONE = <PyTreeKind.NONE: 2>

ORDEREDDICT = <PyTreeKind.ORDEREDDICT: 7>

STRUCTSEQUENCE = <PyTreeKind.STRUCTSEQUENCE: 10>

TUPLE = <PyTreeKind.TUPLE: 3>

__annotations__ = {}

__delattr__(*name*, */*)

Implement delattr(self, name).

__eq__(*self: object*, *other: object*) → bool

__ge__(*value*, */*)

Return self>=value.

__getattribute__(*name*, */*)

Return getattr(self, name).

__getstate__(*self: object*) → int

__gt__(*value*, */*)

Return self>value.

__hash__(*self: object*) → int

__index__(*self: optree._C.PyTreeKind*) → int

__init__(*self: optree._C.PyTreeKind*, *value: int*) → None

__int__(*self: optree._C.PyTreeKind*) → int

__le__(*value*, */*)

Return self<=value.

__lt__(*value*, */*)

Return self<value.

__members__ = {'CUSTOM': <PyTreeKind.CUSTOM: 0>, 'DEFAULTDICT': <PyTreeKind.DEFAULTDICT: 8>, 'DEQUE': <PyTreeKind.DEQUE: 9>, 'DICT': <PyTreeKind.DICT: 5>, 'LEAF': <PyTreeKind.LEAF: 1>, 'LIST': <PyTreeKind.LIST: 4>, 'NAMEDTUPLE': <PyTreeKind.NAMEDTUPLE: 6>, 'NONE': <PyTreeKind.NONE: 2>, 'ORDEREDDICT': <PyTreeKind.ORDEREDDICT: 7>, 'STRUCTSEQUENCE': <PyTreeKind.STRUCTSEQUENCE: 10>, 'TUPLE': <PyTreeKind.TUPLE: 3>}

__ne__(*self: object*, *other: object*) → bool

__new__(***kwargs*)

__setattr__(*name*, *value*, */*)

Implement setattr(self, name, value).

__setstate__(*self: optree._C.PyTreeKind*, *state: int*) → None

property name

property value

class optree.**PyTree**

Bases: Generic[T]

Generic PyTree type.

```
>>> import torch
>>> from optree.typing import PyTree
>>> TensorTree = PyTree[torch.Tensor]
>>> TensorTree
typing.Union[torch.Tensor,
             typing.Tuple[ForwardRef('PyTree[torch.Tensor]'), ...],
             typing.List[ForwardRef('PyTree[torch.Tensor]')],
             typing.Dict[typing.Any, ForwardRef('PyTree[torch.Tensor]')],
             typing.Deque[ForwardRef('PyTree[torch.Tensor]')],
             optree.typing.CustomTreeNode[ForwardRef('PyTree[torch.Tensor]')]]
```

Prohibit instantiation.

classmethod __class_getitem__(*cls*, *item*)

Instantiate a PyTree type with the given type.

Return type
TypeAlias

static __new__(*cls*)

Prohibit instantiation.

Return type
NoReturn

classmethod __init_subclass__(**args*, ***kwargs*)

Prohibit subclassing.

Return type
NoReturn

__copy__()

Immutable copy.

Return type
PyTree

__deepcopy__(*memo*)

Immutable copy.

Return type
PyTree

__annotations__ = {}

__orig_bases__ = (typing.Generic[~T],)

__parameters__ = (~T,)

optree.**PyTreeTypeVar**(*name: str*, *param: type*) → TypeAlias

Type variable for PyTree.

```
>>> import torch
>>> from optree.typing import PyTreeTypeVar
>>> TensorTree = PyTreeTypeVar('TensorTree', torch.Tensor)
>>> TensorTree
typing.Union[torch.Tensor,
             typing.Tuple[ForwardRef('TensorTree'), ...],
             typing.List[ForwardRef('TensorTree')],
             typing.Dict[typing.Any, ForwardRef('TensorTree')],
             typing.Deque[ForwardRef('TensorTree')],
             optree.typing.CustomTreeNode[ForwardRef('TensorTree')]]
```

class optree.**CustomTreeNode**(**args*, ***kwargs*)

Bases: Protocol[T]

The abstract base class for custom pytree nodes.

tree_flatten()

Flatten the custom pytree node into children and auxiliary data.

Return type
tuple[Iterable[TypeVar(T)], Optional[TypeVar(_MetaData, bound= Hashable)]] | tuple[Iterable[TypeVar(T)], Optional[TypeVar(_MetaData, bound= Hashable)], Optional[Iterable[Any]]]

classmethod tree_unflatten(*metadata*, *children*)

Unflatten the children and auxiliary data into the custom pytree node.

Return type
CustomTreeNode[TypeVar(T)]

__abstractmethods__ = frozenset({})

__annotations__ = {}

__init__(**args*, ***kwargs*)

__non_callable_proto_members__ = {}

__orig_bases__ = (typing_extensions.Protocol[~T],)

__parameters__ = (~T,)

__protocol_attrs__ = {'tree_flatten', 'tree_unflatten'}

classmethod __subclasshook__(*other*)

Abstract classes can override this to customize issubclass().

This is invoked early on by abc.ABCMeta.__subclasscheck__(). It should return True, False or NotImplemented. If it returns NotImplemented, the normal algorithm is used. Otherwise, it overrides the normal algorithm (and the outcome is cached).

optree.**is_namedtuple**(*obj: object*) → bool

Return whether the object is an instance of namedtuple or a subclass of namedtuple.

optree.**is_namedtuple_instance**(*obj: object*) → bool

Return whether the object is an instance of namedtuple.

optree.**is_namedtuple_class**(*cls: object*) → bool

Return whether the class is a subclass of namedtuple.

optree.**namedtuple_fields**(*obj: object*) → tuple

Return the field names of a namedtuple.

optree.**is_structseq**(*obj: object*) → bool

Return whether the object is an instance of PyStructSequence or a class of PyStructSequence.

optree.**is_structseq_instance**(*obj: object*) → bool

Return whether the object is an instance of PyStructSequence.

optree.**is_structseq_class**(*cls: object*) → bool

Return whether the object is a class of PyStructSequence.

optree.**structseq_fields**(*obj: object*) → tuple

Return the field names of a PyStructSequence.

CHAPTER

FIVE

API REFERENCES

OpTree: Optimized PyTree Utilities.

optree.**MAX_RECURSION_DEPTH: int = 1000**

Maximum recursion depth for pytree traversal. It is 1000.

This limit prevents infinite recursion from causing an overflow of the C stack and crashing Python.

optree.**NONE_IS_NODE: bool = False**

Literal constant that treats `None` as a pytree non-leaf node.

optree.**NONE_IS_LEAF: bool = True**

Literal constant that treats `None` as a pytree leaf node.

optree.**tree_flatten**(*tree*, *is_leaf=None*, *, *none_is_leaf=False*, *namespace=''*)

Flatten a pytree.

See also *tree_flatten_with_path()* and *tree_unflatten()*.

The flattening order (i.e., the order of elements in the output list) is deterministic, corresponding to a left-to-right depth-first tree traversal.

```
>>> tree = {'b': (2, [3, 4]), 'a': 1, 'c': None, 'd': 5}
>>> tree_flatten(tree)
(
    [1, 2, 3, 4, 5],
    PyTreeSpec({'a': *, 'b': (*, [*, *]), 'c': None, 'd': *})
)
>>> tree_flatten(tree, none_is_leaf=True)
(
    [1, 2, 3, 4, None, 5],
    PyTreeSpec({'a': *, 'b': (*, [*, *]), 'c': *, 'd': *}, NoneIsLeaf)
)
>>> tree_flatten(1)
([1], PyTreeSpec(*))
>>> tree_flatten(None)
([], PyTreeSpec(None))
>>> tree_flatten(None, none_is_leaf=True)
([None], PyTreeSpec(*, NoneIsLeaf))
```

For unordered dictionaries, `dict` and `collections.defaultdict`, the order is dependent on the **sorted** keys in the dictionary. Please use `collections.OrderedDict` if you want to keep the keys in the insertion order.

```
>>> from collections import OrderedDict
>>> tree = OrderedDict([('b', (2, [3, 4])), ('a', 1), ('c', None), ('d', 5)])
>>> tree_flatten(tree)
(
    [2, 3, 4, 1, 5],
    PyTreeSpec(OrderedDict({'b': (*, [*, *]), 'a': *, 'c': None, 'd': *}))
)
>>> tree_flatten(tree, none_is_leaf=True)
(
    [2, 3, 4, 1, None, 5],
    PyTreeSpec(OrderedDict({'b': (*, [*, *]), 'a': *, 'c': *, 'd': *}), NoneIsLeaf)
)
```

Parameters

- **tree** (*pytree*) – A pytree to flatten.
- **is_leaf** (*callable, optional*) – An optionally specified function that will be called at each flattening step. It should return a boolean, with `True` stopping the traversal and the whole subtree being treated as a leaf, and `False` indicating the flattening should traverse the current object.
- **none_is_leaf** (*bool, optional*) – Whether to treat `None` as a leaf. If `False`, `None` is a non-leaf node with arity 0. Thus `None` is contained in the treespec rather than in the leaves list. (default: `False`)
- **namespace** (*str, optional*) – The registry namespace used for custom pytree node types. (default: `''`, i.e., the global namespace)

Return type

`tuple`[`list`[`TypeVar`(T)], *PyTreeSpec*]

Returns

A pair `(leaves, treespec)` where the first element is a list of leaf values and the second element is a treespec representing the structure of the pytree.

optree.**tree_flatten_with_path**(*tree*, *is_leaf=None*, *, *none_is_leaf=False*, *namespace=''*)

Flatten a pytree and additionally record the paths.

See also *tree_flatten()*, *tree_paths()*, and *treespec_paths()*.

The flattening order (i.e., the order of elements in the output list) is deterministic, corresponding to a left-to-right depth-first tree traversal.

```
>>> tree = {'b': (2, [3, 4]), 'a': 1, 'c': None, 'd': 5}
>>> tree_flatten_with_path(tree)
(
    [('a',), ('b', 0), ('b', 1, 0), ('b', 1, 1), ('d',)],
    [1, 2, 3, 4, 5],
    PyTreeSpec({'a': *, 'b': (*, [*, *]), 'c': None, 'd': *})
)
>>> tree_flatten_with_path(tree, none_is_leaf=True)
(
    [('a',), ('b', 0), ('b', 1, 0), ('b', 1, 1), ('c',), ('d',)],
    [1, 2, 3, 4, None, 5],
    PyTreeSpec({'a': *, 'b': (*, [*, *]), 'c': *, 'd': *}, NoneIsLeaf)
)
```

(continues on next page)

(continued from previous page)

```
>>> tree_flatten_with_path(1)
([()], [1], PyTreeSpec(*))
>>> tree_flatten_with_path(None)
([], [], PyTreeSpec(None))
>>> tree_flatten_with_path(None, none_is_leaf=True)
([()], [None], PyTreeSpec(*, NoneIsLeaf))
```

For unordered dictionaries, `dict` and `collections.defaultdict`, the order is dependent on the **sorted** keys in the dictionary. Please use `collections.OrderedDict` if you want to keep the keys in the insertion order.

```
>>> from collections import OrderedDict
>>> tree = OrderedDict([('b', (2, [3, 4])), ('a', 1), ('c', None), ('d', 5)])
>>> tree_flatten_with_path(tree)
(
    [('b', 0), ('b', 1, 0), ('b', 1, 1), ('a',), ('d',)],
    [2, 3, 4, 1, 5],
    PyTreeSpec(OrderedDict({'b': (*, [*, *]), 'a': *, 'c': None, 'd': *}))
)
>>> tree_flatten_with_path(tree, none_is_leaf=True)
(
    [('b', 0), ('b', 1, 0), ('b', 1, 1), ('a',), ('c',), ('d',)],
    [2, 3, 4, 1, None, 5],
    PyTreeSpec(OrderedDict({'b': (*, [*, *]), 'a': *, 'c': *, 'd': *}), NoneIsLeaf)
)
```

Parameters

- **tree** (*pytree*) – A pytree to flatten.
- **is_leaf** (*callable, optional*) – An optionally specified function that will be called at each flattening step. It should return a boolean, with `True` stopping the traversal and the whole subtree being treated as a leaf, and `False` indicating the flattening should traverse the current object.
- **none_is_leaf** (*bool, optional*) – Whether to treat `None` as a leaf. If `False`, `None` is a non-leaf node with arity 0. Thus `None` is contained in the treespec rather than in the leaves list. (default: `False`)
- **namespace** (*str, optional*) – The registry namespace used for custom pytree node types. (default: `''`, i.e., the global namespace)

Return type

`tuple`[`list`[`tuple`[`Any`, `...`]], `list`[`TypeVar`(`T`)], *PyTreeSpec*]

Returns

A triple `(paths, leaves, treespec)`. The first element is a list of the paths to the leaf values, while each path is a tuple of the index or keys. The second element is a list of leaf values and the last element is a treespec representing the structure of the pytree.

optree.**tree_flatten_with_accessor**(*tree*, *is_leaf=None*, *, *none_is_leaf=False*, *namespace=''*)

Flatten a pytree and additionally record the accessors.

See also *tree_flatten()*, *tree_accessors()*, and *treespec_accessors()*.

The flattening order (i.e., the order of elements in the output list) is deterministic, corresponding to a left-to-right depth-first tree traversal.

```
>>> tree = {'b': (2, [3, 4]), 'a': 1, 'c': None, 'd': 5}
>>> tree_flatten_with_accessor(tree)
(
    [
        PyTreeAccessor(*['a'], (MappingEntry(key='a', type=<class 'dict'>),)),
        PyTreeAccessor(*['b'][0], (MappingEntry(key='b', type=<class 'dict'>),␣
↪SequenceEntry(index=0, type=<class 'tuple'>))),
        PyTreeAccessor(*['b'][1][0], (MappingEntry(key='b', type=<class 'dict'>),␣
↪SequenceEntry(index=1, type=<class 'tuple'>), SequenceEntry(index=0, type=<class
↪'list'>))),
        PyTreeAccessor(*['b'][1][1], (MappingEntry(key='b', type=<class 'dict'>),␣
↪SequenceEntry(index=1, type=<class 'tuple'>), SequenceEntry(index=1, type=<class
↪'list'>))),
        PyTreeAccessor(*['d'], (MappingEntry(key='d', type=<class 'dict'>),))
    ],
    [1, 2, 3, 4, 5],
    PyTreeSpec({'a': *, 'b': (*, [*, *]), 'c': None, 'd': *})
)
>>> tree_flatten_with_accessor(tree, none_is_leaf=True)
(
    [
        PyTreeAccessor(*['a'], (MappingEntry(key='a', type=<class 'dict'>),)),
        PyTreeAccessor(*['b'][0], (MappingEntry(key='b', type=<class 'dict'>),␣
↪SequenceEntry(index=0, type=<class 'tuple'>))),
        PyTreeAccessor(*['b'][1][0], (MappingEntry(key='b', type=<class 'dict'>),␣
↪SequenceEntry(index=1, type=<class 'tuple'>), SequenceEntry(index=0, type=<class
↪'list'>))),
        PyTreeAccessor(*['b'][1][1], (MappingEntry(key='b', type=<class 'dict'>),␣
↪SequenceEntry(index=1, type=<class 'tuple'>), SequenceEntry(index=1, type=<class
↪'list'>))),
        PyTreeAccessor(*['c'], (MappingEntry(key='c', type=<class 'dict'>),)),
        PyTreeAccessor(*['d'], (MappingEntry(key='d', type=<class 'dict'>),))
    ],
    [1, 2, 3, 4, None, 5],
    PyTreeSpec({'a': *, 'b': (*, [*, *]), 'c': *, 'd': *}, NoneIsLeaf)
)
>>> tree_flatten_with_accessor(1)
([PyTreeAccessor(*, ())], [1], PyTreeSpec(*))
>>> tree_flatten_with_accessor(None)
([], [], PyTreeSpec(None))
>>> tree_flatten_with_accessor(None, none_is_leaf=True)
([PyTreeAccessor(*, ())], [None], PyTreeSpec(*, NoneIsLeaf))
```

For unordered dictionaries, `dict` and `collections.defaultdict`, the order is dependent on the **sorted** keys in the dictionary. Please use `collections.OrderedDict` if you want to keep the keys in the insertion order.

```
>>> from collections import OrderedDict
>>> tree = OrderedDict([('b', (2, [3, 4])), ('a', 1), ('c', None), ('d', 5)])
>>> tree_flatten_with_accessor(tree)
(
    [
        PyTreeAccessor(*['b'][0], (MappingEntry(key='b', type=<class 'collections.
↪OrderedDict'>), SequenceEntry(index=0, type=<class 'tuple'>))),
```

(continues on next page)

(continued from previous page)

```
        PyTreeAccessor(*['b'][1][0], (MappingEntry(key='b', type=<class
↪'collections.OrderedDict'>), SequenceEntry(index=1, type=<class 'tuple'>),␣
↪SequenceEntry(index=0, type=<class 'list'>))),
        PyTreeAccessor(*['b'][1][1], (MappingEntry(key='b', type=<class
↪'collections.OrderedDict'>), SequenceEntry(index=1, type=<class 'tuple'>),␣
↪SequenceEntry(index=1, type=<class 'list'>))),
        PyTreeAccessor(*['a'], (MappingEntry(key='a', type=<class 'collections.
↪OrderedDict'>),)),
        PyTreeAccessor(*['d'], (MappingEntry(key='d', type=<class 'collections.
↪OrderedDict'>),))
    ],
    [2, 3, 4, 1, 5],
    PyTreeSpec(OrderedDict({'b': (*, [*, *]), 'a': *, 'c': None, 'd': *}))
)
>>> tree_flatten_with_accessor(tree, none_is_leaf=True)
(
    [
        PyTreeAccessor(*['b'][0], (MappingEntry(key='b', type=<class 'collections.
↪OrderedDict'>), SequenceEntry(index=0, type=<class 'tuple'>))),
        PyTreeAccessor(*['b'][1][0], (MappingEntry(key='b', type=<class
↪'collections.OrderedDict'>), SequenceEntry(index=1, type=<class 'tuple'>),␣
↪SequenceEntry(index=0, type=<class 'list'>))),
        PyTreeAccessor(*['b'][1][1], (MappingEntry(key='b', type=<class
↪'collections.OrderedDict'>), SequenceEntry(index=1, type=<class 'tuple'>),␣
↪SequenceEntry(index=1, type=<class 'list'>))),
        PyTreeAccessor(*['a'], (MappingEntry(key='a', type=<class 'collections.
↪OrderedDict'>),)),
        PyTreeAccessor(*['c'], (MappingEntry(key='c', type=<class 'collections.
↪OrderedDict'>),)),
        PyTreeAccessor(*['d'], (MappingEntry(key='d', type=<class 'collections.
↪OrderedDict'>),))
    ],
    [2, 3, 4, 1, None, 5],
    PyTreeSpec(OrderedDict({'b': (*, [*, *]), 'a': *, 'c': *, 'd': *}), NoneIsLeaf)
)
```

Parameters

- **tree** (*pytree*) – A pytree to flatten.
- **is_leaf** (*callable, optional*) – An optionally specified function that will be called at each flattening step. It should return a boolean, with `True` stopping the traversal and the whole subtree being treated as a leaf, and `False` indicating the flattening should traverse the current object.
- **none_is_leaf** (*bool, optional*) – Whether to treat `None` as a leaf. If `False`, `None` is a non-leaf node with arity 0. Thus `None` is contained in the treespec rather than in the leaves list. (default: `False`)
- **namespace** (*str, optional*) – The registry namespace used for custom pytree node types. (default: `''`, i.e., the global namespace)

Return type

`tuple`[`list`[PyTreeAccessor], `list`[`TypeVar`(T)], *PyTreeSpec*]

Returns

A triple `(accessors, leaves, treespec)`. The first element is a list of accessors to the leaf values. The second element is a list of leaf values and the last element is a treespec representing the structure of the pytree.

optree.**tree_unflatten**(*treespec*, *leaves*)

Reconstruct a pytree from the treespec and the leaves.

The inverse of *tree_flatten()*.

```
>>> tree = {'b': (2, [3, 4]), 'a': 1, 'c': None, 'd': 5}
>>> leaves, treespec = tree_flatten(tree)
>>> tree == tree_unflatten(treespec, leaves)
True
```

Parameters

- **treespec** (PyTreeSpec) – The treespec to reconstruct.
- **leaves** (*iterable*) – The list of leaves to use for reconstruction. The list must match the number of leaves of the treespec.

Return type

Union[TypeVar(T), Tuple[PyTree[T], ...], List[PyTree[T]], Dict[Any, PyTree[T]], Deque[PyTree[T]], *CustomTreeNode*[PyTree[T]]]

Returns

The reconstructed pytree, containing the `leaves` placed in the structure described by `treespec`.

optree.**tree_iter**(*tree*, *is_leaf=None*, ***, *none_is_leaf=False*, *namespace=''*)

Get an iterator over the leaves of a pytree.

See also *tree_flatten()* and *tree_leaves()*.

```
>>> tree = {'b': (2, [3, 4]), 'a': 1, 'c': None, 'd': 5}
>>> list(tree_iter(tree))
[1, 2, 3, 4, 5]
>>> list(tree_iter(tree, none_is_leaf=True))
[1, 2, 3, 4, None, 5]
>>> list(tree_iter(1))
[1]
>>> list(tree_iter(None))
[]
>>> list(tree_iter(None, none_is_leaf=True))
[None]
```

Parameters

- **tree** (*pytree*) – A pytree to iterate over.
- **is_leaf** (*callable, optional*) – An optionally specified function that will be called at each flattening step. It should return a boolean, with True stopping the traversal and the whole subtree being treated as a leaf, and False indicating the flattening should traverse the current object.
- **none_is_leaf** (*bool, optional*) – Whether to treat None as a leaf. If False, None is a non-leaf node with arity 0. Thus None is contained in the treespec rather than in the leaves list. (default: False)

- **namespace** (*str, optional*) – The registry namespace used for custom pytree node types. (default: `''`, i.e., the global namespace)

Return type
`Iterable`[`TypeVar`(`T`)]

Returns
An iterator over the leaf values.

`optree.tree_leaves`(*tree*, *is_leaf=None*, *, *none_is_leaf=False*, *namespace=''*)

Get the leaves of a pytree.

See also *tree_flatten()* and *tree_iter()*.

```
>>> tree = {'b': (2, [3, 4]), 'a': 1, 'c': None, 'd': 5}
>>> tree_leaves(tree)
[1, 2, 3, 4, 5]
>>> tree_leaves(tree, none_is_leaf=True)
[1, 2, 3, 4, None, 5]
>>> tree_leaves(1)
[1]
>>> tree_leaves(None)
[]
>>> tree_leaves(None, none_is_leaf=True)
[None]
```

Parameters

- **tree** (*pytree*) – A pytree to flatten.
- **is_leaf** (*callable, optional*) – An optionally specified function that will be called at each flattening step. It should return a boolean, with `True` stopping the traversal and the whole subtree being treated as a leaf, and `False` indicating the flattening should traverse the current object.
- **none_is_leaf** (*bool, optional*) – Whether to treat `None` as a leaf. If `False`, `None` is a non-leaf node with arity 0. Thus `None` is contained in the treespec rather than in the leaves list. (default: `False`)
- **namespace** (*str, optional*) – The registry namespace used for custom pytree node types. (default: `''`, i.e., the global namespace)

Return type
`list`[`TypeVar`(`T`)]

Returns
A list of leaf values.

`optree.tree_structure`(*tree*, *is_leaf=None*, *, *none_is_leaf=False*, *namespace=''*)

Get the treespec for a pytree.

See also *tree_flatten()*.

```
>>> tree = {'b': (2, [3, 4]), 'a': 1, 'c': None, 'd': 5}
>>> tree_structure(tree)
PyTreeSpec({'a': *, 'b': (*, [*, *]), 'c': None, 'd': *})
>>> tree_structure(tree, none_is_leaf=True)
PyTreeSpec({'a': *, 'b': (*, [*, *]), 'c': *, 'd': *}, NoneIsLeaf)
>>> tree_structure(1)
```

(continues on next page)

(continued from previous page)

```
PyTreeSpec(*)
>>> tree_structure(None)
PyTreeSpec(None)
>>> tree_structure(None, none_is_leaf=True)
PyTreeSpec(*, NoneIsLeaf)
```

Parameters

- **tree** (*pytree*) – A pytree to flatten.
- **is_leaf** (*callable, optional*) – An optionally specified function that will be called at each flattening step. It should return a boolean, with `True` stopping the traversal and the whole subtree being treated as a leaf, and `False` indicating the flattening should traverse the current object.
- **none_is_leaf** (*bool, optional*) – Whether to treat `None` as a leaf. If `False`, `None` is a non-leaf node with arity 0. Thus `None` is contained in the treespec rather than in the leaves list. (default: `False`)
- **namespace** (*str, optional*) – The registry namespace used for custom pytree node types. (default: `''`, i.e., the global namespace)

Return type
PyTreeSpec

Returns
A treespec object representing the structure of the pytree.

optree.**tree_paths**(*tree*, *is_leaf=None*, *, *none_is_leaf=False*, *namespace=''*)

Get the path entries to the leaves of a pytree.

See also *tree_flatten()*, *tree_flatten_with_path()*, and *treespec_paths()*.

```
>>> tree = {'b': (2, [3, 4]), 'a': 1, 'c': None, 'd': 5}
>>> tree_paths(tree)
[('a',), ('b', 0), ('b', 1, 0), ('b', 1, 1), ('d',)]
>>> tree_paths(tree, none_is_leaf=True)
[('a',), ('b', 0), ('b', 1, 0), ('b', 1, 1), ('c',), ('d',)]
>>> tree_paths(1)
[()]
>>> tree_paths(None)
[]
>>> tree_paths(None, none_is_leaf=True)
[()]
```

Parameters

- **tree** (*pytree*) – A pytree to flatten.
- **is_leaf** (*callable, optional*) – An optionally specified function that will be called at each flattening step. It should return a boolean, with `True` stopping the traversal and the whole subtree being treated as a leaf, and `False` indicating the flattening should traverse the current object.
- **none_is_leaf** (*bool, optional*) – Whether to treat `None` as a leaf. If `False`, `None` is a non-leaf node with arity 0. Thus `None` is contained in the treespec rather than in the leaves list. (default: `False`)

- **namespace** (*str, optional*) – The registry namespace used for custom pytree node types. (default: `''`, i.e., the global namespace)

Return type
`list[tuple[Any, ...]]`

Returns
A list of the paths to the leaf values, while each path is a tuple of the index or keys.

optree.**tree_accessors**(*tree*, *is_leaf=None*, *, *none_is_leaf=False*, *namespace=''*)

Get the accessors to the leaves of a pytree.

See also *tree_flatten()*, *tree_flatten_with_accessor()*, and *treespec_accessors()*.

```
>>> tree = {'b': (2, [3, 4]), 'a': 1, 'c': None, 'd': 5}
>>> tree_accessors(tree)
[
    PyTreeAccessor(*['a'], (MappingEntry(key='a', type=<class 'dict'>),)),
    PyTreeAccessor(*['b'][0], (MappingEntry(key='b', type=<class 'dict'>),␣
↪SequenceEntry(index=0, type=<class 'tuple'>))),
    PyTreeAccessor(*['b'][1][0], (MappingEntry(key='b', type=<class 'dict'>),␣
↪SequenceEntry(index=1, type=<class 'tuple'>), SequenceEntry(index=0, type=<class
↪'list'>))),
    PyTreeAccessor(*['b'][1][1], (MappingEntry(key='b', type=<class 'dict'>),␣
↪SequenceEntry(index=1, type=<class 'tuple'>), SequenceEntry(index=1, type=<class
↪'list'>))),
    PyTreeAccessor(*['d'], (MappingEntry(key='d', type=<class 'dict'>),))
]
>>> tree_accessors(tree, none_is_leaf=True)
[
    PyTreeAccessor(*['a'], (MappingEntry(key='a', type=<class 'dict'>),)),
    PyTreeAccessor(*['b'][0], (MappingEntry(key='b', type=<class 'dict'>),␣
↪SequenceEntry(index=0, type=<class 'tuple'>))),
    PyTreeAccessor(*['b'][1][0], (MappingEntry(key='b', type=<class 'dict'>),␣
↪SequenceEntry(index=1, type=<class 'tuple'>), SequenceEntry(index=0, type=<class
↪'list'>))),
    PyTreeAccessor(*['b'][1][1], (MappingEntry(key='b', type=<class 'dict'>),␣
↪SequenceEntry(index=1, type=<class 'tuple'>), SequenceEntry(index=1, type=<class
↪'list'>))),
    PyTreeAccessor(*['c'], (MappingEntry(key='c', type=<class 'dict'>),)),
    PyTreeAccessor(*['d'], (MappingEntry(key='d', type=<class 'dict'>),))
]
>>> tree_accessors(1)
[PyTreeAccessor(*, ())]
>>> tree_accessors(None)
[]
>>> tree_accessors(None, none_is_leaf=True)
[PyTreeAccessor(*, ())]
```

Parameters

- **tree** (*pytree*) – A pytree to flatten.
- **is_leaf** (*callable, optional*) – An optionally specified function that will be called at each flattening step. It should return a boolean, with `True` stopping the traversal and the whole subtree being treated as a leaf, and `False` indicating the flattening should traverse the

current object.

- **none_is_leaf** (*bool, optional*) – Whether to treat `None` as a leaf. If `False`, `None` is a non-leaf node with arity 0. Thus `None` is contained in the treespec rather than in the leaves list. (default: `False`)
- **namespace** (*str, optional*) – The registry namespace used for custom pytree node types. (default: `''`, i.e., the global namespace)

Return type
`list`[PyTreeAccessor]

Returns
A list of accessors to the leaf values.

optree.**tree_is_leaf**(*tree*, *is_leaf=None*, *, *none_is_leaf=False*, *namespace=''*)

Test whether the given object is a leaf node.

See also *tree_flatten()*, *tree_leaves()*, and *all_leaves()*.

```
>>> tree_is_leaf(1)
True
>>> tree_is_leaf(None)
False
>>> tree_is_leaf(None, none_is_leaf=True)
True
>>> tree_is_leaf({'a': 1, 'b': (2, 3)})
False
```

Parameters

- **tree** (*pytree*) – A pytree to check if it is a leaf node.
- **is_leaf** (*callable, optional*) – An optionally specified function that will be called at each flattening step. It should return a boolean, with `True` stopping the traversal and the whole subtree being treated as a leaf, and `False` indicating the flattening should traverse the current object.
- **none_is_leaf** (*bool, optional*) – Whether to treat `None` as a leaf. If `False`, `None` is a non-leaf node with arity 0. Thus `None` is contained in the treespec rather than a leaf. (default: `False`)
- **namespace** (*str, optional*) – The registry namespace used for custom pytree node types. (default: `''`, i.e., the global namespace)

Return type
`bool`

Returns
A boolean indicating if all elements in the input iterable are leaves.

optree.**all_leaves**(*iterable*, *is_leaf=None*, *, *none_is_leaf=False*, *namespace=''*)

Test whether all elements in the given iterable are all leaves.

See also *tree_flatten()*, *tree_leaves()*, and *tree_is_leaf()*.

```
>>> tree = {'a': [1, 2, 3]}
>>> all_leaves(tree_leaves(tree))
True
>>> all_leaves([tree])
```

(continues on next page)

(continued from previous page)

```
False
>>> all_leaves([1, 2, None, 3])
False
>>> all_leaves([1, 2, None, 3], none_is_leaf=True)
True
```

Note that this function iterates and checks the elements in the input iterable object, which uses the `iter()` function. For dictionaries, `iter(d)` for a dictionary `d` iterates the keys of the dictionary, not the values.

```
>>> list({'a': 1, 'b': (2, 3)})
['a', 'b']
>>> all_leaves({'a': 1, 'b': (2, 3)})
True
```

This function is useful in advanced cases. For example, if a library allows arbitrary map operations on a flat list of leaves it may want to check if the result is still a flat list of leaves.

Parameters

- **iterable** (*iterable*) – A iterable of leaves.
- **is_leaf** (*callable, optional*) – An optionally specified function that will be called at each flattening step. It should return a boolean, with `True` stopping the traversal and the whole subtree being treated as a leaf, and `False` indicating the flattening should traverse the current object.
- **none_is_leaf** (*bool, optional*) – Whether to treat `None` as a leaf. If `False`, `None` is a non-leaf node with arity 0. Thus `None` is contained in the treespec rather than a leaf. (default: `False`)
- **namespace** (*str, optional*) – The registry namespace used for custom pytree node types. (default: `''`, i.e., the global namespace)

Return type

`bool`

Returns

A boolean indicating if all elements in the input iterable are leaves.

optree.**tree_map**(*func*, *tree*, **rests*, *is_leaf=None*, *none_is_leaf=False*, *namespace=''*)

Map a multi-input function over pytree args to produce a new pytree.

See also *tree_map_()*, *tree_map_with_path()*, *tree_map_with_path_()*, and *tree_broadcast_map()*.

```
>>> tree_map(lambda x: x + 1, {'x': 7, 'y': (42, 64)})
{'x': 8, 'y': (43, 65)}
>>> tree_map(lambda x: x + 1, {'x': 7, 'y': (42, 64), 'z': None})
{'x': 8, 'y': (43, 65), 'z': None}
>>> tree_map(lambda x: x is None, {'x': 7, 'y': (42, 64), 'z': None})
{'x': False, 'y': (False, False), 'z': None}
>>> tree_map(lambda x: x is None, {'x': 7, 'y': (42, 64), 'z': None}, none_is_
↪leaf=True)
{'x': False, 'y': (False, False), 'z': True}
```

If multiple inputs are given, the structure of the tree is taken from the first input; subsequent inputs need only have `tree` as a prefix:

```
>>> tree_map(lambda x, y: [x] + y, [5, 6], [[7, 9], [1, 2]])
[[5, 7, 9], [6, 1, 2]]
```

Parameters

- **func** (*callable*) – A function that takes `1 + len(rests)` arguments, to be applied at the corresponding leaves of the pytrees.
- **tree** (*pytree*) – A pytree to be mapped over, with each leaf providing the first positional argument to function `func`.
- **rests** (*tuple of pytree*) – A tuple of pytrees, each of which has the same structure as `tree` or has `tree` as a prefix.
- **is_leaf** (*callable, optional*) – An optionally specified function that will be called at each flattening step. It should return a boolean, with `True` stopping the traversal and the whole subtree being treated as a leaf, and `False` indicating the flattening should traverse the current object.
- **none_is_leaf** (*bool, optional*) – Whether to treat `None` as a leaf. If `False`, `None` is a non-leaf node with arity 0. Thus `None` is contained in the treespec rather than in the leaves list and `None` will be remain in the result pytree. (default: `False`)
- **namespace** (*str, optional*) – The registry namespace used for custom pytree node types. (default: `''`, i.e., the global namespace)

Return type

`Union`[`TypeVar`(U), `Tuple`[PyTree[U], `...`], `List`[PyTree[U]], `Dict`[`Any`, PyTree[U]], `Deque`[PyTree[U]], *`CustomTreeNode`*[PyTree[U]]]

Returns

A new pytree with the same structure as `tree` but with the value at each leaf given by `func(x, *xs)` where `x` is the value at the corresponding leaf in `tree` and `xs` is the tuple of values at corresponding nodes in `rests`.

optree.**tree_map_**(*func*, *tree*, **rests*, *is_leaf=None*, *none_is_leaf=False*, *namespace=''*)

Like *tree_map()*, but do an inplace call on each leaf and return the original tree.

See also *tree_map()*, *tree_map_with_path()*, and *tree_map_with_path_()*.

Parameters

- **func** (*callable*) – A function that takes `1 + len(rests)` arguments, to be applied at the corresponding leaves of the pytrees.
- **tree** (*pytree*) – A pytree to be mapped over, with each leaf providing the first positional argument to function `func`.
- **rests** (*tuple of pytree*) – A tuple of pytrees, each of which has the same structure as `tree` or has `tree` as a prefix.
- **is_leaf** (*callable, optional*) – An optionally specified function that will be called at each flattening step. It should return a boolean, with `True` stopping the traversal and the whole subtree being treated as a leaf, and `False` indicating the flattening should traverse the current object.
- **none_is_leaf** (*bool, optional*) – Whether to treat `None` as a leaf. If `False`, `None` is a non-leaf node with arity 0. Thus `None` is contained in the treespec rather than in the leaves list and `None` will be remain in the result pytree. (default: `False`)

- **namespace** (*str, optional*) – The registry namespace used for custom pytree node types. (default: `''`, i.e., the global namespace)

Return type
Union[TypeVar(T), Tuple[PyTree[T], ...], List[PyTree[T]], Dict[Any, PyTree[T]], Deque[PyTree[T]], *CustomTreeNode*[PyTree[T]]]

Returns
The original `tree` with the value at each leaf is given by the side-effect of function `func(x, *xs)` (not the return value) where `x` is the value at the corresponding leaf in `tree` and `xs` is the tuple of values at values at corresponding nodes in `rests`.

optree.**tree_map_with_path**(*func*, *tree*, **rests*, *is_leaf=None*, *none_is_leaf=False*, *namespace=''*)

Map a multi-input function over pytree args as well as the tree paths to produce a new pytree.

See also *tree_map()*, *tree_map_()*, and *tree_map_with_path_()*.

```
>>> tree_map_with_path(lambda p, x: (len(p), x), {'x': 7, 'y': (42, 64)})
{'x': (1, 7), 'y': ((2, 42), (2, 64))}
>>> tree_map_with_path(lambda p, x: x + len(p), {'x': 7, 'y': (42, 64), 'z': None})
{'x': 8, 'y': (44, 66), 'z': None}
>>> tree_map_with_path(lambda p, x: p, {'x': 7, 'y': (42, 64), 'z': {1.5: None}})
{'x': ('x',), 'y': (('y', 0), ('y', 1)), 'z': {1.5: None}}
>>> tree_map_with_path(lambda p, x: p, {'x': 7, 'y': (42, 64), 'z': {1.5: None}},␣
↪none_is_leaf=True)
{'x': ('x',), 'y': (('y', 0), ('y', 1)), 'z': {1.5: ('z', 1.5)}}
```

Parameters

- **func** (*callable*) – A function that takes `2 + len(rests)` arguments, to be applied at the corresponding leaves of the pytrees with extra paths.
- **tree** (*pytree*) – A pytree to be mapped over, with each leaf providing the second positional argument and the corresponding path providing the first positional argument to function `func`.
- **rests** (*tuple of pytree*) – A tuple of pytrees, each of which has the same structure as `tree` or has `tree` as a prefix.
- **is_leaf** (*callable, optional*) – An optionally specified function that will be called at each flattening step. It should return a boolean, with `True` stopping the traversal and the whole subtree being treated as a leaf, and `False` indicating the flattening should traverse the current object.
- **none_is_leaf** (*bool, optional*) – Whether to treat `None` as a leaf. If `False`, `None` is a non-leaf node with arity 0. Thus `None` is contained in the treespec rather than in the leaves list and `None` will be remain in the result pytree. (default: `False`)
- **namespace** (*str, optional*) – The registry namespace used for custom pytree node types. (default: `''`, i.e., the global namespace)

Return type
Union[TypeVar(U), Tuple[PyTree[U], ...], List[PyTree[U]], Dict[Any, PyTree[U]], Deque[PyTree[U]], *CustomTreeNode*[PyTree[U]]]

Returns
A new pytree with the same structure as `tree` but with the value at each leaf given by `func(p, x, *xs)` where `(p, x)` are the path and value at the corresponding leaf in `tree` and `xs` is the tuple of values at corresponding nodes in `rests`.

optree.**tree_map_with_path_**(*func*, *tree*, **rests*, *is_leaf=None*, *none_is_leaf=False*, *namespace=''*)

Like *tree_map_with_path()*, but do an inplace call on each leaf and return the original tree.

See also *tree_map()*, *tree_map_()*, and *tree_map_with_path()*.

Parameters

- **func** (*callable*) – A function that takes `2 + len(rests)` arguments, to be applied at the corresponding leaves of the pytrees with extra paths.
- **tree** (*pytree*) – A pytree to be mapped over, with each leaf providing the second positional argument and the corresponding path providing the first positional argument to function `func`.
- **rests** (*tuple of pytree*) – A tuple of pytrees, each of which has the same structure as `tree` or has `tree` as a prefix.
- **is_leaf** (*callable, optional*) – An optionally specified function that will be called at each flattening step. It should return a boolean, with `True` stopping the traversal and the whole subtree being treated as a leaf, and `False` indicating the flattening should traverse the current object.
- **none_is_leaf** (*bool, optional*) – Whether to treat `None` as a leaf. If `False`, `None` is a non-leaf node with arity 0. Thus `None` is contained in the treespec rather than in the leaves list and `None` will be remain in the result pytree. (default: `False`)
- **namespace** (*str, optional*) – The registry namespace used for custom pytree node types. (default: `''`, i.e., the global namespace)

Return type

`Union`[`TypeVar`(T), `Tuple`[PyTree[T], `...`], `List`[PyTree[T]], `Dict`[`Any`, PyTree[T]], `Deque`[PyTree[T]], *`CustomTreeNode`*[PyTree[T]]]

Returns

The original `tree` with the value at each leaf is given by the side-effect of function `func(p, x, *xs)` (not the return value) where `(p, x)` are the path and value at the corresponding leaf in `tree` and `xs` is the tuple of values at values at corresponding nodes in `rests`.

optree.**tree_map_with_accessor**(*func*, *tree*, **rests*, *is_leaf=None*, *none_is_leaf=False*, *namespace=''*)

Map a multi-input function over pytree args as well as the tree accessors to produce a new pytree.

See also *tree_map()*, *tree_map_()*, and *tree_map_with_accessor_()*.

```
>>> tree_map_with_accessor(lambda a, x: f'{a.codegen("tree")} = {x!r}', {'x': 7, 'y
↪': (42, 64)})
{'x': "tree['x'] = 7", 'y': ("tree['y'][0] = 42", "tree['y'][1] = 64")}
>>> tree_map_with_accessor(lambda a, x: x + len(a), {'x': 7, 'y': (42, 64), 'z':␣
↪None})
{'x': 8, 'y': (44, 66), 'z': None}
>>> tree_map_with_accessor(
...     lambda a, x: a,
...     {'x': 7, 'y': (42, 64), 'z': {1.5: None}},
... )
{
    'x': PyTreeAccessor(*['x'], (MappingEntry(key='x', type=<class 'dict'>),)),
    'y': (
        PyTreeAccessor(*['y'][0], (MappingEntry(key='y', type=<class 'dict'>),␣
↪SequenceEntry(index=0, type=<class 'tuple'>))),
        PyTreeAccessor(*['y'][1], (MappingEntry(key='y', type=<class 'dict'>),␣
```

(continues on next page)

(continued from previous page)

```
↪SequenceEntry(index=1, type=<class 'tuple'>)))
    ),
    'z': {1.5: None}
}
>>> tree_map_with_accessor(
...     lambda a, x: a,
...     {'x': 7, 'y': (42, 64), 'z': {1.5: None}},
...     none_is_leaf=True,
... )
{
    'x': PyTreeAccessor(*['x'], (MappingEntry(key='x', type=<class 'dict'>),)),
    'y': (
        PyTreeAccessor(*['y'][0], (MappingEntry(key='y', type=<class 'dict'>),␣
↪SequenceEntry(index=0, type=<class 'tuple'>))),
        PyTreeAccessor(*['y'][1], (MappingEntry(key='y', type=<class 'dict'>),␣
↪SequenceEntry(index=1, type=<class 'tuple'>)))
    ),
    'z': {
        1.5: PyTreeAccessor(*['z'][1.5], (MappingEntry(key='z', type=<class 'dict'>
↪), MappingEntry(key=1.5, type=<class 'dict'>)))
    }
}
```

Parameters

- **func** (*callable*) – A function that takes `2 + len(rests)` arguments, to be applied at the corresponding leaves of the pytrees with extra accessors.
- **tree** (*pytree*) – A pytree to be mapped over, with each leaf providing the second positional argument and the corresponding path providing the first positional argument to function `func`.
- **rests** (*tuple of pytree*) – A tuple of pytrees, each of which has the same structure as `tree` or has `tree` as a prefix.
- **is_leaf** (*callable, optional*) – An optionally specified function that will be called at each flattening step. It should return a boolean, with `True` stopping the traversal and the whole subtree being treated as a leaf, and `False` indicating the flattening should traverse the current object.
- **none_is_leaf** (*bool, optional*) – Whether to treat `None` as a leaf. If `False`, `None` is a non-leaf node with arity 0. Thus `None` is contained in the treespec rather than in the leaves list and `None` will be remain in the result pytree. (default: `False`)
- **namespace** (*str, optional*) – The registry namespace used for custom pytree node types. (default: `''`, i.e., the global namespace)

Return type

`Union`[`TypeVar`(U), `Tuple`[PyTree[U], `...`], `List`[PyTree[U]], `Dict`[`Any`, PyTree[U]], `Deque`[PyTree[U]], *`CustomTreeNode`*[PyTree[U]]]

Returns

A new pytree with the same structure as `tree` but with the value at each leaf given by `func(a, x, *xs)` where `(a, x)` are the accessor and value at the corresponding leaf in `tree` and `xs` is the tuple of values at corresponding nodes in `rests`.

optree.**tree_map_with_accessor_**(*func*, *tree*, **rests*, *is_leaf=None*, *none_is_leaf=False*, *namespace=''*)

Like *tree_map_with_accessor()*, but do an inplace call on each leaf and return the original tree.

See also *tree_map()*, *tree_map_()*, and *tree_map_with_accessor()*.

Parameters

- **func** (*callable*) – A function that takes `2 + len(rests)` arguments, to be applied at the corresponding leaves of the pytrees with extra accessors.
- **tree** (*pytree*) – A pytree to be mapped over, with each leaf providing the second positional argument and the corresponding path providing the first positional argument to function `func`.
- **rests** (*tuple of pytree*) – A tuple of pytrees, each of which has the same structure as `tree` or has `tree` as a prefix.
- **is_leaf** (*callable, optional*) – An optionally specified function that will be called at each flattening step. It should return a boolean, with `True` stopping the traversal and the whole subtree being treated as a leaf, and `False` indicating the flattening should traverse the current object.
- **none_is_leaf** (*bool, optional*) – Whether to treat `None` as a leaf. If `False`, `None` is a non-leaf node with arity 0. Thus `None` is contained in the treespec rather than in the leaves list and `None` will be remain in the result pytree. (default: `False`)
- **namespace** (*str, optional*) – The registry namespace used for custom pytree node types. (default: `''`, i.e., the global namespace)

Return type

`Union`[`TypeVar`(`T`), `Tuple`[PyTree[T], `...`], `List`[PyTree[T]], `Dict`[`Any`, PyTree[T]], `Deque`[PyTree[T]], *CustomTreeNode*[PyTree[T]]]

Returns

The original `tree` with the value at each leaf is given by the side-effect of function `func(a, x, *xs)` (not the return value) where `(a, x)` are the accessor and value at the corresponding leaf in `tree` and `xs` is the tuple of values at values at corresponding nodes in `rests`.

optree.**tree_replace_nones**(*sentinel*, *tree*, *namespace=''*)

Replace `None` in `tree` with `sentinel`.

See also *tree_flatten()* and *tree_map()*.

```
>>> tree_replace_nones(0, {'a': 1, 'b': None, 'c': (2, None)})
{'a': 1, 'b': 0, 'c': (2, 0)}
>>> tree_replace_nones(0, None)
0
```

Parameters

- **sentinel** (*object*) – The value to replace `None` with.
- **tree** (*pytree*) – A pytree to be transformed.
- **namespace** (*str, optional*) – The registry namespace used for custom pytree node types. (default: `''`, i.e., the global namespace)

Return type

`Union`[`TypeVar`(`T`), `Tuple`[PyTree[T], `...`], `List`[PyTree[T]], `Dict`[`Any`, PyTree[T]], `Deque`[PyTree[T]], *CustomTreeNode*[PyTree[T]]]

Returns

A new pytree with the same structure as `tree` but with `None` replaced.

`optree.tree_transpose`(*outer_treespec*, *inner_treespec*, *tree*, *is_leaf=None*)

Transform a tree having tree structure (outer, inner) into one having structure (inner, outer).

See also *tree_flatten()*, *tree_structure()*, and *tree_transpose_map()*.

```
>>> outer_treespec = tree_structure({'a': 1, 'b': 2, 'c': (3, 4)})
>>> outer_treespec
PyTreeSpec({'a': *, 'b': *, 'c': (*, *)})
>>> inner_treespec = tree_structure((1, 2))
>>> inner_treespec
PyTreeSpec((*, *))
>>> tree = {'a': (1, 2), 'b': (3, 4), 'c': ((5, 6), (7, 8))}
>>> tree_transpose(outer_treespec, inner_treespec, tree)
({'a': 1, 'b': 3, 'c': (5, 7)}, {'a': 2, 'b': 4, 'c': (6, 8)})
```

For performance reasons, this function is only checks for the number of leaves in the input pytree, not the structure. The result is only enumerated up to the original order of leaves in `tree`, then transpose depends on the number of leaves in structure (inner, outer). The caller is responsible for ensuring that the input pytree has a prefix structure of `outer_treespec` followed by a prefix structure of `inner_treespec`. Otherwise, the result may be incorrect.

```
>>> tree_transpose(outer_treespec, inner_treespec, list(range(1, 9)))
({'a': 1, 'b': 3, 'c': (5, 7)}, {'a': 2, 'b': 4, 'c': (6, 8)})
```

Parameters

- **outer_treespec** (PyTreeSpec) – A treespec object representing the outer structure of the pytree.
- **inner_treespec** (PyTreeSpec) – A treespec object representing the inner structure of the pytree.
- **tree** (*pytree*) – A pytree to be transposed.
- **is_leaf** (*callable, optional*) – An optionally specified function that will be called at each flattening step. It should return a boolean, with `True` stopping the traversal and the whole subtree being treated as a leaf, and `False` indicating the flattening should traverse the current object.

Return type

Union[TypeVar(T), Tuple[PyTree[T], ...], List[PyTree[T]], Dict[Any, PyTree[T]], Deque[PyTree[T]], *CustomTreeNode*[PyTree[T]]]

Returns

A new pytree with the same structure as `inner_treespec` but with the value at each leaf has the same structure as `outer_treespec`.

`optree.tree_transpose_map`(*func*, *tree*, **rests*, *inner_treespec=None*, *is_leaf=None*, *none_is_leaf=False*, *namespace=''*)

Map a multi-input function over pytree args to produce a new pytree with transposed structure.

See also *tree_map()*, *tree_map_with_path()*, and *tree_transpose()*.

```
>>> tree = {'b': (2, [3, 4]), 'a': 1, 'c': (5, 6)}
>>> tree_transpose_map(
...     lambda x: {'identity': x, 'double': 2 * x},
```

(continues on next page)

(continued from previous page)

```
...     tree,
... )
{
    'identity': {'b': (2, [3, 4]), 'a': 1, 'c': (5, 6)},
    'double': {'b': (4, [6, 8]), 'a': 2, 'c': (10, 12)}
}
>>> tree_transpose_map(
...     lambda x: {'identity': x, 'double': (x, x)},
...     tree,
... )
{
    'identity': {'b': (2, [3, 4]), 'a': 1, 'c': (5, 6)},
    'double': (
        {'b': (2, [3, 4]), 'a': 1, 'c': (5, 6)},
        {'b': (2, [3, 4]), 'a': 1, 'c': (5, 6)}
    )
}
>>> tree_transpose_map(
...     lambda x: {'identity': x, 'double': (x, x)},
...     tree,
...     inner_treespec=tree_structure({'identity': 0, 'double': 0}),
... )
{
    'identity': {'b': (2, [3, 4]), 'a': 1, 'c': (5, 6)},
    'double': {'b': ((2, 2), [(3, 3), (4, 4)]), 'a': (1, 1), 'c': ((5, 5), (6, 6))}
}
```

Parameters

- **func** (*callable*) – A function that takes `1 + len(rests)` arguments, to be applied at the corresponding leaves of the pytrees.
- **tree** (*pytree*) – A pytree to be mapped over, with each leaf providing the first positional argument to function `func`.
- **rests** (*tuple of pytree*) – A tuple of pytrees, each of which has the same structure as `tree` or has `tree` as a prefix.
- **inner_treespec** (`PyTreeSpec`, *optional*) – The treespec object representing the inner structure of the result pytree. If not specified, the inner structure is inferred from the result of the function `func` on the first leaf. (default: `None`)
- **is_leaf** (*callable, optional*) – An optionally specified function that will be called at each flattening step. It should return a boolean, with `True` stopping the traversal and the whole subtree being treated as a leaf, and `False` indicating the flattening should traverse the current object.
- **none_is_leaf** (*bool, optional*) – Whether to treat `None` as a leaf. If `False`, `None` is a non-leaf node with arity 0. Thus `None` is contained in the treespec rather than in the leaves list and `None` will be remain in the result pytree. (default: `False`)
- **namespace** (*str, optional*) – The registry namespace used for custom pytree node types. (default: `''`, i.e., the global namespace)

Return type

`Union`[`TypeVar`(U), `Tuple`[PyTree[U], `...`], `List`[PyTree[U]], `Dict`[`Any`, PyTree[U]],

Deque[PyTree[U]], *CustomTreeNode*[PyTree[U]]]

Returns

A new nested pytree with the same structure as `inner_treespec` but with the value at each leaf has the same structure as `tree`. The subtree at each leaf is given by the result of function `func(x, *xs)` where `x` is the value at the corresponding leaf in `tree` and `xs` is the tuple of values at corresponding nodes in `rests`.

optree.**tree_transpose_map_with_path**(*func*, *tree*, **rests*, *inner_treespec=None*, *is_leaf=None*, *none_is_leaf=False*, *namespace=''*)

Map a multi-input function over pytree args as well as the tree paths to produce a new pytree with transposed structure.

See also *tree_map_with_path()*, *tree_transpose_map()*, and *tree_transpose()*.

```
>>> tree = {'b': (2, [3, 4]), 'a': 1, 'c': (5, 6)}
>>> tree_transpose_map_with_path(
...     lambda p, x: {'depth': len(p), 'value': x},
...     tree,
... )
{
    'depth': {'b': (2, [3, 3]), 'a': 1, 'c': (2, 2)},
    'value': {'b': (2, [3, 4]), 'a': 1, 'c': (5, 6)}
}
>>> tree_transpose_map_with_path(
...     lambda p, x: {'path': p, 'value': x},
...     tree,
...     inner_treespec=tree_structure({'path': 0, 'value': 0}),
... )
{
    'path': {
        'b': (('b', 0), [('b', 1, 0), ('b', 1, 1)]),
        'a': ('a',),
        'c': (('c', 0), ('c', 1))
    },
    'value': {'b': (2, [3, 4]), 'a': 1, 'c': (5, 6)}
}
```

Parameters

- **func** (*callable*) – A function that takes `2 + len(rests)` arguments, to be applied at the corresponding leaves of the pytrees with extra paths.
- **tree** (*pytree*) – A pytree to be mapped over, with each leaf providing the second positional argument and the corresponding path providing the first positional argument to function `func`.
- **rests** (*tuple of pytree*) – A tuple of pytrees, each of which has the same structure as `tree` or has `tree` as a prefix.
- **inner_treespec** (PyTreeSpec, *optional*) – The treespec object representing the inner structure of the result pytree. If not specified, the inner structure is inferred from the result of the function `func` on the first leaf. (default: `None`)
- **is_leaf** (*callable, optional*) – An optionally specified function that will be called at each flattening step. It should return a boolean, with `True` stopping the traversal and the

whole subtree being treated as a leaf, and `False` indicating the flattening should traverse the current object.

- **none_is_leaf** (*bool, optional*) – Whether to treat `None` as a leaf. If `False`, `None` is a non-leaf node with arity 0. Thus `None` is contained in the treespec rather than in the leaves list and `None` will be remain in the result pytree. (default: `False`)
- **namespace** (*str, optional*) – The registry namespace used for custom pytree node types. (default: `''`, i.e., the global namespace)

Return type

`Union`[`TypeVar`(U), `Tuple`[PyTree[U], `...`], `List`[PyTree[U]], `Dict`[`Any`, PyTree[U]], `Deque`[PyTree[U]], *`CustomTreeNode`*[PyTree[U]]]

Returns

A new nested pytree with the same structure as `inner_treespec` but with the value at each leaf has the same structure as `tree`. The subtree at each leaf is given by the result of function `func(p, x, *xs)` where `(p, x)` are the path and value at the corresponding leaf in `tree` and `xs` is the tuple of values at corresponding nodes in `rests`.

optree.**tree_transpose_map_with_accessor**(*func*, *tree*, **rests*, *inner_treespec=None*, *is_leaf=None*, *none_is_leaf=False*, *namespace=''*)

Map a multi-input function over pytree args as well as the tree accessors to produce a new pytree with transposed structure.

See also *tree_map_with_accessor()*, *tree_transpose_map()*, and *tree_transpose()*.

```
>>> tree = {'b': (2, [3, 4]), 'a': 1, 'c': (5, 6)}
>>> tree_transpose_map_with_accessor(
...     lambda a, x: {'depth': len(a), 'code': a.codegen('tree'), 'value': x},
...     tree,
... )
{
    'depth': {
        'b': (2, [3, 3]),
        'a': 1,
        'c': (2, 2)
    },
    'code': {
        'b': ("tree['b'][0]", ["tree['b'][1][0]", "tree['b'][1][1]"]),
        'a': "tree['a']",
        'c': ("tree['c'][0]", "tree['c'][1]")
    },
    'value': {
        'b': (2, [3, 4]),
        'a': 1,
        'c': (5, 6)
    }
}
>>> tree_transpose_map_with_accessor(
...     lambda a, x: {'path': a.path, 'accessor': a, 'value': x},
...     tree,
...     inner_treespec=tree_structure({'path': 0, 'accessor': 0, 'value': 0}),
... )
{
    'path': {
```

(continues on next page)

(continued from previous page)

```
        'b': (('b', 0), [('b', 1, 0), ('b', 1, 1)]),
        'a': ('a',),
        'c': (('c', 0), ('c', 1))
    },
    'accessor': {
        'b': (
            PyTreeAccessor(*['b'][0], (MappingEntry(key='b', type=<class 'dict'>),␣
↪SequenceEntry(index=0, type=<class 'tuple'>))),
            [
                PyTreeAccessor(*['b'][1][0], (MappingEntry(key='b', type=<class
↪'dict'>), SequenceEntry(index=1, type=<class 'tuple'>), SequenceEntry(index=0,␣
↪type=<class 'list'>))),
                PyTreeAccessor(*['b'][1][1], (MappingEntry(key='b', type=<class
↪'dict'>), SequenceEntry(index=1, type=<class 'tuple'>), SequenceEntry(index=1,␣
↪type=<class 'list'>)))
            ]
        ),
        'a': PyTreeAccessor(*['a'], (MappingEntry(key='a', type=<class 'dict'>),)),
        'c': (
            PyTreeAccessor(*['c'][0], (MappingEntry(key='c', type=<class 'dict'>),␣
↪SequenceEntry(index=0, type=<class 'tuple'>))),
            PyTreeAccessor(*['c'][1], (MappingEntry(key='c', type=<class 'dict'>),␣
↪SequenceEntry(index=1, type=<class 'tuple'>)))
        )
    },
    'value': {'b': (2, [3, 4]), 'a': 1, 'c': (5, 6)}
}
```

Parameters

- **func** (*callable*) – A function that takes `2 + len(rests)` arguments, to be applied at the corresponding leaves of the pytrees with extra accessors.
- **tree** (*pytree*) – A pytree to be mapped over, with each leaf providing the second positional argument and the corresponding path providing the first positional argument to function `func`.
- **rests** (*tuple of pytree*) – A tuple of pytrees, each of which has the same structure as `tree` or has `tree` as a prefix.
- **inner_treespec** (`PyTreeSpec`, *optional*) – The treespec object representing the inner structure of the result pytree. If not specified, the inner structure is inferred from the result of the function `func` on the first leaf. (default: `None`)
- **is_leaf** (*callable*, *optional*) – An optionally specified function that will be called at each flattening step. It should return a boolean, with `True` stopping the traversal and the whole subtree being treated as a leaf, and `False` indicating the flattening should traverse the current object.
- **none_is_leaf** (*bool*, *optional*) – Whether to treat `None` as a leaf. If `False`, `None` is a non-leaf node with arity 0. Thus `None` is contained in the treespec rather than in the leaves list and `None` will be remain in the result pytree. (default: `False`)
- **namespace** (*str*, *optional*) – The registry namespace used for custom pytree node types. (default: `''`, i.e., the global namespace)

Return type
Union[TypeVar(U), Tuple[PyTree[U], ...], List[PyTree[U]], Dict[Any, PyTree[U]], Deque[PyTree[U]], *CustomTreeNode*[PyTree[U]]]

Returns
A new nested pytree with the same structure as `inner_treespec` but with the value at each leaf has the same structure as `tree`. The subtree at each leaf is given by the result of function `func(a, x, *xs)` where `(a, x)` are the accessor and value at the corresponding leaf in `tree` and `xs` is the tuple of values at corresponding nodes in `rests`.

optree.**tree_broadcast_prefix**(*prefix_tree*, *full_tree*, *is_leaf=None*, ***, *none_is_leaf=False*, *namespace=''*)

Return a pytree of same structure of `full_tree` with broadcasted subtrees in `prefix_tree`.

See also *broadcast_prefix()*, *tree_broadcast_common()*, and *treespec_is_prefix()*.

If a `prefix_tree` is a prefix of a `full_tree`, this means the `full_tree` can be constructed by replacing the leaves of `prefix_tree` with appropriate **subtrees**.

This function returns a pytree with the same size as `full_tree`. The leaves are replicated from `prefix_tree`. The number of replicas is determined by the corresponding subtree in `full_tree`.

```
>>> tree_broadcast_prefix(1, [2, 3, 4])
[1, 1, 1]
>>> tree_broadcast_prefix([1, 2, 3], [4, 5, 6])
[1, 2, 3]
>>> tree_broadcast_prefix([1, 2, 3], [4, 5, 6, 7])
Traceback (most recent call last):
    ...
ValueError: list arity mismatch; expected: 3, got: 4; list: [4, 5, 6, 7].
>>> tree_broadcast_prefix([1, 2, 3], [4, 5, (6, 7)])
[1, 2, (3, 3)]
>>> tree_broadcast_prefix([1, 2, 3], [4, 5, {'a': 6, 'b': 7, 'c': (None, 8)}])
[1, 2, {'a': 3, 'b': 3, 'c': (None, 3)}]
>>> tree_broadcast_prefix([1, 2, 3], [4, 5, {'a': 6, 'b': 7, 'c': (None, 8)}], none_
↪is_leaf=True)
[1, 2, {'a': 3, 'b': 3, 'c': (3, 3)}]
```

Parameters

- **prefix_tree** (*pytree*) – A pytree with the prefix structure of `full_tree`.
- **full_tree** (*pytree*) – A pytree with the suffix structure of `prefix_tree`.
- **is_leaf** (*callable, optional*) – An optionally specified function that will be called at each flattening step. It should return a boolean, with `True` stopping the traversal and the whole subtree being treated as a leaf, and `False` indicating the flattening should traverse the current object.
- **none_is_leaf** (*bool, optional*) – Whether to treat `None` as a leaf. If `False`, `None` is a non-leaf node with arity 0. Thus `None` is contained in the treespec rather than in the leaves list and `None` will be remain in the result pytree. (default: `False`)
- **namespace** (*str, optional*) – The registry namespace used for custom pytree node types. (default: `''`, i.e., the global namespace)

Return type
Union[TypeVar(T), Tuple[PyTree[T], ...], List[PyTree[T]], Dict[Any, PyTree[T]], Deque[PyTree[T]], *CustomTreeNode*[PyTree[T]]]

Returns
A pytree of same structure of `full_tree` with broadcasted subtrees in `prefix_tree`.

optree.**broadcast_prefix**(*prefix_tree*, *full_tree*, *is_leaf=None*, ***, *none_is_leaf=False*, *namespace=''*)

Return a list of broadcasted leaves in `prefix_tree` to match the number of leaves in `full_tree`.

See also *tree_broadcast_prefix()*, *broadcast_common()*, and *treespec_is_prefix()*.

If a `prefix_tree` is a prefix of a `full_tree`, this means the `full_tree` can be constructed by replacing the leaves of `prefix_tree` with appropriate **subtrees**.

This function returns a list of leaves with the same size as `full_tree`. The leaves are replicated from `prefix_tree`. The number of replicas is determined by the corresponding subtree in `full_tree`.

```
>>> broadcast_prefix(1, [2, 3, 4])
[1, 1, 1]
>>> broadcast_prefix([1, 2, 3], [4, 5, 6])
[1, 2, 3]
>>> broadcast_prefix([1, 2, 3], [4, 5, 6, 7])
Traceback (most recent call last):
    ...
ValueError: list arity mismatch; expected: 3, got: 4; list: [4, 5, 6, 7].
>>> broadcast_prefix([1, 2, 3], [4, 5, (6, 7)])
[1, 2, 3, 3]
>>> broadcast_prefix([1, 2, 3], [4, 5, {'a': 6, 'b': 7, 'c': (None, 8)}])
[1, 2, 3, 3, 3]
>>> broadcast_prefix([1, 2, 3], [4, 5, {'a': 6, 'b': 7, 'c': (None, 8)}], none_is_
↪leaf=True)
[1, 2, 3, 3, 3, 3]
```

Parameters

- **prefix_tree** (*pytree*) – A pytree with the prefix structure of `full_tree`.
- **full_tree** (*pytree*) – A pytree with the suffix structure of `prefix_tree`.
- **is_leaf** (*callable, optional*) – An optionally specified function that will be called at each flattening step. It should return a boolean, with `True` stopping the traversal and the whole subtree being treated as a leaf, and `False` indicating the flattening should traverse the current object.
- **none_is_leaf** (*bool, optional*) – Whether to treat `None` as a leaf. If `False`, `None` is a non-leaf node with arity 0. Thus `None` is contained in the treespec rather than in the leaves list and `None` will be remain in the result pytree. (default: `False`)
- **namespace** (*str, optional*) – The registry namespace used for custom pytree node types. (default: `''`, i.e., the global namespace)

Return type
`list`[`TypeVar`(T)]

Returns
A list of leaves in `prefix_tree` broadcasted to match the number of leaves in `full_tree`.

optree.**tree_broadcast_common**(*tree*, *other_tree*, *is_leaf=None*, ***, *none_is_leaf=False*, *namespace=''*)

Return two pytrees of common suffix structure of `tree` and `other_tree` with broadcasted subtrees.

See also *broadcast_common()*, *tree_broadcast_prefix()*, and *treespec_is_prefix()*.

If a `suffix_tree` is a suffix of a `tree`, this means the `suffix_tree` can be constructed by replacing the leaves of `tree` with appropriate **subtrees**.

This function returns two pytrees with the same structure. The tree structure is the common suffix structure of `tree` and `other_tree`. The leaves are replicated from `tree` and `other_tree`. The number of replicas is determined by the corresponding subtree in the suffix structure.

```
>>> tree_broadcast_common(1, [2, 3, 4])
([1, 1, 1], [2, 3, 4])
>>> tree_broadcast_common([1, 2, 3], [4, 5, 6])
([1, 2, 3], [4, 5, 6])
>>> tree_broadcast_common([1, 2, 3], [4, 5, 6, 7])
Traceback (most recent call last):
    ...
ValueError: list arity mismatch; expected: 3, got: 4.
>>> tree_broadcast_common([1, (2, 3), 4], [5, 6, (7, 8)])
([1, (2, 3), (4, 4)], [5, (6, 6), (7, 8)])
>>> tree_broadcast_common([1, {'a': (2, 3)}, 4], [5, 6, {'a': 7, 'b': 8, 'c': (None,
↪ 9)}])
([1, {'a': (2, 3)}, {'a': 4, 'b': 4, 'c': (None, 4)}],
 [5, {'a': (6, 6)}, {'a': 7, 'b': 8, 'c': (None, 9)}])
>>> tree_broadcast_common([1, {'a': (2, 3)}, 4], [5, 6, {'a': 7, 'b': 8, 'c': (None,
↪ 9)}], none_is_leaf=True)
([1, {'a': (2, 3)}, {'a': 4, 'b': 4, 'c': (4, 4)}],
 [5, {'a': (6, 6)}, {'a': 7, 'b': 8, 'c': (None, 9)}])
>>> tree_broadcast_common([1, None], [None, 2])
([None, None], [None, None])
>>> tree_broadcast_common([1, None], [None, 2], none_is_leaf=True)
([1, None], [None, 2])
```

Parameters

- **tree** (*pytree*) – A pytree has a common suffix structure of `other_tree`.
- **other_tree** (*pytree*) – A pytree has a common suffix structure of `tree`.
- **is_leaf** (*callable, optional*) – An optionally specified function that will be called at each flattening step. It should return a boolean, with `True` stopping the traversal and the whole subtree being treated as a leaf, and `False` indicating the flattening should traverse the current object.
- **none_is_leaf** (*bool, optional*) – Whether to treat `None` as a leaf. If `False`, `None` is a non-leaf node with arity 0. Thus `None` is contained in the treespec rather than in the leaves list and `None` will be remain in the result pytree. (default: `False`)
- **namespace** (*str, optional*) – The registry namespace used for custom pytree node types. (default: `''`, i.e., the global namespace)

Return type

tuple[Union[TypeVar(T), Tuple[Union[TypeVar(T), Tuple[PyTree[T], ...], List[PyTree[T]], Dict[Any, PyTree[T]], Deque[PyTree[T]], *CustomTreeNode*[PyTree[T]]], ...], List[Union[TypeVar(T), Tuple[PyTree[T], ...], List[PyTree[T]], Dict[Any, PyTree[T]], Deque[PyTree[T]], *CustomTreeNode*[PyTree[T]]]], Dict[Any, Union[TypeVar(T), Tuple[PyTree[T], ...], List[PyTree[T]], Dict[Any, PyTree[T]], Deque[PyTree[T]], *CustomTreeNode*[PyTree[T]]]], Deque[Union[TypeVar(T), Tuple[PyTree[T], ...], List[PyTree[T]], Dict[Any, PyTree[T]], Deque[PyTree[T]], *CustomTreeNode*[PyTree[T]]]], *CustomTreeNode*[Union[TypeVar(T),

Tuple[PyTree[T], ...], List[PyTree[T]], Dict[Any, PyTree[T]], Deque[PyTree[T]], *CustomTreeNode*[PyTree[T]]]]], Union[TypeVar(T), Tuple[Union[TypeVar(T), Tuple[PyTree[T], ...], List[PyTree[T]], Dict[Any, PyTree[T]], Deque[PyTree[T]], *CustomTreeNode*[PyTree[T]]], ...], List[Union[TypeVar(T), Tuple[PyTree[T], ...], List[PyTree[T]], Dict[Any, PyTree[T]], Deque[PyTree[T]], *CustomTreeNode*[PyTree[T]]]], Dict[Any, Union[TypeVar(T), Tuple[PyTree[T], ...], List[PyTree[T]], Dict[Any, PyTree[T]], Deque[PyTree[T]], *CustomTreeNode*[PyTree[T]]]], Deque[Union[TypeVar(T), Tuple[PyTree[T], ...], List[PyTree[T]], Dict[Any, PyTree[T]], Deque[PyTree[T]], *CustomTreeNode*[PyTree[T]]]], *CustomTreeNode*[Union[TypeVar(T), Tuple[PyTree[T], ...], List[PyTree[T]], Dict[Any, PyTree[T]], Deque[PyTree[T]], *CustomTreeNode*[PyTree[T]]]]]

Returns

Two pytrees of common suffix structure of `tree` and `other_tree` with broadcasted subtrees.

optree.**broadcast_common**(*tree*, *other_tree*, *is_leaf=None*, ***, *none_is_leaf=False*, *namespace=''*)

Return two lists of leaves in `tree` and `other_tree` broadcasted to match the number of leaves in the common suffix structure.

See also *tree_broadcast_common()*, *broadcast_prefix()*, and *treespec_is_prefix()*.

If a `suffix_tree` is a suffix of a `tree`, this means the `suffix_tree` can be constructed by replacing the leaves of `tree` with appropriate **subtrees**.

This function returns two pytrees with the same structure. The tree structure is the common suffix structure of `tree` and `other_tree`. The leaves are replicated from `tree` and `other_tree`. The number of replicas is determined by the corresponding subtree in the suffix structure.

```
>>> broadcast_common(1, [2, 3, 4])
([1, 1, 1], [2, 3, 4])
>>> broadcast_common([1, 2, 3], [4, 5, 6])
([1, 2, 3], [4, 5, 6])
>>> broadcast_common([1, 2, 3], [4, 5, 6, 7])
Traceback (most recent call last):
    ...
ValueError: list arity mismatch; expected: 3, got: 4.
>>> broadcast_common([1, (2, 3), 4], [5, 6, (7, 8)])
([1, 2, 3, 4, 4], [5, 6, 6, 7, 8])
>>> broadcast_common([1, {'a': (2, 3)}, 4], [5, 6, {'a': 7, 'b': 8, 'c': (None, 9)}
↪])
([1, 2, 3, 4, 4, 4], [5, 6, 6, 7, 8, 9])
>>> broadcast_common([1, {'a': (2, 3)}, 4], [5, 6, {'a': 7, 'b': 8, 'c': (None, 9)}
↪], none_is_leaf=True)
([1, 2, 3, 4, 4, 4, 4], [5, 6, 6, 7, 8, None, 9])
>>> broadcast_common([1, None], [None, 2])
([], [])
>>> broadcast_common([1, None], [None, 2], none_is_leaf=True)
([1, None], [None, 2])
```

Parameters

- **tree** (*pytree*) – A pytree has a common suffix structure of `other_tree`.
- **other_tree** (*pytree*) – A pytree has a common suffix structure of `tree`.
- **is_leaf** (*callable, optional*) – An optionally specified function that will be called at each flattening step. It should return a boolean, with `True` stopping the traversal and the

whole subtree being treated as a leaf, and `False` indicating the flattening should traverse the current object.

- **none_is_leaf** (*bool, optional*) – Whether to treat `None` as a leaf. If `False`, `None` is a non-leaf node with arity 0. Thus `None` is contained in the treespec rather than in the leaves list and `None` will be remain in the result pytree. (default: `False`)
- **namespace** (*str, optional*) – The registry namespace used for custom pytree node types. (default: `''`, i.e., the global namespace)

Return type
`tuple[list[TypeVar(T)], list[TypeVar(T)]]`

Returns
Two lists of leaves in `tree` and `other_tree` broadcasted to match the number of leaves in the common suffix structure.

optree.**tree_broadcast_map**(*func*, *tree*, **rests*, *is_leaf=None*, *none_is_leaf=False*, *namespace=''*)

Map a multi-input function over pytree args to produce a new pytree.

See also *tree_broadcast_map_with_path()*, *tree_map()*, *tree_map_()*, and *tree_map_with_path()*.

If only one input is provided, this function is the same as *tree_map()*:

```
>>> tree_broadcast_map(lambda x: x + 1, {'x': 7, 'y': (42, 64)})
{'x': 8, 'y': (43, 65)}
>>> tree_broadcast_map(lambda x: x + 1, {'x': 7, 'y': (42, 64), 'z': None})
{'x': 8, 'y': (43, 65), 'z': None}
>>> tree_broadcast_map(lambda x: x is None, {'x': 7, 'y': (42, 64), 'z': None})
{'x': False, 'y': (False, False), 'z': None}
>>> tree_broadcast_map(lambda x: x is None, {'x': 7, 'y': (42, 64), 'z': None},␣
↪none_is_leaf=True)
{'x': False, 'y': (False, False), 'z': True}
```

If multiple inputs are given, all input trees will be broadcasted to the common suffix structure of all inputs:

```
>>> tree_broadcast_map(lambda x, y: x * y, [5, 6, (3, 4)], [{'a': 7, 'b': 9}, [1,␣
↪2], 8])
[{'a': 35, 'b': 45}, [6, 12], (24, 32)]
```

Parameters

- **func** (*callable*) – A function that takes `1 + len(rests)` arguments, to be applied at the corresponding leaves of the pytrees.
- **tree** (*pytree*) – A pytree to be mapped over, with each leaf providing the first positional argument to function `func`.
- **rests** (*tuple of pytree*) – A tuple of pytrees, they should have a common suffix structure with each other and with `tree`.
- **is_leaf** (*callable, optional*) – An optionally specified function that will be called at each flattening step. It should return a boolean, with `True` stopping the traversal and the whole subtree being treated as a leaf, and `False` indicating the flattening should traverse the current object.
- **none_is_leaf** (*bool, optional*) – Whether to treat `None` as a leaf. If `False`, `None` is a non-leaf node with arity 0. Thus `None` is contained in the treespec rather than in the leaves list and `None` will be remain in the result pytree. (default: `False`)

- **namespace** (*str, optional*) – The registry namespace used for custom pytree node types. (default: `''`, i.e., the global namespace)

Return type

Union[TypeVar(U), Tuple[PyTree[U], ...], List[PyTree[U]], Dict[Any, PyTree[U]], Deque[PyTree[U]], *CustomTreeNode*[PyTree[U]]]

Returns

A new pytree with the structure as the common suffix structure of `tree` and `rests` but with the value at each leaf given by `func(x, *xs)` where `x` is the value at the corresponding leaf (may be broadcasted) in `tree` and `xs` is the tuple of values at corresponding leaves (may be broadcasted) in `rests`.

optree.**tree_broadcast_map_with_path**(*func*, *tree*, **rests*, *is_leaf=None*, *none_is_leaf=False*, *namespace=''*)

Map a multi-input function over pytree args as well as the tree paths to produce a new pytree.

See also *tree_broadcast_map()*, *tree_map()*, *tree_map_()*, and *tree_map_with_path()*.

If only one input is provided, this function is the same as *tree_map()*:

```
>>> tree_broadcast_map_with_path(lambda p, x: (len(p), x), {'x': 7, 'y': (42, 64)})
{'x': (1, 7), 'y': ((2, 42), (2, 64))}
>>> tree_broadcast_map_with_path(lambda p, x: x + len(p), {'x': 7, 'y': (42, 64), 'z
↪': None})
{'x': 8, 'y': (44, 66), 'z': None}
>>> tree_broadcast_map_with_path(lambda p, x: p, {'x': 7, 'y': (42, 64), 'z': {1.5:␣
↪None}})
{'x': ('x',), 'y': (('y', 0), ('y', 1)), 'z': {1.5: None}}
>>> tree_broadcast_map_with_path(lambda p, x: p, {'x': 7, 'y': (42, 64), 'z': {1.5:␣
↪None}}, none_is_leaf=True)
{'x': ('x',), 'y': (('y', 0), ('y', 1)), 'z': {1.5: ('z', 1.5)}}
```

If multiple inputs are given, all input trees will be broadcasted to the common suffix structure of all inputs:

```
>>> tree_broadcast_map_with_path(
...     lambda p, x, y: (p, x * y),
...     [5, 6, (3, 4)],
...     [{'a': 7, 'b': 9}, [1, 2], 8],
... )
[
    {'a': ((0, 'a'), 35), 'b': ((0, 'b'), 45)},
    [((1, 0), 6), ((1, 1), 12)],
    (((2, 0), 24), ((2, 1), 32))
]
```

Parameters

- **func** (*callable*) – A function that takes `2 + len(rests)` arguments, to be applied at the corresponding leaves of the pytrees with extra paths.
- **tree** (*pytree*) – A pytree to be mapped over, with each leaf providing the first positional argument to function `func`.
- **rests** (*tuple of pytree*) – A tuple of pytrees, they should have a common suffix structure with each other and with `tree`.
- **is_leaf** (*callable, optional*) – An optionally specified function that will be called at each flattening step. It should return a boolean, with `True` stopping the traversal and the

whole subtree being treated as a leaf, and `False` indicating the flattening should traverse the current object.

- **none_is_leaf** (*bool, optional*) – Whether to treat `None` as a leaf. If `False`, `None` is a non-leaf node with arity 0. Thus `None` is contained in the treespec rather than in the leaves list and `None` will be remain in the result pytree. (default: `False`)
- **namespace** (*str, optional*) – The registry namespace used for custom pytree node types. (default: `''`, i.e., the global namespace)

Return type

`Union`[`TypeVar`(U), `Tuple`[PyTree[U], `...`], `List`[PyTree[U]], `Dict`[`Any`, PyTree[U]], `Deque`[PyTree[U]], *`CustomTreeNode`*[PyTree[U]]]

Returns

A new pytree with the structure as the common suffix structure of `tree` and `rests` but with the value at each leaf given by `func(p, x, *xs)` where `(p, x)` are the path and value at the corresponding leaf (may be broadcasted) in and `xs` is the tuple of values at corresponding leaves (may be broadcasted) in `rests`.

optree.**tree_broadcast_map_with_accessor**(*func*, *tree*, **rests*, *is_leaf=None*, *none_is_leaf=False*, *namespace=''*)

Map a multi-input function over pytree args as well as the tree accessors to produce a new pytree.

See also *tree_broadcast_map()*, *tree_map()*, *tree_map_()*, and *tree_map_with_accessor()*.

If only one input is provided, this function is the same as *tree_map()*:

```
>>> tree_broadcast_map_with_accessor(lambda a, x: (len(a), x), {'x': 7, 'y': (42,␣
↪64)})
{'x': (1, 7), 'y': ((2, 42), (2, 64))}
>>> tree_broadcast_map_with_accessor(lambda a, x: x + len(a), {'x': 7, 'y': (42,␣
↪64), 'z': None})
{'x': 8, 'y': (44, 66), 'z': None}
>>> tree_broadcast_map_with_accessor(
...     lambda a, x: a.codegen('tree'),
...     {'x': 7, 'y': (42, 64), 'z': {1.5: None}},
... )
{
    'x': "tree['x']",
    'y': ("tree['y'][0]", "tree['y'][1]"),
    'z': {1.5: None}
}
>>> tree_broadcast_map_with_accessor(
...     lambda a, x: a.codegen('tree'),
...     {'x': 7, 'y': (42, 64), 'z': {1.5: None}},
...     none_is_leaf=True,
... )
{
    'x': "tree['x']",
    'y': ("tree['y'][0]", "tree['y'][1]"),
    'z': {1.5: "tree['z'][1.5]"}
}
```

If multiple inputs are given, all input trees will be broadcasted to the common suffix structure of all inputs:

```
>>> tree_broadcast_map_with_accessor(
...     lambda a, x, y: f'{a.codegen("tree")} = {x * y}',
...     [5, 6, (3, 4)],
...     [{'a': 7, 'b': 9}, [1, 2], 8],
... )
[
    {'a': "tree[0]['a'] = 35", 'b': "tree[0]['b'] = 45"},
    ['tree[1][0] = 6', 'tree[1][1] = 12'],
    ('tree[2][0] = 24', 'tree[2][1] = 32')
]
```

Parameters

- **func** (*callable*) – A function that takes `2 + len(rests)` arguments, to be applied at the corresponding leaves of the pytrees with extra accessors.
- **tree** (*pytree*) – A pytree to be mapped over, with each leaf providing the first positional argument to function `func`.
- **rests** (*tuple of pytree*) – A tuple of pytrees, they should have a common suffix structure with each other and with `tree`.
- **is_leaf** (*callable, optional*) – An optionally specified function that will be called at each flattening step. It should return a boolean, with `True` stopping the traversal and the whole subtree being treated as a leaf, and `False` indicating the flattening should traverse the current object.
- **none_is_leaf** (*bool, optional*) – Whether to treat `None` as a leaf. If `False`, `None` is a non-leaf node with arity 0. Thus `None` is contained in the treespec rather than in the leaves list and `None` will be remain in the result pytree. (default: `False`)
- **namespace** (*str, optional*) – The registry namespace used for custom pytree node types. (default: `''`, i.e., the global namespace)

Return type

`Union`[`TypeVar`(U), `Tuple`[PyTree[U], `...`], `List`[PyTree[U]], `Dict`[`Any`, PyTree[U]], `Deque`[PyTree[U]], *`CustomTreeNode`*[PyTree[U]]]

Returns

A new pytree with the structure as the common suffix structure of `tree` and `rests` but with the value at each leaf given by `func(a, x, *xs)` where `(a, x)` are the accessor and value at the corresponding leaf (may be broadcasted) in and `xs` is the tuple of values at corresponding leaves (may be broadcasted) in `rests`.

optree.**tree_reduce**(*func*, *tree*, *initial=<MISSING>*, ***, *is_leaf=None*, *none_is_leaf=False*, *namespace=''*)

Traversal through a pytree and reduce the leaves in left-to-right depth-first order.

See also *tree_leaves()* and *tree_sum()*.

```
>>> tree_reduce(lambda x, y: x + y, {'x': 1, 'y': (2, 3)})
6
>>> tree_reduce(lambda x, y: x + y, {'x': 1, 'y': (2, None), 'z': 3})  # `None` is a␣
↪non-leaf node with arity 0 by default
6
>>> tree_reduce(lambda x, y: x and y, {'x': 1, 'y': (2, None), 'z': 3})
3
>>> tree_reduce(lambda x, y: x and y, {'x': 1, 'y': (2, None), 'z': 3}, none_is_
```

(continues on next page)

(continued from previous page)

```
↪leaf=True)
None
```

Parameters

- **func** (*callable*) – A function that takes two arguments and returns a value of the same type.
- **tree** (*pytree*) – A pytree to be traversed.
- **initial** (*object, optional*) – An initial value to be used for the reduction. If not provided, the first leaf value is used as the initial value.
- **is_leaf** (*callable, optional*) – An optionally specified function that will be called at each flattening step. It should return a boolean, with `True` stopping the traversal and the whole subtree being treated as a leaf, and `False` indicating the flattening should traverse the current object.
- **none_is_leaf** (*bool, optional*) – Whether to treat `None` as a leaf. If `False`, `None` is a non-leaf node with arity 0. Thus `None` is contained in the treespec rather than in the leaves list and `None` will be remain in the result pytree. (default: `False`)
- **namespace** (*str, optional*) – The registry namespace used for custom pytree node types. (default: `''`, i.e., the global namespace)

Return type
`TypeVar`(T)

Returns
The result of reducing the leaves of the pytree using `func`.

optree.**tree_sum**(*tree*, *start=0*, *, *is_leaf=None*, *none_is_leaf=False*, *namespace=''*)

Sum `start` and leaf values in `tree` in left-to-right depth-first order and return the total.

See also *tree_leaves()* and *tree_reduce()*.

```
>>> tree_sum({'x': 1, 'y': (2, 3)})
6
>>> tree_sum({'x': 1, 'y': (2, None), 'z': 3})  # `None` is a non-leaf node with␣
↪arity 0 by default
6
>>> tree_sum({'x': 1, 'y': (2, None), 'z': 3}, none_is_leaf=True)
Traceback (most recent call last):
    ...
TypeError: unsupported operand type(s) for +: 'int' and 'NoneType'
>>> tree_sum({'x': 'a', 'y': ('b', None), 'z': 'c'}, start='')
'abc'
>>> tree_sum({'x': [1], 'y': ([2], [None]), 'z': [3]}, start=[], is_leaf=lambda x:␣
↪isinstance(x, list))
[1, 2, None, 3]
```

Parameters

- **tree** (*pytree*) – A pytree to be traversed.
- **start** (*object, optional*) – An initial value to be used for the sum. (default: `0`)

- **is_leaf** (*callable, optional*) – An optionally specified function that will be called at each flattening step. It should return a boolean, with `True` stopping the traversal and the whole subtree being treated as a leaf, and `False` indicating the flattening should traverse the current object.
- **none_is_leaf** (*bool, optional*) – Whether to treat `None` as a leaf. If `False`, `None` is a non-leaf node with arity 0. Thus `None` is contained in the treespec rather than in the leaves list and `None` will be remain in the result pytree. (default: `False`)
- **namespace** (*str, optional*) – The registry namespace used for custom pytree node types. (default: `''`, i.e., the global namespace)

Return type
`TypeVar`(T)

Returns
The total sum of `start` and leaf values in `tree`.

optree.**tree_max**(*tree*, *, *default=<MISSING>*, *key=None*, *is_leaf=None*, *none_is_leaf=False*, *namespace=''*)

Return the maximum leaf value in `tree`.

See also *tree_leaves()* and *tree_min()*.

```
>>> tree_max({})
Traceback (most recent call last):
    ...
ValueError: max() iterable argument is empty
>>> tree_max({}, default=0)
0
>>> tree_max({'x': 0, 'y': (2, 1)})
2
>>> tree_max({'x': 0, 'y': (2, 1)}, key=lambda x: -x)
0
>>> tree_max({'a': None})  # `None` is a non-leaf node with arity 0 by default
Traceback (most recent call last):
    ...
ValueError: max() iterable argument is empty
>>> tree_max({'a': None}, default=0)  # `None` is a non-leaf node with arity 0 by␣
↪default
0
>>> tree_max({'a': None}, none_is_leaf=True)
None
>>> tree_max(None)  # `None` is a non-leaf node with arity 0 by default
Traceback (most recent call last):
    ...
ValueError: max() iterable argument is empty
>>> tree_max(None, default=0)
0
>>> tree_max(None, none_is_leaf=True)
None
```

Parameters

- **tree** (*pytree*) – A pytree to be traversed.
- **default** (*object, optional*) – The default value to return if `tree` is empty. If the `tree` is empty and `default` is not specified, raise a `ValueError`.

- **key** (*callable or None, optional*) – An one argument ordering function like that used for `list.sort()`.
- **is_leaf** (*callable, optional*) – An optionally specified function that will be called at each flattening step. It should return a boolean, with `True` stopping the traversal and the whole subtree being treated as a leaf, and `False` indicating the flattening should traverse the current object.
- **none_is_leaf** (*bool, optional*) – Whether to treat `None` as a leaf. If `False`, `None` is a non-leaf node with arity 0. Thus `None` is contained in the treespec rather than in the leaves list and `None` will be remain in the result pytree. (default: `False`)
- **namespace** (*str, optional*) – The registry namespace used for custom pytree node types. (default: `''`, i.e., the global namespace)

Return type
`TypeVar`(T)

Returns
The maximum leaf value in `tree`.

optree.**tree_min**(*tree*, *, *default=<MISSING>*, *key=None*, *is_leaf=None*, *none_is_leaf=False*, *namespace=''*)

Return the minimum leaf value in `tree`.

See also *tree_leaves()* and *tree_max()*.

```
>>> tree_min({})
Traceback (most recent call last):
    ...
ValueError: min() iterable argument is empty
>>> tree_min({}, default=0)
0
>>> tree_min({'x': 0, 'y': (2, 1)})
0
>>> tree_min({'x': 0, 'y': (2, 1)}, key=lambda x: -x)
2
>>> tree_min({'a': None})  # `None` is a non-leaf node with arity 0 by default
Traceback (most recent call last):
    ...
ValueError: min() iterable argument is empty
>>> tree_min({'a': None}, default=0)  # `None` is a non-leaf node with arity 0 by␣
↪default
0
>>> tree_min({'a': None}, none_is_leaf=True)
None
>>> tree_min(None)  # `None` is a non-leaf node with arity 0 by default
Traceback (most recent call last):
    ...
ValueError: min() iterable argument is empty
>>> tree_min(None, default=0)
0
>>> tree_min(None, none_is_leaf=True)
None
```

Parameters

- **tree** (*pytree*) – A pytree to be traversed.

- **default** (*object, optional*) – The default value to return if `tree` is empty. If the `tree` is empty and `default` is not specified, raise a `ValueError`.
- **key** (*callable or None, optional*) – An one argument ordering function like that used for `list.sort()`.
- **is_leaf** (*callable, optional*) – An optionally specified function that will be called at each flattening step. It should return a boolean, with `True` stopping the traversal and the whole subtree being treated as a leaf, and `False` indicating the flattening should traverse the current object.
- **none_is_leaf** (*bool, optional*) – Whether to treat `None` as a leaf. If `False`, `None` is a non-leaf node with arity 0. Thus `None` is contained in the treespec rather than in the leaves list and `None` will be remain in the result pytree. (default: `False`)
- **namespace** (*str, optional*) – The registry namespace used for custom pytree node types. (default: `''`, i.e., the global namespace)

Return type
`TypeVar`(T)

Returns
The minimum leaf value in `tree`.

`optree.tree_all`(*tree*, *, *is_leaf=None*, *none_is_leaf=False*, *namespace=''*)

Test whether all leaves in `tree` are true (or if `tree` is empty).

See also *tree_leaves()* and *tree_any()*.

```
>>> tree_all({})
True
>>> tree_all({'x': 1, 'y': (2, 3)})
True
>>> tree_all({'x': 1, 'y': (2, None), 'z': 3})  # `None` is a non-leaf node by␣
↪default
True
>>> tree_all({'x': 1, 'y': (2, None), 'z': 3}, none_is_leaf=True)
False
>>> tree_all(None)  # `None` is a non-leaf node by default
True
>>> tree_all(None, none_is_leaf=True)
False
```

Parameters

- **tree** (*pytree*) – A pytree to be traversed.
- **is_leaf** (*callable, optional*) – An optionally specified function that will be called at each flattening step. It should return a boolean, with `True` stopping the traversal and the whole subtree being treated as a leaf, and `False` indicating the flattening should traverse the current object.
- **none_is_leaf** (*bool, optional*) – Whether to treat `None` as a leaf. If `False`, `None` is a non-leaf node with arity 0. Thus `None` is contained in the treespec rather than in the leaves list and `None` will be remain in the result pytree. (default: `False`)
- **namespace** (*str, optional*) – The registry namespace used for custom pytree node types. (default: `''`, i.e., the global namespace)

Return type
bool

Returns
True if all leaves in tree are true, or if tree is empty. Otherwise, False.

optree.**tree_any**(*tree*, *, *is_leaf=None*, *none_is_leaf=False*, *namespace=''*)

Test whether all leaves in tree are true (or False if tree is empty).

See also *tree_leaves()* and *tree_all()*.

```
>>> tree_any({})
False
>>> tree_any({'x': 0, 'y': (2, 0)})
True
>>> tree_any({'a': None})  # `None` is a non-leaf node with arity 0 by default
False
>>> tree_any({'a': None}, none_is_leaf=True)  # `None` is evaluated as false
False
>>> tree_any(None)  # `None` is a non-leaf node with arity 0 by default
False
>>> tree_any(None, none_is_leaf=True)  # `None` is evaluated as false
False
```

Parameters

- **tree** (*pytree*) – A pytree to be traversed.
- **is_leaf** (*callable, optional*) – An optionally specified function that will be called at each flattening step. It should return a boolean, with True stopping the traversal and the whole subtree being treated as a leaf, and False indicating the flattening should traverse the current object.
- **none_is_leaf** (*bool, optional*) – Whether to treat None as a leaf. If False, None is a non-leaf node with arity 0. Thus None is contained in the treespec rather than in the leaves list and None will be remain in the result pytree. (default: False)
- **namespace** (*str, optional*) – The registry namespace used for custom pytree node types. (default: '', i.e., the global namespace)

Return type
bool

Returns
True if any leaves in tree are true, otherwise, False. If tree is empty, return False.

optree.**tree_flatten_one_level**(*tree*, *is_leaf=None*, *, *none_is_leaf=False*, *namespace=''*)

Flatten the pytree one level, returning a 4-tuple of children, auxiliary data, path entries, and an unflatten function.

See also *tree_flatten()*, *tree_flatten_with_path()*.

```
>>> children, metadata, entries, unflatten_func = tree_flatten_one_level({'b': (2,␣
↪[3, 4]), 'a': 1, 'c': None, 'd': 5})
>>> children, metadata, entries
([1, (2, [3, 4]), None, 5], ['a', 'b', 'c', 'd'], ('a', 'b', 'c', 'd'))
>>> unflatten_func(metadata, children)
{'a': 1, 'b': (2, [3, 4]), 'c': None, 'd': 5}
>>> children, metadata, entries, unflatten_func = tree_flatten_one_level([{'a': 1,
```

(continues on next page)

(continued from previous page)

```
↪'b': (2, 3)}, (4, 5)])
>>> children, metadata, entries
([{'a': 1, 'b': (2, 3)}, (4, 5)], None, (0, 1))
>>> unflatten_func(metadata, children)
[{'a': 1, 'b': (2, 3)}, (4, 5)]
```

Parameters

- **tree** (*pytree*) – A pytree to be traversed.
- **is_leaf** (*callable, optional*) – An optionally specified function that will be called at each flattening step. It should return a boolean, with `True` stopping the traversal and the whole subtree being treated as a leaf, and `False` indicating the flattening should traverse the current object.
- **none_is_leaf** (*bool, optional*) – Whether to treat `None` as a leaf. If `False`, `None` is a non-leaf node with arity 0. Thus `None` is contained in the treespec rather than in the leaves list and `None` will be remain in the result pytree. (default: `False`)
- **namespace** (*str, optional*) – The registry namespace used for custom pytree node types. (default: `''`, i.e., the global namespace)

Return type

tuple[list[Union[TypeVar(T), Tuple[Union[TypeVar(T), Tuple[PyTree[T], ...], List[PyTree[T]], Dict[Any, PyTree[T]], Deque[PyTree[T]], *CustomTreeNode*[PyTree[T]]], ...], List[Union[TypeVar(T), Tuple[PyTree[T], ...], List[PyTree[T]], Dict[Any, PyTree[T]], Deque[PyTree[T]], *CustomTreeNode*[PyTree[T]]]], Dict[Any, Union[TypeVar(T), Tuple[PyTree[T], ...], List[PyTree[T]], Dict[Any, PyTree[T]], Deque[PyTree[T]], *CustomTreeNode*[PyTree[T]]]], Deque[Union[TypeVar(T), Tuple[PyTree[T], ...], List[PyTree[T]], Dict[Any, PyTree[T]], Deque[PyTree[T]], *CustomTreeNode*[PyTree[T]]]], *CustomTreeNode*[Union[TypeVar(T), Tuple[PyTree[T], ...], List[PyTree[T]], Dict[Any, PyTree[T]], Deque[PyTree[T]], *CustomTreeNode*[PyTree[T]]]]], Optional[TypeVar(_MetaData, bound= Hashable)], tuple[Any, ...], Callable[[Optional[TypeVar(_MetaData, bound= Hashable)], list[Union[TypeVar(T), Tuple[Union[TypeVar(T), Tuple[PyTree[T], ...], List[PyTree[T]], Dict[Any, PyTree[T]], Deque[PyTree[T]], *CustomTreeNode*[PyTree[T]]], ...], List[Union[TypeVar(T), Tuple[PyTree[T], ...], List[PyTree[T]], Dict[Any, PyTree[T]], Deque[PyTree[T]], *CustomTreeNode*[PyTree[T]]]], Dict[Any, Union[TypeVar(T), Tuple[PyTree[T], ...], List[PyTree[T]], Dict[Any, PyTree[T]], Deque[PyTree[T]], *CustomTreeNode*[PyTree[T]]]], Deque[Union[TypeVar(T), Tuple[PyTree[T], ...], List[PyTree[T]], Dict[Any, PyTree[T]], Deque[PyTree[T]], *CustomTreeNode*[PyTree[T]]]], *CustomTreeNode*[Union[TypeVar(T), Tuple[PyTree[T], ...], List[PyTree[T]], Dict[Any, PyTree[T]], Deque[PyTree[T]], *CustomTreeNode*[PyTree[T]]]]], Union[TypeVar(T), Tuple[Union[TypeVar(T), Tuple[PyTree[T], ...], List[PyTree[T]], Dict[Any, PyTree[T]], Deque[PyTree[T]], *CustomTreeNode*[PyTree[T]]], ...], List[Union[TypeVar(T), Tuple[PyTree[T], ...], List[PyTree[T]], Dict[Any, PyTree[T]], Deque[PyTree[T]], *CustomTreeNode*[PyTree[T]]]], Dict[Any, Union[TypeVar(T), Tuple[PyTree[T], ...], List[PyTree[T]], Dict[Any, PyTree[T]], Deque[PyTree[T]], *CustomTreeNode*[PyTree[T]]]], Deque[Union[TypeVar(T), Tuple[PyTree[T], ...], List[PyTree[T]], Dict[Any, PyTree[T]], Deque[PyTree[T]], *CustomTreeNode*[PyTree[T]]]], *CustomTreeNode*[Union[TypeVar(T), Tuple[PyTree[T], ...], List[PyTree[T]], Dict[Any, PyTree[T]], Deque[PyTree[T]], *CustomTreeNode*[PyTree[T]]]]]]]

Returns

A 4-tuple (children, metadata, entries, unflatten_func). The first element is a list of one-level children of the pytree node. The second element is the auxiliary data used to reconstruct the pytree node. The third element is a tuple of path entries to the children. The fourth element is a function that can be used to unflatten the auxiliary data and children back to the pytree node.

optree.**prefix_errors**(*prefix_tree*, *full_tree*, *is_leaf=None*, *, *none_is_leaf=False*, *namespace=''*)

Return a list of errors that would be raised by *broadcast_prefix()*.

Return type
list[Callable[[str], ValueError]]

optree.**treespec_paths**(*treespec*)

Return a list of paths to the leaves of a treespec.

See also *tree_flatten_with_path()*, *tree_paths()*, and *PyTreeSpec.paths()*.

Return type
list[tuple[Any, ...]]

optree.**treespec_accessors**(*treespec*)

Return a list of accessors to the leaves of a treespec.

See also *tree_flatten_with_accessor()*, *tree_accessors()* and *PyTreeSpec.accessors()*.

Return type
list[PyTreeAccessor]

optree.**treespec_entries**(*treespec*)

Return a list of one-level entries of a treespec to its children.

See also *treespec_entry()*, *treespec_paths()*, *treespec_children()*, and *PyTreeSpec.entries()*.

Return type
list[Any]

optree.**treespec_entry**(*treespec*, *index*)

Return the entry of a treespec at the given index.

See also *treespec_entries()*, *treespec_children()*, and *PyTreeSpec.entry()*.

Return type
Any

optree.**treespec_children**(*treespec*)

Return a list of treespecs for the children of a treespec.

See also *treespec_child()*, *treespec_paths()*, *treespec_entries()*, and *PyTreeSpec.children()*.

Return type
list[*PyTreeSpec*]

optree.**treespec_child**(*treespec*, *index*)

Return the treespec of the child of a treespec at the given index.

See also *treespec_children()*, *treespec_entries()*, and *PyTreeSpec.child()*.

Return type
PyTreeSpec

optree.**treespec_is_leaf**(*treespec*, *strict=True*)

Return whether the treespec is a leaf that has no children.

See also *treespec_is_strict_leaf()* and *PyTreeSpec.is_leaf()*.

This function is equivalent to `treespec.is_leaf(strict=strict)`. If `strict=False`, it will return `True` if and only if the treespec represents a strict leaf. If `strict=False`, it will return `True` if the treespec represents a strict leaf or `None` or an empty container (e.g., an empty tuple).

```
>>> treespec_is_leaf(tree_structure(1))
True
>>> treespec_is_leaf(tree_structure((1, 2)))
False
>>> treespec_is_leaf(tree_structure(None))
False
>>> treespec_is_leaf(tree_structure(None), strict=False)
True
>>> treespec_is_leaf(tree_structure(None, none_is_leaf=False))
False
>>> treespec_is_leaf(tree_structure(None, none_is_leaf=True))
True
>>> treespec_is_leaf(tree_structure(()))
False
>>> treespec_is_leaf(tree_structure(()), strict=False)
True
>>> treespec_is_leaf(tree_structure([]))
False
>>> treespec_is_leaf(tree_structure([]), strict=False)
True
```

Parameters

- **treespec** (PyTreeSpec) – A treespec.
- **strict** (*bool, optional*) – Whether not to treat `None` or an empty container (e.g., an empty tuple) as a leaf. (default: `True`)

Return type

bool

Returns

`True` if the treespec represents a leaf that has no children, otherwise, `False`.

optree.**treespec_is_strict_leaf**(*treespec*)

Return whether the treespec is a strict leaf.

See also *treespec_is_leaf()* and *PyTreeSpec.is_leaf()*.

This function respects the `none_is_leaf` setting in the treespec. It is equivalent to `treespec.is_leaf(strict=True)`. It will return `True` if and only if the treespec represents a strict leaf.

```
>>> treespec_is_strict_leaf(tree_structure(1))
True
>>> treespec_is_strict_leaf(tree_structure((1, 2)))
False
>>> treespec_is_strict_leaf(tree_structure(None))
False
```

(continues on next page)

(continued from previous page)

```
>>> treespec_is_strict_leaf(tree_structure(None, none_is_leaf=False))
False
>>> treespec_is_strict_leaf(tree_structure(None, none_is_leaf=True))
True
>>> treespec_is_strict_leaf(tree_structure(()))
False
>>> treespec_is_strict_leaf(tree_structure([]))
False
```

Parameters
treespec (PyTreeSpec) – A treespec.

Return type
bool

Returns
True if the treespec represents a strict leaf, otherwise, False.

optree.**treespec_is_prefix**(*treespec*, *other_treespec*, *strict=False*)

Return whether treespec is a prefix of other_treespec.

See also *treespec_is_prefix()* and *PyTreeSpec.is_prefix()*.

Return type
bool

optree.**treespec_is_suffix**(*treespec*, *other_treespec*, *strict=False*)

Return whether treespec is a suffix of other_treespec.

See also *treespec_is_suffix()* *PyTreeSpec.is_suffix()*.

Return type
bool

optree.**treespec_leaf**(*, *none_is_leaf=False*, *namespace=''*)

Make a treespec representing a leaf node.

See also *tree_structure()*, *treespec_none()*, and *treespec_tuple()*.

```
>>> treespec_leaf()
PyTreeSpec(*)
>>> treespec_leaf(none_is_leaf=True)
PyTreeSpec(*, NoneIsLeaf)
>>> treespec_leaf(none_is_leaf=False) == treespec_leaf(none_is_leaf=True)
False
>>> treespec_leaf() == tree_structure(1)
True
>>> treespec_leaf(none_is_leaf=True) == tree_structure(1, none_is_leaf=True)
True
>>> treespec_leaf(none_is_leaf=True) == tree_structure(None, none_is_leaf=True)
True
>>> treespec_leaf(none_is_leaf=True) == tree_structure(None, none_is_leaf=False)
False
>>> treespec_leaf(none_is_leaf=True) == treespec_none(none_is_leaf=True)
True
>>> treespec_leaf(none_is_leaf=True) == treespec_none(none_is_leaf=False)
```

(continues on next page)

(continued from previous page)

```
False
>>> treespec_leaf(none_is_leaf=False) == treespec_none(none_is_leaf=True)
False
>>> treespec_leaf(none_is_leaf=False) == treespec_none(none_is_leaf=False)
False
```

Parameters

- **none_is_leaf** (*bool, optional*) – Whether to treat `None` as a leaf. If `False`, `None` is a non-leaf node with arity 0. Thus `None` is contained in the treespec rather than in the leaves list and `None` will be remain in the result pytree. (default: `False`)
- **namespace** (*str, optional*) – The registry namespace used for custom pytree node types. (default: `''`, i.e., the global namespace)

Return type
PyTreeSpec

Returns
A treespec representing a leaf node.

optree.**treespec_none**(*, *none_is_leaf=False*, *namespace=''*)

Make a treespec representing a `None` node.

See also *tree_structure()*, *treespec_leaf()*, and *treespec_tuple()*.

```
>>> treespec_none()
PyTreeSpec(None)
>>> treespec_none(none_is_leaf=True)
PyTreeSpec(*, NoneIsLeaf)
>>> treespec_none(none_is_leaf=False) == treespec_none(none_is_leaf=True)
False
>>> treespec_none() == tree_structure(None)
True
>>> treespec_none() == tree_structure(1)
False
>>> treespec_none(none_is_leaf=True) == tree_structure(1, none_is_leaf=True)
True
>>> treespec_none(none_is_leaf=True) == tree_structure(None, none_is_leaf=True)
True
>>> treespec_none(none_is_leaf=True) == tree_structure(None, none_is_leaf=False)
False
>>> treespec_none(none_is_leaf=True) == treespec_leaf(none_is_leaf=True)
True
>>> treespec_none(none_is_leaf=False) == treespec_leaf(none_is_leaf=True)
False
>>> treespec_none(none_is_leaf=True) == treespec_leaf(none_is_leaf=False)
False
>>> treespec_none(none_is_leaf=False) == treespec_leaf(none_is_leaf=False)
False
```

Parameters

- **none_is_leaf** (*bool, optional*) – Whether to treat `None` as a leaf. If `False`, `None` is a non-leaf node with arity 0. Thus `None` is contained in the treespec rather than in the leaves

list and None will be remain in the result pytree. (default: False)

- **namespace** (*str, optional*) – The registry namespace used for custom pytree node types. (default: '', i.e., the global namespace)

Return type
PyTreeSpec

Returns
A treespec representing a None node.

optree.**treespec_tuple**(*iterable=()*, *, *none_is_leaf=False*, *namespace=''*)

Make a tuple treespec from a list of child treespecs.

See also *tree_structure()*, *treespec_leaf()*, and *treespec_none()*.

```
>>> treespec_tuple([treespec_leaf(), treespec_leaf()])
PyTreeSpec((*, *))
>>> treespec_tuple([treespec_leaf(), treespec_leaf(), treespec_none()])
PyTreeSpec((*, *, None))
>>> treespec_tuple()
PyTreeSpec(())
>>> treespec_tuple([treespec_leaf(), treespec_tuple([treespec_leaf(), treespec_
↪leaf()])])
PyTreeSpec((*, (*, *)))
>>> treespec_tuple([treespec_leaf(), tree_structure({'a': 1, 'b': 2})])
PyTreeSpec((*, {'a': *, 'b': *}))
>>> treespec_tuple([treespec_leaf(), tree_structure({'a': 1, 'b': 2}, none_is_
↪leaf=True)])
Traceback (most recent call last):
    ...
ValueError: Expected treespec(s) with `node_is_leaf=False`.
```

Parameters

- **iterable** (*iterable of PyTreeSpec, optional*) – A iterable of child treespecs. They must have the same node_is_leaf and namespace values.
- **none_is_leaf** (*bool, optional*) – Whether to treat None as a leaf. If False, None is a non-leaf node with arity 0. Thus None is contained in the treespec rather than in the leaves list and None will be remain in the result pytree. (default: False)
- **namespace** (*str, optional*) – The registry namespace used for custom pytree node types. (default: '', i.e., the global namespace)

Return type
PyTreeSpec

Returns
A treespec representing a tuple node with the given children.

optree.**treespec_list**(*iterable=()*, *, *none_is_leaf=False*, *namespace=''*)

Make a list treespec from a list of child treespecs.

See also *tree_structure()*, *treespec_leaf()*, and *treespec_none()*.

```
>>> treespec_list([treespec_leaf(), treespec_leaf()])
PyTreeSpec([*, *])
>>> treespec_list([treespec_leaf(), treespec_leaf(), treespec_none()])
```

(continues on next page)

(continued from previous page)

```
PyTreeSpec([*, *, None])
>>> treespec_list()
PyTreeSpec([])
>>> treespec_list([treespec_leaf(), treespec_tuple([treespec_leaf(), treespec_
↪leaf()])])
PyTreeSpec([*, (*, *)])
>>> treespec_list([treespec_leaf(), tree_structure({'a': 1, 'b': 2})])
PyTreeSpec([*, {'a': *, 'b': *}])
>>> treespec_list([treespec_leaf(), tree_structure({'a': 1, 'b': 2}, none_is_
↪leaf=True)])
Traceback (most recent call last):
    ...
ValueError: Expected treespec(s) with `node_is_leaf=False`.
```

Parameters

- **iterable** (*iterable of PyTreeSpec, optional*) – A iterable of child treespecs. They must have the same `node_is_leaf` and `namespace` values.
- **none_is_leaf** (*bool, optional*) – Whether to treat `None` as a leaf. If `False`, `None` is a non-leaf node with arity 0. Thus `None` is contained in the treespec rather than in the leaves list and `None` will be remain in the result pytree. (default: `False`)
- **namespace** (*str, optional*) – The registry namespace used for custom pytree node types. (default: `''`, i.e., the global namespace)

Return type

PyTreeSpec

Returns

A treespec representing a list node with the given children.

optree.**treespec_dict**(*mapping=()*, *, *none_is_leaf=False*, *namespace=''*, ***kwargs*)

Make a dict treespec from a dict of child treespecs.

See also *tree_structure()*, *treespec_leaf()*, and *treespec_none()*.

```
>>> treespec_dict({'a': treespec_leaf(), 'b': treespec_leaf()})
PyTreeSpec({'a': *, 'b': *})
>>> treespec_dict([('b', treespec_leaf()), ('c', treespec_leaf()), ('a', treespec_
↪none())])
PyTreeSpec({'a': None, 'b': *, 'c': *})
>>> treespec_dict()
PyTreeSpec({})
>>> treespec_dict(a=treespec_leaf(), b=treespec_tuple([treespec_leaf(), treespec_
↪leaf()]))
PyTreeSpec({'a': *, 'b': (*, *)})
>>> treespec_dict({'a': treespec_leaf(), 'b': tree_structure([1, 2])})
PyTreeSpec({'a': *, 'b': [*, *]})
>>> treespec_dict({'a': treespec_leaf(), 'b': tree_structure([1, 2], none_is_
↪leaf=True)})
Traceback (most recent call last):
    ...
ValueError: Expected treespec(s) with `node_is_leaf=False`.
```

Parameters

- **mapping** (*mapping of* PyTreeSpec*, optional*) – A mapping of child treespecs. They must have the same `node_is_leaf` and `namespace` values.
- **none_is_leaf** (*bool, optional*) – Whether to treat `None` as a leaf. If `False`, `None` is a non-leaf node with arity 0. Thus `None` is contained in the treespec rather than in the leaves list and `None` will be remain in the result pytree. (default: `False`)
- **namespace** (*str, optional*) – The registry namespace used for custom pytree node types. (default: `''`, i.e., the global namespace)

Return type
PyTreeSpec

Returns
A treespec representing a dict node with the given children.

optree.**treespec_namedtuple**(*namedtuple*, *, *none_is_leaf=False*, *namespace=''*)

Make a namedtuple treespec from a namedtuple of child treespecs.

See also *tree_structure()*, *treespec_leaf()*, and *treespec_none()*.

```
>>> from collections import namedtuple
>>> Point = namedtuple('Point', ['x', 'y'])
>>> treespec_namedtuple(Point(x=treespec_leaf(), y=treespec_leaf()))
PyTreeSpec(Point(x=*, y=*))
>>> treespec_namedtuple(Point(x=treespec_leaf(), y=treespec_tuple([treespec_leaf(),␣
↪treespec_leaf()])))
PyTreeSpec(Point(x=*, y=(*, *)))
>>> treespec_namedtuple(Point(x=treespec_leaf(), y=tree_structure([1, 2])))
PyTreeSpec(Point(x=*, y=[*, *]))
>>> treespec_namedtuple(Point(x=treespec_leaf(), y=tree_structure([1, 2], none_is_
↪leaf=True)))
Traceback (most recent call last):
    ...
ValueError: Expected treespec(s) with `node_is_leaf=False`.
```

Parameters

- **namedtuple** (*namedtuple of* PyTreeSpec) – A namedtuple of child treespecs. They must have the same `node_is_leaf` and `namespace` values.
- **none_is_leaf** (*bool, optional*) – Whether to treat `None` as a leaf. If `False`, `None` is a non-leaf node with arity 0. Thus `None` is contained in the treespec rather than in the leaves list and `None` will be remain in the result pytree. (default: `False`)
- **namespace** (*str, optional*) – The registry namespace used for custom pytree node types. (default: `''`, i.e., the global namespace)

Returns
A treespec representing a dict node with the given children.

optree.**treespec_ordereddict**(*mapping=()*, *, *none_is_leaf=False*, *namespace=''*, ***kwargs*)

Make an OrderedDict treespec from an OrderedDict of child treespecs.

See also *tree_structure()*, *treespec_leaf()*, and *treespec_none()*.

```
>>> treespec_ordereddict({'a': treespec_leaf(), 'b': treespec_leaf()})
PyTreeSpec(OrderedDict({'a': *, 'b': *}))
>>> treespec_ordereddict([('b', treespec_leaf()), ('c', treespec_leaf()), ('a',␣
↪treespec_none())])
PyTreeSpec(OrderedDict({'b': *, 'c': *, 'a': None}))
>>> treespec_ordereddict()
PyTreeSpec(OrderedDict())
>>> treespec_ordereddict(a=treespec_leaf(), b=treespec_tuple([treespec_leaf(),␣
↪treespec_leaf()]))
PyTreeSpec(OrderedDict({'a': *, 'b': (*, *)}))
>>> treespec_ordereddict({'a': treespec_leaf(), 'b': tree_structure([1, 2])})
PyTreeSpec(OrderedDict({'a': *, 'b': [*, *]}))
>>> treespec_ordereddict({'a': treespec_leaf(), 'b': tree_structure([1, 2], none_is_
↪leaf=True)})
Traceback (most recent call last):
    ...
ValueError: Expected treespec(s) with `node_is_leaf=False`.
```

Parameters

- **mapping** (*mapping of PyTreeSpec, optional*) – A mapping of child treespecs. They must have the same `node_is_leaf` and `namespace` values.
- **none_is_leaf** (*bool, optional*) – Whether to treat `None` as a leaf. If `False`, `None` is a non-leaf node with arity 0. Thus `None` is contained in the treespec rather than in the leaves list and `None` will be remain in the result pytree. (default: `False`)
- **namespace** (*str, optional*) – The registry namespace used for custom pytree node types. (default: `''`, i.e., the global namespace)

Return type
PyTreeSpec

Returns
A treespec representing an OrderedDict node with the given children.

optree.**treespec_defaultdict**(*default_factory=None*, *mapping=()*, *, *none_is_leaf=False*, *namespace=''*, ***kwargs*)

Make a defaultdict treespec from a defaultdict of child treespecs.

See also *tree_structure()*, *treespec_leaf()*, and *treespec_none()*.

```
>>> treespec_defaultdict(int, {'a': treespec_leaf(), 'b': treespec_leaf()})
PyTreeSpec(defaultdict(<class 'int'>, {'a': *, 'b': *}))
>>> treespec_defaultdict(int, [('b', treespec_leaf()), ('c', treespec_leaf()), ('a',
↪ treespec_none())])
PyTreeSpec(defaultdict(<class 'int'>, {'a': None, 'b': *, 'c': *}))
>>> treespec_defaultdict()
PyTreeSpec(defaultdict(None, {}))
>>> treespec_defaultdict(int)
PyTreeSpec(defaultdict(<class 'int'>, {}))
>>> treespec_defaultdict(int, a=treespec_leaf(), b=treespec_tuple([treespec_leaf(),␣
↪treespec_leaf()]))
PyTreeSpec(defaultdict(<class 'int'>, {'a': *, 'b': (*, *)}))
>>> treespec_defaultdict(int, {'a': treespec_leaf(), 'b': tree_structure([1, 2])})
```

(continues on next page)

(continued from previous page)

```
PyTreeSpec(defaultdict(<class 'int'>, {'a': *, 'b': [*, *]}))
>>> treespec_defaultdict(int, {'a': treespec_leaf(), 'b': tree_structure([1, 2],␣
↪none_is_leaf=True)})
Traceback (most recent call last):
    ...
ValueError: Expected treespec(s) with `node_is_leaf=False`.
```

Parameters

- **default_factory** (*callable or None, optional*) – A factory function that will be used to create a missing value. (default: `None`)
- **mapping** (*mapping of PyTreeSpec, optional*) – A mapping of child treespecs. They must have the same `node_is_leaf` and `namespace` values.
- **none_is_leaf** (*bool, optional*) – Whether to treat `None` as a leaf. If `False`, `None` is a non-leaf node with arity 0. Thus `None` is contained in the treespec rather than in the leaves list and `None` will be remain in the result pytree. (default: `False`)
- **namespace** (*str, optional*) – The registry namespace used for custom pytree node types. (default: `''`, i.e., the global namespace)

Return type
PyTreeSpec

Returns
A treespec representing a defaultdict node with the given children.

optree.**treespec_deque**(*iterable=()*, *maxlen=None*, *, *none_is_leaf=False*, *namespace=''*)

Make a deque treespec from a deque of child treespecs.

See also *tree_structure()*, *treespec_leaf()*, and *treespec_none()*.

```
>>> treespec_deque([treespec_leaf(), treespec_leaf()])
PyTreeSpec(deque([*, *]))
>>> treespec_deque([treespec_leaf(), treespec_leaf(), treespec_none()], maxlen=5)
PyTreeSpec(deque([*, *, None], maxlen=5))
>>> treespec_deque()
PyTreeSpec(deque([]))
>>> treespec_deque([treespec_leaf(), treespec_tuple([treespec_leaf(), treespec_
↪leaf()])])
PyTreeSpec(deque([*, (*, *)]))
>>> treespec_deque([treespec_leaf(), tree_structure({'a': 1, 'b': 2})], maxlen=5)
PyTreeSpec(deque([*, {'a': *, 'b': *}], maxlen=5))
>>> treespec_deque([treespec_leaf(), tree_structure({'a': 1, 'b': 2}, none_is_
↪leaf=True)], maxlen=5)
Traceback (most recent call last):
    ...
ValueError: Expected treespec(s) with `node_is_leaf=False`.
```

Parameters

- **iterable** (*iterable of PyTreeSpec, optional*) – A iterable of child treespecs. They must have the same `node_is_leaf` and `namespace` values.
- **maxlen** (*int or None, optional*) – The maximum size of a deque or `None` if unbounded. (default: `None`)

- **none_is_leaf** (*bool, optional*) – Whether to treat `None` as a leaf. If `False`, `None` is a non-leaf node with arity 0. Thus `None` is contained in the treespec rather than in the leaves list and `None` will be remain in the result pytree. (default: `False`)
- **namespace** (*str, optional*) – The registry namespace used for custom pytree node types. (default: `''`, i.e., the global namespace)

Return type
PyTreeSpec

Returns
A treespec representing a deque node with the given children.

optree.**treespec_structseq**(*structseq*, *, *none_is_leaf=False*, *namespace=''*)

Make a PyStructSequence treespec from a PyStructSequence of child treespecs.

See also *tree_structure()*, *treespec_leaf()*, and *treespec_none()*.

Parameters

- **structseq** (*PyStructSequence of* `PyTreeSpec`) – A PyStructSequence of child treespecs. They must have the same `node_is_leaf` and `namespace` values.
- **none_is_leaf** (*bool, optional*) – Whether to treat `None` as a leaf. If `False`, `None` is a non-leaf node with arity 0. Thus `None` is contained in the treespec rather than in the leaves list and `None` will be remain in the result pytree. (default: `False`)
- **namespace** (*str, optional*) – The registry namespace used for custom pytree node types. (default: `''`, i.e., the global namespace)

Return type
PyTreeSpec

Returns
A treespec representing a PyStructSequence node with the given children.

optree.**treespec_from_collection**(*collection*, *, *none_is_leaf=False*, *namespace=''*)

Make a treespec from a collection of child treespecs.

See also *tree_structure()*, *treespec_leaf()*, and *treespec_none()*.

```
>>> treespec_from_collection(None)
PyTreeSpec(None)
>>> treespec_from_collection(None, none_is_leaf=True)
PyTreeSpec(*, NoneIsLeaf)
>>> treespec_from_collection(object())
PyTreeSpec(*)
>>> treespec_from_collection([treespec_leaf(), treespec_none()])
PyTreeSpec([*, None])
>>> treespec_from_collection({'a': treespec_leaf(), 'b': treespec_none()})
PyTreeSpec({'a': *, 'b': None})
>>> treespec_from_collection(deque([treespec_leaf(), tree_structure({'a': 1, 'b': 2}
↪)], maxlen=5))
PyTreeSpec(deque([*, {'a': *, 'b': *}], maxlen=5))
>>> treespec_from_collection({'a': treespec_leaf(), 'b': (treespec_leaf(), treespec_
↪none())})
Traceback (most recent call last):
    ...
ValueError: Expected a(n) dict of PyTreeSpec(s), got {'a': PyTreeSpec(*), 'b':␣
```

(continues on next page)

(continued from previous page)

```
↪(PyTreeSpec(*), PyTreeSpec(None))}.
>>> treespec_from_collection([treespec_leaf(), tree_structure({'a': 1, 'b': 2},␣
↪none_is_leaf=True)])
Traceback (most recent call last):
    ...
ValueError: Expected treespec(s) with `node_is_leaf=False`.
```

Parameters

- **collection** (*collection of PyTreeSpec*) – A collection of child treespecs. They must have the same `node_is_leaf` and `namespace` values.
- **none_is_leaf** (*bool, optional*) – Whether to treat `None` as a leaf. If `False`, `None` is a non-leaf node with arity 0. Thus `None` is contained in the treespec rather than in the leaves list and `None` will be remain in the result pytree. (default: `False`)
- **namespace** (*str, optional*) – The registry namespace used for custom pytree node types. (default: `''`, i.e., the global namespace)

Return type
PyTreeSpec

Returns
A treespec representing the same structure of the collection with the given children.

class optree.**PyTreeEntry**(*entry*, *type*, *kind*)

Bases: `object`

Base class for path entries.

entry: *Any*

type: *type*

kind: *PyTreeKind*

__post_init__()

Post-initialize the path entry.

Return type
None

__call__(*obj*)

Get the child object.

Return type
Any

__add__(*other*)

Join the path entry with another path entry or accessor.

Return type
PyTreeAccessor

__eq__(*other*)

Check if the path entries are equal.

Return type
bool

__hash__()

Get the hash of the path entry.

Return type
int

__repr__()

Get the representation of the path entry.

Return type
str

codegen(*node=''*)

Generate code for accessing the path entry.

Return type
str

__annotations__ = {'entry': 'Any', 'kind': 'PyTreeKind', 'type': 'builtins.type'}

__dataclass_fields__ = {'entry': Field(name='entry',type='Any',default=<dataclasses._MISSING_TYPE object>,default_factory=<dataclasses._MISSING_TYPE object>,init=True,repr=True,hash=None,compare=True,metadata=mappingproxy({}),kw_only=False,_field_type=_FIELD), 'kind': Field(name='kind',type='PyTreeKind',default=<dataclasses._MISSING_TYPE object>,default_factory=<dataclasses._MISSING_TYPE object>,init=True,repr=True,hash=None,compare=True,metadata=mappingproxy({}),kw_only=False,_field_type=_FIELD), 'type': Field(name='type',type='builtins.type',default=<dataclasses._MISSING_TYPE object>,default_factory=<dataclasses._MISSING_TYPE object>,init=True,repr=True,hash=None,compare=True,metadata=mappingproxy({}),kw_only=False,_field_type=_FIELD)}

__dataclass_params__ = _DataclassParams(init=True,repr=False,eq=False,order=False,unsafe_hash=False,frozen=True)

__delattr__(*name*)

Implement delattr(self, name).

__getstate__()

Helper for pickle.

__init__(*entry*, *type*, *kind*)

__match_args__ = ('entry', 'type', 'kind')

__module__ = 'optree.accessor'

__setattr__(*name*, *value*)

Implement setattr(self, name, value).

__setstate__(*state*)

__slots__ = ('entry', 'type', 'kind')

class optree.GetAttrEntry(*entry*, *type*, *kind*)

Bases: PyTreeEntry

A generic path entry class for nodes that access their children by __getattr__().

__slots__: ClassVar[tuple] = ()

property name: str

Get the attribute name.

__call__(*obj*)

Get the child object.

Return type
Any

codegen(*node=''*)

Generate code for accessing the path entry.

Return type
str

__annotations__ = {'__slots__': 'ClassVar[tuple[()]]', 'entry': 'str', 'kind': 'PyTreeKind', 'type': 'builtins.type'}

__module__ = 'optree.accessor'

class optree.GetItemEntry(*entry*, *type*, *kind*)

Bases: PyTreeEntry

A generic path entry class for nodes that access their children by __getitem__().

__slots__: ClassVar[tuple] = ()

__call__(*obj*)

Get the child object.

Return type
Any

codegen(*node=''*)

Generate code for accessing the path entry.

Return type
str

__annotations__ = {'__slots__': 'ClassVar[tuple[()]]', 'entry': 'Any', 'kind': 'PyTreeKind', 'type': 'builtins.type'}

__module__ = 'optree.accessor'

class optree.FlattenedEntry(*entry*, *type*, *kind*)

Bases: PyTreeEntry

A fallback path entry class for flattened objects.

__slots__: ClassVar[tuple] = ()

__annotations__ = {'__slots__': 'ClassVar[tuple[()]]', 'entry': 'Any', 'kind': 'PyTreeKind', 'type': 'builtins.type'}

__module__ = 'optree.accessor'

class optree.AutoEntry(*entry*, *type*, *kind*)

Bases: PyTreeEntry

A generic path entry class that determines the entry type on creation automatically.

Create a new path entry.

__slots__: ClassVar[tuple] = ()

static __new__(*cls*, *entry*, *type*, *kind*)

Create a new path entry.

Return type
PyTreeEntry

__annotations__ = {'__slots__': 'ClassVar[tuple[()]]', 'entry': 'Any', 'kind': 'PyTreeKind', 'type': 'builtins.type'}

__module__ = 'optree.accessor'

class optree.SequenceEntry(*entry*, *type*, *kind*)

Bases: GetItemEntry, Generic[T_co]

A path entry class for sequences.

__slots__: ClassVar[tuple] = ()

property index: int

Get the index.

__call__(*obj*)

Get the child object.

Return type
TypeVar(T_co, covariant=True)

__repr__()

Get the representation of the path entry.

Return type
str

__annotations__ = {'__slots__': 'ClassVar[tuple[()]]', 'entry': 'int', 'kind': 'PyTreeKind', 'type': 'builtins.type[Sequence[T_co]]'}

__module__ = 'optree.accessor'

__orig_bases__ = (<class 'optree.accessor.GetItemEntry'>, typing.Generic[+T_co])

__parameters__ = (+T_co,)

class optree.MappingEntry(*entry*, *type*, *kind*)

Bases: GetItemEntry, Generic[KT_co, VT_co]

A path entry class for mappings.

__slots__: ClassVar[tuple] = ()

property key: KT_co

Get the key.

__call__(*obj*)

Get the child object.

Return type
TypeVar(VT_co, covariant=True)

__repr__()

Get the representation of the path entry.

Return type
str

__annotations__ = {'__slots__': 'ClassVar[tuple[()]]', 'entry': 'KT_co', 'kind': 'PyTreeKind', 'type': 'builtins.type[Mapping[KT_co, VT_co]]'}

__module__ = 'optree.accessor'

__orig_bases__ = (<class 'optree.accessor.GetItemEntry'>, typing.Generic[+KT_co, +VT_co])

__parameters__ = (+KT_co, +VT_co)

class optree.NamedTupleEntry(*entry*, *type*, *kind*)

Bases: SequenceEntry[T]

A path entry class for namedtuple objects.

__slots__: ClassVar[tuple[()]] = ()

property fields: tuple[str, ...]

Get the field names.

property field: str

Get the field name.

__repr__()

Get the representation of the path entry.

Return type
str

codegen(*node=''*)

Generate code for accessing the path entry.

Return type
str

__annotations__ = {'__slots__': 'ClassVar[tuple[()]]', 'entry': 'int', 'kind': 'Literal[PyTreeKind.NAMEDTUPLE]', 'type': 'builtins.type[NamedTuple[T]]'}

__module__ = 'optree.accessor'

__orig_bases__ = (optree.accessor.SequenceEntry[~T],)

__parameters__ = (~T,)

class optree.StructSequenceEntry(*entry*, *type*, *kind*)

Bases: SequenceEntry[T]

A path entry class for PyStructSequence objects.

__slots__: ClassVar[tuple] = ()

property fields: tuple[str, ...]

Get the field names.

property field: str

Get the field name.

__repr__()

Get the representation of the path entry.

Return type
str

codegen(*node=''*)

Generate code for accessing the path entry.

Return type
str

__annotations__ = {'__slots__': 'ClassVar[tuple[()]]', 'entry': 'int', 'kind': 'Literal[PyTreeKind.STRUCTSEQUENCE]', 'type': 'builtins.type[structseq[T]]'}

__module__ = 'optree.accessor'

__orig_bases__ = (optree.accessor.SequenceEntry[~T],)

__parameters__ = (~T,)

class optree.DataclassEntry(*entry*, *type*, *kind*)

Bases: GetAttrEntry

A path entry class for dataclasses.

__slots__: ClassVar[tuple] = ()

property fields: tuple[str, ...]

Get the field names.

property field: str

Get the field name.

property name: str

Get the attribute name.

__repr__()

Get the representation of the path entry.

Return type
str

__annotations__ = {'__slots__': 'ClassVar[tuple[()]]', 'entry': 'str | int', 'kind': 'PyTreeKind', 'type': 'builtins.type'}

__module__ = 'optree.accessor'

class optree.PyTreeAccessor(*path: Iterable[PyTreeEntry] = ()*)

Bases: Tuple[PyTreeEntry, ...]

A path class for PyTrees.

Create a new accessor instance.

__slots__: ClassVar[tuple] = ()

property path: tuple[Any, ...]

Get the path of the accessor.

static __new__(*cls*, *path=()*)

Create a new accessor instance.

Return type
Self

__call__(*obj*)

Get the child object.

Return type
Any

__getitem__(*index*)

Get the child path entry or an accessor for a subpath.

Return type
PyTreeEntry | PyTreeAccessor

__add__(*other*)

Join the accessor with another path entry or accessor.

Return type
PyTreeAccessor

__mul__(*value*)

Repeat the accessor.

Return type
PyTreeAccessor

__rmul__(*value*)

Repeat the accessor.

Return type
PyTreeAccessor

__eq__(*other*)

Check if the accessors are equal.

Return type
bool

__hash__()

Get the hash of the accessor.

Return type
int

__annotations__ = {'__slots__': 'ClassVar[tuple[()]]'}

__module__ = 'optree.accessor'

__orig_bases__ = (typing.Tuple[optree.accessor.PyTreeEntry, ...],)

__parameters__ = ()

__repr__()

Get the representation of the accessor.

Return type
str

codegen(*root='*'*)

Generate code for accessing the path.

Return type
str

optree.**register_pytree_node**(*cls*, *flatten_func*, *unflatten_func*, *, *path_entry_type=<class 'optree.accessor.AutoEntry'>*, *namespace*)

Extend the set of types that are considered internal nodes in pytrees.

See also *register_pytree_node_class()* and *unregister_pytree_node()*.

The `namespace` argument is used to avoid collisions that occur when different libraries register the same Python type with different behaviors. It is recommended to add a unique prefix to the namespace to avoid conflicts with other libraries. Namespaces can also be used to specify the same class in different namespaces for different use cases.

Warning: For safety reasons, a `namespace` must be specified while registering a custom type. It is used to isolate the behavior of flattening and unflattening a pytree node type. This is to prevent accidental collisions between different libraries that may register the same type.

Parameters

- **cls** (*type*) – A Python type to treat as an internal pytree node.
- **flatten_func** (*callable*) – A function to be used during flattening, taking an instance of `cls` and returning a triple or optionally a pair, with (1) an iterable for the children to be flattened recursively, and (2) some hashable auxiliary data to be stored in the treespec and to be passed to the `unflatten_func`, and (3) (optional) an iterable for the tree path entries to the corresponding children. If the entries are not provided or given by `None`, then *range(len(children))* will be used.
- **unflatten_func** (*callable*) – A function taking two arguments: the auxiliary data that was returned by `flatten_func` and stored in the treespec, and the unflattened children. The function should return an instance of `cls`.
- **path_entry_type** (*type, optional*) – The type of the path entry to be used in the treespec. (default: `AutoEntry`)
- **namespace** (*str*) – A non-empty string that uniquely identifies the namespace of the type registry. This is used to isolate the registry from other modules that might register a different custom behavior for the same type.

Return type
Type[*CustomTreeNode*[TypeVar(T)]]

Returns
The same type as the input `cls`.

Raises

- **TypeError** – If the input type is not a class.
- **TypeError** – If the path entry class is not a subclass of `PyTreeEntry`.

- **TypeError** – If the namespace is not a string.
- **ValueError** – If the namespace is an empty string.
- **ValueError** – If the type is already registered in the registry.

Examples

```
>>> # Registry a Python type with lambda functions
>>> register_pytree_node(
...     set,
...     lambda s: (sorted(s), None, None),
...     lambda _, children: set(children),
...     namespace='set',
... )
<class 'set'>
```

```
>>> # Register a Python type into a namespace
>>> import torch
>>> register_pytree_node(
...     torch.Tensor,
...     flatten_func=lambda tensor: (
...         (tensor.cpu().detach().numpy(),),
...         {'dtype': tensor.dtype, 'device': tensor.device, 'requires_grad':␣
↪tensor.requires_grad},
...     ),
...     unflatten_func=lambda metadata, children: torch.tensor(children[0],␣
↪**metadata),
...     namespace='torch2numpy',
... )
<class 'torch.Tensor'>
```

```
>>>
>>> tree = {'weight': torch.ones(size=(1, 2)).cuda(), 'bias': torch.zeros(size=(2,
↪))}
>>> tree
{'weight': tensor([[1., 1.]], device='cuda:0'), 'bias': tensor([0., 0.])}
```

```
>>> # Flatten without specifying the namespace
>>> tree_flatten(tree)  # `torch.Tensor`s are leaf nodes
([tensor([0., 0.]), tensor([[1., 1.]], device='cuda:0')], PyTreeSpec({'bias': *,
↪'weight': *}))
```

```
>>> # Flatten with the namespace
>>> tree_flatten(tree, namespace='torch2numpy')
(
    [array([0., 0.], dtype=float32), array([[1., 1.]], dtype=float32)],
    PyTreeSpec(
        {
            'bias': CustomTreeNode(Tensor[{'dtype': torch.float32, 'device':␣
↪device(type='cpu'), 'requires_grad': False}], [*]),
            'weight': CustomTreeNode(Tensor[{'dtype': torch.float32, 'device':␣
```

(continues on next page)

(continued from previous page)

```
↪device(type='cuda', index=0), 'requires_grad': False}], [*])
        },
        namespace='torch2numpy'
    )
)
```

```
>>> # Register the same type with a different namespace for different behaviors
>>> def tensor2flatparam(tensor):
...     return [torch.nn.Parameter(tensor.reshape(-1))], tensor.shape, None
...
... def flatparam2tensor(metadata, children):
...     return children[0].reshape(metadata)
...
... register_pytree_node(
...     torch.Tensor,
...     flatten_func=tensor2flatparam,
...     unflatten_func=flatparam2tensor,
...     namespace='tensor2flatparam',
... )
<class 'torch.Tensor'>
```

```
>>> # Flatten with the new namespace
>>> tree_flatten(tree, namespace='tensor2flatparam')
(
    [
        Parameter containing: tensor([0., 0.], requires_grad=True),
        Parameter containing: tensor([1., 1.], device='cuda:0', requires_grad=True)
    ],
    PyTreeSpec(
        {
            'bias': CustomTreeNode(Tensor[torch.Size([2])], [*]),
            'weight': CustomTreeNode(Tensor[torch.Size([1, 2])], [*])
        },
        namespace='tensor2flatparam'
    )
)
```

optree.**register_pytree_node_class**(*cls=None*, ***, *path_entry_type=None*, *namespace=None*)

Extend the set of types that are considered internal nodes in pytrees.

See also *register_pytree_node()* and *unregister_pytree_node()*.

The `namespace` argument is used to avoid collisions that occur when different libraries register the same Python type with different behaviors. It is recommended to add a unique prefix to the namespace to avoid conflicts with other libraries. Namespaces can also be used to specify the same class in different namespaces for different use cases.

Warning: For safety reasons, a `namespace` must be specified while registering a custom type. It is used to isolate the behavior of flattening and unflattening a pytree node type. This is to prevent accidental collisions between different libraries that may register the same type.

Parameters

- **cls** (*type, optional*) – A Python type to treat as an internal pytree node.
- **path_entry_type** (*type, optional*) – The type of the path entry to be used in the treespec. (default: `AutoEntry`)
- **namespace** (*str, optional*) – A non-empty string that uniquely identifies the namespace of the type registry. This is used to isolate the registry from other modules that might register a different custom behavior for the same type.

Return type

Union[Type[*CustomTreeNode*[TypeVar(T)]], Callable[[Type[*CustomTreeNode*[TypeVar(T)]]], Type[*CustomTreeNode*[TypeVar(T)]]]]

Returns

The same type as the input `cls` if the argument presents. Otherwise, return a decorator function that registers the class as a pytree node.

Raises

- **TypeError** – If the path entry class is not a subclass of `PyTreeEntry`.
- **TypeError** – If the namespace is not a string.
- **ValueError** – If the namespace is an empty string.
- **ValueError** – If the type is already registered in the registry.

This function is a thin wrapper around *register_pytree_node()*, and provides a class-oriented interface:

```
@register_pytree_node_class(namespace='foo')
class Special:
    TREE_PATH_ENTRY_TYPE = GetAttrEntry

    def __init__(self, x, y):
        self.x = x
        self.y = y

    def tree_flatten(self):
        return ((self.x, self.y), None, ('x', 'y'))

    @classmethod
    def tree_unflatten(cls, metadata, children):
        return cls(*children)

@register_pytree_node_class('mylist')
class MyList(UserList):
    TREE_PATH_ENTRY_TYPE = SequenceEntry

    def tree_flatten(self):
        return self.data, None, None

    @classmethod
    def tree_unflatten(cls, metadata, children):
        return cls(*children)
```

optree.**unregister_pytree_node**(*cls*, *, *namespace*)

Remove a type from the pytree node registry.

See also *register_pytree_node()* and *register_pytree_node_class()*.

This function is the inverse operation of function *register_pytree_node()*.

Parameters

- **cls** (*type*) – A Python type to remove from the pytree node registry.
- **namespace** (*str*) – The namespace of the pytree node registry to remove the type from.

Return type
PyTreeNodeRegistryEntry

Returns
The removed registry entry.

Raises

- **TypeError** – If the input type is not a class.
- **TypeError** – If the namespace is not a string.
- **ValueError** – If the namespace is an empty string.
- **ValueError** – If the type is a built-in type that cannot be unregistered.
- **ValueError** – If the type is not found in the registry.

Examples

```
>>> # Register a Python type with lambda functions
>>> register_pytree_node(
...     set,
...     lambda s: (sorted(s), None, None),
...     lambda _, children: set(children),
...     namespace='temp',
... )
<class 'set'>
```

```
>>> # Unregister the Python type
>>> unregister_pytree_node(set, namespace='temp')
```

class optree.**PyTreeSpec**

Bases: pybind11_object

Representing the structure of the pytree.

__delattr__(*name*, */*)

Implement delattr(self, name).

__eq__(*self:* optree.PyTreeSpec, *other:* optree.PyTreeSpec) → bool

Test for equality to another object.

__ge__(*self:* optree.PyTreeSpec, *other:* optree.PyTreeSpec) → bool

Test for this treespec is a suffix of another object.

__getattribute__(*name*, */*)

Return getattr(self, name).

__getstate__(*self:* optree.PyTreeSpec) → object

__gt__(*self:* optree.PyTreeSpec, *other:* optree.PyTreeSpec) → bool
Test for this treespec is a strict suffix of another object.

__hash__(*self:* optree.PyTreeSpec) → int
Return the hash of the treespec.

__init__(**args*, ***kwargs*)

__le__(*self:* optree.PyTreeSpec, *other:* optree.PyTreeSpec) → bool
Test for this treespec is a prefix of another object.

__len__(*self:* optree.PyTreeSpec) → int
Number of leaves in the tree.

__lt__(*self:* optree.PyTreeSpec, *other:* optree.PyTreeSpec) → bool
Test for this treespec is a strict prefix of another object.

__module__ = 'optree'

__ne__(*self:* optree.PyTreeSpec, *other:* optree.PyTreeSpec) → bool
Test for inequality to another object.

__new__(***kwargs*)

__repr__(*self:* optree.PyTreeSpec) → str
Return a string representation of the treespec.

__setattr__(*name*, *value*, */*)
Implement setattr(self, name, value).

__setstate__(*self:* optree.PyTreeSpec, *state: object*) → None
Serialization support for PyTreeSpec.

__str__()
Return str(self).

accessors(*self:* optree.PyTreeSpec) → list[object]
Return a list of accessors to the leaves in the treespec.

broadcast_to_common_suffix(*self:* optree.PyTreeSpec, *other:* optree.PyTreeSpec) → *optree.PyTreeSpec*
Broadcast to the common suffix of this treespec and other treespec.

child(*self:* optree.PyTreeSpec, *index: int*) → *optree.PyTreeSpec*
Return the treespec for the child at the given index.

children(*self:* optree.PyTreeSpec) → list[*optree.PyTreeSpec*]
Return a list of treespecs for the children.

compose(*self:* optree.PyTreeSpec, *inner_treespec:* optree.PyTreeSpec) → *optree.PyTreeSpec*
Compose two treespecs. Constructs the inner treespec as a subtree at each leaf node.

entries(*self:* optree.PyTreeSpec) → list
Return a list of one-level entries to the children.

entry(*self:* optree.PyTreeSpec, *index: int*) → object
Return the entry at the given index.

flatten_up_to(*self:* optree.PyTreeSpec, *full_tree: object*) → list
Flatten the subtrees in `full_tree` up to the structure of this treespec and return a list of subtrees.

is_leaf(*self:* optree.PyTreeSpec, *strict: bool = True*) → bool

Test whether the current node is a leaf.

is_prefix(*self:* optree.PyTreeSpec, *other:* optree.PyTreeSpec, *strict: bool = False*) → bool

Test whether this treespec is a prefix of the given treespec.

is_suffix(*self:* optree.PyTreeSpec, *other:* optree.PyTreeSpec, *strict: bool = False*) → bool

Test whether this treespec is a suffix of the given treespec.

property kind

The kind of the current node.

property namespace

The registry namespace used to resolve the custom pytree node types.

property none_is_leaf

Whether to treat None as a leaf. If false, None is a non-leaf node with arity 0. Thus None is contained in the treespec rather than in the leaves list.

property num_children

Number of children in the current node. Note that a leaf is also a node but has no children.

property num_leaves

Number of leaves in the tree.

property num_nodes

Number of nodes in the tree. Note that a leaf is also a node but has no children.

paths(*self:* optree.PyTreeSpec) → list[tuple]

Return a list of paths to the leaves of the treespec.

property type

The type of the current node. Return None if the current node is a leaf.

unflatten(*self:* optree.PyTreeSpec, *leaves: Iterable*) → object

Reconstruct a pytree from the leaves.

walk(*self:* optree.PyTreeSpec, *f_node: Callable*, *f_leaf: object*, *leaves: Iterable*) → object

Walk over the pytree structure, calling `f_node(children, node_data)` at nodes, and `f_leaf(leaf)` at leaves.

optree.**PyTreeDef**

alias of *PyTreeSpec*

class optree.**PyTreeKind**(*self: optree._C.PyTreeKind*, *value: int*)

Bases: `pybind11_object`

The kind of a pytree node.

Members:

CUSTOM : A custom type.

LEAF : An opaque leaf node.

NONE : None.

TUPLE : A tuple.

LIST : A list.

DICT : A dict.

NAMEDTUPLE : A collections.namedtuple.

ORDEREDDICT : A collections.OrderedDict.

DEFAULTDICT : A collections.defaultdict.

DEQUE : A collections.deque.

STRUCTSEQUENCE : A PyStructSequence.

CUSTOM = <PyTreeKind.CUSTOM: 0>

DEFAULTDICT = <PyTreeKind.DEFAULTDICT: 8>

DEQUE = <PyTreeKind.DEQUE: 9>

DICT = <PyTreeKind.DICT: 5>

LEAF = <PyTreeKind.LEAF: 1>

LIST = <PyTreeKind.LIST: 4>

NAMEDTUPLE = <PyTreeKind.NAMEDTUPLE: 6>

NONE = <PyTreeKind.NONE: 2>

ORDEREDDICT = <PyTreeKind.ORDEREDDICT: 7>

STRUCTSEQUENCE = <PyTreeKind.STRUCTSEQUENCE: 10>

TUPLE = <PyTreeKind.TUPLE: 3>

__annotations__ = {}

__delattr__(*name*, */*)

Implement delattr(self, name).

__eq__(*self: object*, *other: object*) → bool

__ge__(*value*, */*)

Return self>=value.

__getattribute__(*name*, */*)

Return getattr(self, name).

__getstate__(*self: object*) → int

__gt__(*value*, */*)

Return self>value.

__hash__(*self: object*) → int

__index__(*self: optree._C.PyTreeKind*) → int

__init__(*self: optree._C.PyTreeKind*, *value: int*) → None

__int__(*self: optree._C.PyTreeKind*) → int

__le__(*value*, */*)

Return self<=value.

__lt__(*value*, */*)

Return self<value.

__members__ = {'CUSTOM': <PyTreeKind.CUSTOM: 0>, 'DEFAULTDICT': <PyTreeKind.DEFAULTDICT: 8>, 'DEQUE': <PyTreeKind.DEQUE: 9>, 'DICT': <PyTreeKind.DICT: 5>, 'LEAF': <PyTreeKind.LEAF: 1>, 'LIST': <PyTreeKind.LIST: 4>, 'NAMEDTUPLE': <PyTreeKind.NAMEDTUPLE: 6>, 'NONE': <PyTreeKind.NONE: 2>, 'ORDEREDDICT': <PyTreeKind.ORDEREDDICT: 7>, 'STRUCTSEQUENCE': <PyTreeKind.STRUCTSEQUENCE: 10>, 'TUPLE': <PyTreeKind.TUPLE: 3>}

__module__ = 'optree'

__ne__(*self: object*, *other: object*) → bool

__new__(***kwargs*)

__repr__(*self: object*) → str

__setattr__(*name*, *value*, */*)

Implement setattr(self, name, value).

__setstate__(*self: optree._C.PyTreeKind*, *state: int*) → None

__str__(*self: object*) → str

property name

property value

class optree.**PyTree**

Bases: Generic[T]

Generic PyTree type.

```
>>> import torch
>>> from optree.typing import PyTree
>>> TensorTree = PyTree[torch.Tensor]
>>> TensorTree
typing.Union[torch.Tensor,
             typing.Tuple[ForwardRef('PyTree[torch.Tensor]'), ...],
             typing.List[ForwardRef('PyTree[torch.Tensor]')],
             typing.Dict[typing.Any, ForwardRef('PyTree[torch.Tensor]')],
             typing.Deque[ForwardRef('PyTree[torch.Tensor]')],
             optree.typing.CustomTreeNode[ForwardRef('PyTree[torch.Tensor]')]]
```

Prohibit instantiation.

classmethod __class_getitem__(*cls*, *item*)

Instantiate a PyTree type with the given type.

Return type
TypeAlias

static __new__(*cls*)

Prohibit instantiation.

Return type
NoReturn

classmethod __init_subclass__(**args*, ***kwargs*)

Prohibit subclassing.

Return type
NoReturn

__copy__()

Immutable copy.

Return type
PyTree

__deepcopy__(*memo*)

Immutable copy.

Return type
PyTree

__annotations__ = {}

__dict__ = mappingproxy({'__module__': 'optree.typing', '__doc__': "Generic PyTree type.\n\n >>> import torch\n >>> from optree.typing import PyTree\n >>> TensorTree = PyTree[torch.Tensor]\n >>> TensorTree # doctest: +IGNORE_WHITESPACE\n typing.Union[torch.Tensor,\n typing.Tuple[ForwardRef('PyTree[torch.Tensor]'), ...],\n typing.List[ForwardRef('PyTree[torch.Tensor]')],\n typing.Dict[typing.Any, ForwardRef('PyTree[torch.Tensor]')],\n typing.Deque[ForwardRef('PyTree[torch.Tensor]')],\n optree.typing.CustomTreeNode[ForwardRef('PyTree[torch.Tensor]')]]\n ", '__class_getitem__': <classmethod(<function PyTree.__class_getitem__>)>, '__new__': <staticmethod(<function PyTree.__new__>)>, '__init_subclass__': <classmethod(<function PyTree.__init_subclass__>)>, '__copy__': <function PyTree.__copy__>, '__deepcopy__': <function PyTree.__deepcopy__>, '__orig_bases__': (typing.Generic[~T],), '__dict__': <attribute '__dict__' of 'PyTree' objects>, '__weakref__': <attribute '__weakref__' of 'PyTree' objects>, '__parameters__': (~T,), '__annotations__': {}})

__module__ = 'optree.typing'

__orig_bases__ = (typing.Generic[~T],)

__parameters__ = (~T,)

__weakref__

list of weak references to the object

class optree.**PyTreeTypeVar**(*name: str*, *param: type*)

Bases: object

Type variable for PyTree.

```
>>> import torch
>>> from optree.typing import PyTreeTypeVar
>>> TensorTree = PyTreeTypeVar('TensorTree', torch.Tensor)
>>> TensorTree
typing.Union[torch.Tensor,
             typing.Tuple[ForwardRef('TensorTree'), ...],
             typing.List[ForwardRef('TensorTree')],
```

(continues on next page)

(continued from previous page)

```
            typing.Dict[typing.Any, ForwardRef('TensorTree')],
            typing.Deque[ForwardRef('TensorTree')],
            optree.typing.CustomTreeNode[ForwardRef('TensorTree')]]
```

Instantiate a PyTree type variable with the given name and parameter.

static __new__(*cls*, *name*, *param*)

Instantiate a PyTree type variable with the given name and parameter.

Return type
TypeAlias

classmethod __init_subclass__(**args*, ***kwargs*)

Prohibit subclassing.

Return type
NoReturn

__copy__()

Immutable copy.

Return type
TypeAlias

__deepcopy__(*memo*)

Immutable copy.

Return type
TypeAlias

__dict__ = mappingproxy({'__module__': 'optree.typing', '__doc__': "Type variable for PyTree.\n\n >>> import torch\n >>> from optree.typing import PyTreeTypeVar\n >>> TensorTree = PyTreeTypeVar('TensorTree', torch.Tensor)\n >>> TensorTree # doctest: +IGNORE_WHITESPACE\n typing.Union[torch.Tensor,\n typing.Tuple[ForwardRef('TensorTree'), ...],\n typing.List[ForwardRef('TensorTree')],\n typing.Dict[typing.Any, ForwardRef('TensorTree')],\n typing.Deque[ForwardRef('TensorTree')],\n optree.typing.CustomTreeNode[ForwardRef('TensorTree')]]\n ", '__new__': <staticmethod(<function PyTreeTypeVar.__new__>)>, '__init_subclass__': <classmethod(<function PyTreeTypeVar.__init_subclass__>)>, '__copy__': <function PyTreeTypeVar.__copy__>, '__deepcopy__': <function PyTreeTypeVar.__deepcopy__>, '__dict__': <attribute '__dict__' of 'PyTreeTypeVar' objects>, '__weakref__': <attribute '__weakref__' of 'PyTreeTypeVar' objects>, '__annotations__': {}})

__module__ = 'optree.typing'

__weakref__

list of weak references to the object

class optree.**CustomTreeNode**(**args*, ***kwargs*)

Bases: Protocol[T]

The abstract base class for custom pytree nodes.

tree_flatten()

Flatten the custom pytree node into children and auxiliary data.

Return type
tuple[Iterable[TypeVar(T)], Optional[TypeVar(_MetaData, bound= Hashable)]] | tuple[Iterable[TypeVar(T)], Optional[TypeVar(_MetaData, bound= Hashable)], Optional[Iterable[Any]]]

classmethod tree_unflatten(*metadata*, *children*)

Unflatten the children and auxiliary data into the custom pytree node.

Return type
CustomTreeNode[TypeVar(T)]

__abstractmethods__ = frozenset({})

__annotations__ = {}

__dict__ = mappingproxy({'__module__': 'optree.typing', '__doc__': 'The abstract base class for custom pytree nodes.', 'tree_flatten': <function CustomTreeNode.tree_flatten>, 'tree_unflatten': <classmethod(<function CustomTreeNode.tree_unflatten>)>, '__orig_bases__': (typing_extensions.Protocol[~T],), '__dict__': <attribute '__dict__' of 'CustomTreeNode' objects>, '__weakref__': <attribute '__weakref__' of 'CustomTreeNode' objects>, '__parameters__': (~T,), '_is_protocol': True, '__subclasshook__': <classmethod(<function _proto_hook>)>, '__init__': <function _no_init>, '__abstractmethods__': frozenset(), '_abc_impl': <_abc._abc_data object>, '__annotations__': {}, '__protocol_attrs__': {'tree_unflatten', 'tree_flatten'}, '_is_runtime_protocol': True, '__non_callable_proto_members__': set()})

__init__(**args*, ***kwargs*)

__module__ = 'optree.typing'

__non_callable_proto_members__ = {}

__orig_bases__ = (typing_extensions.Protocol[~T],)

__parameters__ = (~T,)

__protocol_attrs__ = {'tree_flatten', 'tree_unflatten'}

classmethod __subclasshook__(*other*)

Abstract classes can override this to customize issubclass().

This is invoked early on by abc.ABCMeta.__subclasscheck__(). It should return True, False or NotImplemented. If it returns NotImplemented, the normal algorithm is used. Otherwise, it overrides the normal algorithm (and the outcome is cached).

__weakref__

list of weak references to the object

optree.**is_namedtuple**(*obj: object*) → bool

Return whether the object is an instance of namedtuple or a subclass of namedtuple.

optree.**is_namedtuple_class**(*cls: object*) → bool

Return whether the class is a subclass of namedtuple.

optree.**is_namedtuple_instance**(*obj: object*) → bool

Return whether the object is an instance of namedtuple.

optree.**namedtuple_fields**(*obj: object*) → tuple

Return the field names of a namedtuple.

optree.**is_structseq**(*obj: object*) → bool

Return whether the object is an instance of PyStructSequence or a class of PyStructSequence.

optree.**is_structseq_instance**(*obj: object*) → bool

Return whether the object is an instance of PyStructSequence.

optree.**is_structseq_class**(*cls: object*) → bool

Return whether the object is a class of PyStructSequence.

optree.**structseq_fields**(*obj: object*) → tuple

Return the field names of a PyStructSequence.

CHAPTER

SIX

INTEGRATION WITH THIRD-PARTY LIBRARIES

6.1 Integration for JAX

tree_ravel(tree[, is_leaf, none_is_leaf, ...])	Ravel (flatten) a pytree of arrays down to a 1D array.

optree.integration.jax.**tree_ravel**(*tree*, *is_leaf=None*, ***, *none_is_leaf=False*, *namespace=''*)

Ravel (flatten) a pytree of arrays down to a 1D array.

```
>>> tree = {
...     'layer1': {
...         'weight': jnp.arange(0, 6, dtype=jnp.float32).reshape((2, 3)),
...         'bias': jnp.arange(6, 8, dtype=jnp.float32).reshape((2,)),
...     },
...     'layer2': {
...         'weight': jnp.arange(8, 10, dtype=jnp.float32).reshape((1, 2)),
...         'bias': jnp.arange(10, 11, dtype=jnp.float32).reshape((1,))
...     },
... }
>>> tree
{
    'layer1': {
        'weight': Array([[0., 1., 2.],
                         [3., 4., 5.]], dtype=float32),
        'bias': Array([6., 7.], dtype=float32)
    },
    'layer2': {
        'weight': Array([[8., 9.]], dtype=float32),
        'bias': Array([10.], dtype=float32)
    }
}
>>> flat, unravel_func = tree_ravel(tree)
>>> flat
Array([ 6.,  7.,  0.,  1.,  2.,  3.,  4.,  5., 10.,  8.,  9.], dtype=float32)
>>> unravel_func(flat)
{
    'layer1': {
        'weight': Array([[0., 1., 2.],
                         [3., 4., 5.]], dtype=float32),
        'bias': Array([6., 7.], dtype=float32)
```

(continues on next page)

(continued from previous page)

```
    },
    'layer2': {
        'weight': Array([[8., 9.]], dtype=float32),
        'bias': Array([10.], dtype=float32)
    }
}
```

Parameters

- **tree** (*pytree*) – a pytree of arrays and scalars to ravel.
- **is_leaf** (*callable, optional*) – An optionally specified function that will be called at each flattening step. It should return a boolean, with `True` stopping the traversal and the whole subtree being treated as a leaf, and `False` indicating the flattening should traverse the current object.
- **none_is_leaf** (*bool, optional*) – Whether to treat `None` as a leaf. If `False`, `None` is a non-leaf node with arity 0. Thus `None` is contained in the treespec rather than in the leaves list and `None` will be remain in the result pytree. (default: `False`)
- **namespace** (*str, optional*) – The registry namespace used for custom pytree node types. (default: `''`, i.e., the global namespace)

Return type

tuple[Array, Callable[[Array], Union[Array, Tuple[Union[Array, Tuple[ArrayTree, ...], List[ArrayTree], Dict[Any, ArrayTree], Deque[ArrayTree], *CustomTreeNode*[ArrayTree]], ...], List[Union[Array, Tuple[ArrayTree, ...], List[ArrayTree], Dict[Any, ArrayTree], Deque[ArrayTree], *CustomTreeNode*[ArrayTree]]], Dict[Any, Union[Array, Tuple[ArrayTree, ...], List[ArrayTree], Dict[Any, ArrayTree], Deque[ArrayTree], *CustomTreeNode*[ArrayTree]]], Deque[Union[Array, Tuple[ArrayTree, ...], List[ArrayTree], Dict[Any, ArrayTree], Deque[ArrayTree], *CustomTreeNode*[ArrayTree]]], *CustomTreeNode*[Union[Array, Tuple[ArrayTree, ...], List[ArrayTree], Dict[Any, ArrayTree], Deque[ArrayTree], *CustomTreeNode*[ArrayTree]]]]]]

Returns

A pair `(array, unravel_func)` where the first element is a 1D array representing the flattened and concatenated leaf values, with `dtype` determined by promoting the `dtype`s of leaf values, and the second element is a callable for unflattening a 1D array of the same length back to a pytree of the same structure as the input `tree`. If the input pytree is empty (i.e. has no leaves) then as a convention a 1D empty array of the default dtype is returned in the first component of the output.

6.2 Integration for NumPy

tree_ravel(tree[, is_leaf, none_is_leaf, ...])	Ravel (flatten) a pytree of arrays down to a 1D array.

optree.integration.numpy.**tree_ravel**(*tree*, *is_leaf=None*, *, *none_is_leaf=False*, *namespace=''*)

Ravel (flatten) a pytree of arrays down to a 1D array.

```
>>> tree = {
...     'layer1': {
...         'weight': np.arange(0, 6, dtype=np.float32).reshape((2, 3)),
...         'bias': np.arange(6, 8, dtype=np.float32).reshape((2,)),
...     },
...     'layer2': {
...         'weight': np.arange(8, 10, dtype=np.float32).reshape((1, 2)),
...         'bias': np.arange(10, 11, dtype=np.float32).reshape((1,))
...     },
... }
>>> tree
{
    'layer1': {
        'weight': array([[0., 1., 2.],
                         [3., 4., 5.]], dtype=float32),
        'bias': array([6., 7.], dtype=float32)
    },
    'layer2': {
        'weight': array([[8., 9.]], dtype=float32),
        'bias': array([10.], dtype=float32)
    }
}
>>> flat, unravel_func = tree_ravel(tree)
>>> flat
array([ 6.,  7.,  0.,  1.,  2.,  3.,  4.,  5., 10.,  8.,  9.], dtype=float32)
>>> unravel_func(flat)
{
    'layer1': {
        'weight': array([[0., 1., 2.],
                         [3., 4., 5.]], dtype=float32),
        'bias': array([6., 7.], dtype=float32)
    },
    'layer2': {
        'weight': array([[8., 9.]], dtype=float32),
        'bias': array([10.], dtype=float32)
    }
}
```

Parameters

- **tree** (*pytree*) – a pytree of arrays and scalars to ravel.
- **is_leaf** (*callable, optional*) – An optionally specified function that will be called at each flattening step. It should return a boolean, with `True` stopping the traversal and the whole subtree being treated as a leaf, and `False` indicating the flattening should traverse the current object.

- **none_is_leaf** (*bool, optional*) – Whether to treat `None` as a leaf. If `False`, `None` is a non-leaf node with arity 0. Thus `None` is contained in the treespec rather than in the leaves list and `None` will be remain in the result pytree. (default: `False`)
- **namespace** (*str, optional*) – The registry namespace used for custom pytree node types. (default: `''`, i.e., the global namespace)

Return type

tuple[ndarray, Callable[[ndarray], Union[ndarray, Tuple[Union[ndarray, Tuple[ArrayTree, ...], List[ArrayTree], Dict[Any, ArrayTree], Deque[ArrayTree], *CustomTreeNode*[ArrayTree]], ...], List[Union[ndarray, Tuple[ArrayTree, ...], List[ArrayTree], Dict[Any, ArrayTree], Deque[ArrayTree], *CustomTreeNode*[ArrayTree]]], Dict[Any, Union[ndarray, Tuple[ArrayTree, ...], List[ArrayTree], Dict[Any, ArrayTree], Deque[ArrayTree], *CustomTreeNode*[ArrayTree]]], Deque[Union[ndarray, Tuple[ArrayTree, ...], List[ArrayTree], Dict[Any, ArrayTree], Deque[ArrayTree], *CustomTreeNode*[ArrayTree]]], *CustomTreeNode*[Union[ndarray, Tuple[ArrayTree, ...], List[ArrayTree], Dict[Any, ArrayTree], Deque[ArrayTree], *CustomTreeNode*[ArrayTree]]]]]]

Returns

A pair `(array, unravel_func)` where the first element is a 1D array representing the flattened and concatenated leaf values, with `dtype` determined by promoting the `dtype`s of leaf values, and the second element is a callable for unflattening a 1D array of the same length back to a pytree of the same structure as the input `tree`. If the input pytree is empty (i.e. has no leaves) then as a convention a 1D empty array of the default dtype is returned in the first component of the output.

6.3 Integration for PyTorch

tree_ravel(tree[, is_leaf, none_is_leaf, ...])	Ravel (flatten) a pytree of tensors down to a 1D tensor.

optree.integration.torch.**tree_ravel**(*tree*, *is_leaf=None*, ***, *none_is_leaf=False*, *namespace=''*)

Ravel (flatten) a pytree of tensors down to a 1D tensor.

```
>>> tree = {
...     'layer1': {
...         'weight': torch.arange(0, 6, dtype=torch.float64).reshape((2, 3)),
...         'bias': torch.arange(6, 8, dtype=torch.float64).reshape((2,)),
...     },
...     'layer2': {
...         'weight': torch.arange(8, 10, dtype=torch.float64).reshape((1, 2)),
...         'bias': torch.arange(10, 11, dtype=torch.float64).reshape((1,))
...     },
... }
>>> tree
{
    'layer1': {
        'weight': tensor([[0., 1., 2.],
                          [3., 4., 5.]], dtype=torch.float64),
```

(continues on next page)

(continued from previous page)

```
        'bias': tensor([6., 7.], dtype=torch.float64)
    },
    'layer2': {
        'weight': tensor([[8., 9.]], dtype=torch.float64),
        'bias': tensor([10.], dtype=torch.float64)
    }
}
>>> flat, unravel_func = tree_ravel(tree)
>>> flat
tensor([ 6.,  7.,  0.,  1.,  2.,  3.,  4.,  5., 10.,  8.,  9.], dtype=torch.float64)
>>> unravel_func(flat)
{
    'layer1': {
        'weight': tensor([[0., 1., 2.],
                          [3., 4., 5.]], dtype=torch.float64),
        'bias': tensor([6., 7.], dtype=torch.float64)
    },
    'layer2': {
        'weight': tensor([[8., 9.]], dtype=torch.float64),
        'bias': tensor([10.], dtype=torch.float64)
    }
}
```

Parameters

- **tree** (*pytree*) – a pytree of tensors to ravel.
- **is_leaf** (*callable, optional*) – An optionally specified function that will be called at each flattening step. It should return a boolean, with `True` stopping the traversal and the whole subtree being treated as a leaf, and `False` indicating the flattening should traverse the current object.
- **none_is_leaf** (*bool, optional*) – Whether to treat `None` as a leaf. If `False`, `None` is a non-leaf node with arity 0. Thus `None` is contained in the treespec rather than in the leaves list and `None` will be remain in the result pytree. (default: `False`)
- **namespace** (*str, optional*) – The registry namespace used for custom pytree node types. (default: `''`, i.e., the global namespace)

Return type

tuple[Tensor, Callable[[Tensor], Union[Tensor, Tuple[Union[Tensor, Tuple[TensorTree, ...], List[TensorTree], Dict[Any, TensorTree], Deque[TensorTree], *CustomTreeNode*[TensorTree]], ...], List[Union[Tensor, Tuple[TensorTree, ...], List[TensorTree], Dict[Any, TensorTree], Deque[TensorTree], *CustomTreeNode*[TensorTree]]], Dict[Any, Union[Tensor, Tuple[TensorTree, ...], List[TensorTree], Dict[Any, TensorTree], Deque[TensorTree], *CustomTreeNode*[TensorTree]]], Deque[Union[Tensor, Tuple[TensorTree, ...], List[TensorTree], Dict[Any, TensorTree], Deque[TensorTree], *CustomTreeNode*[TensorTree]]], *CustomTreeNode*[Union[Tensor, Tuple[TensorTree, ...], List[TensorTree], Dict[Any, TensorTree], Deque[TensorTree], *CustomTreeNode*[TensorTree]]]]]]

Returns

A pair `(tensor, unravel_func)` where the first element is a 1D tensor representing the flattened and concatenated leaf values, with `dtype` determined by promoting the `dtype`s of leaf

values, and the second element is a callable for unflattening a 1D tensor of the same length back to a pytree of the same structure as the input `tree`. If the input pytree is empty (i.e. has no leaves) then as a convention a 1D empty tensor of the default dtype is returned in the first component of the output.

CHAPTER

SEVEN

LICENSE

OpTree is released under the Apache License 2.0.

OpTree is heavily based on JAX's implementation of the PyTree utility, with deep refactoring and several improvements. The original licenses can be found at JAX's Apache License 2.0 and Tensorflow's Apache License 2.0.

INDEX

Symbols

A

B

C

D

E

U

V

W