treepathmap

Release release = "0.2a2"

David Scheliga

Mar 20, 2021

CONTENTS:

Warning: The packages development (and its documentation) is in the **alpha** state. It was segregated from another package as being a stand alone package. Therefore major changes will during its further implementation into the targeted projects.

Towards the beta (targeted release Q3/2021)

- naming of modules, classes and methods will change, since the final wording is not done.
- Code inspections are not finished.
- The documentation is broad or incomplete.
- Testing is not complete, as it is added during the first test phase. At this

CHAPTER

ONE

INTRODUCTION

of nested collections. A limited possibility of setting/replacing items within the nested collection is also supported. This package was mainly developed for tagging and grouping items within nested collections with mostly reading items and replacing values at leaf level rather complete branches. For such tasks a different package is redeveloped, from which this package origins.

With **treepathmap** items of nested collection can be

- selected by their paths using unix filename pattern or regular expressions,
- build relations by tagging items with key-value pairs,
- define a different view of the nested collection by using additional paths,
- set value of items within nested collections using selections of these,
- and direct interaction with the nested data.

CHAPTER

TWO

INSTALLATION

Installing the latest release using pip is recommended.

```
$ pip install treepathmap
```

The latest development state can be obtained from gitlab using pip.

```
$ pip install git+https://gitlab.com/david.scheliga/treepathmap.git@dev
```

2.1 Basic Usage

2.1.1 Important

Note: The path delimiters within the *treepathmap.PathMap* are defined as an arrow '->'. This is intentional as the paths should not be mistaken for system file paths.

2.1.2 Examples

A simple nested collection of Sequences and Mappings will be used for the following examples. Within this example two features of **treepathmap** will be shown. The **additional paths** which are like a different view onto the mapped nested collection and **meta attributes**, which provide the possibility to tag tree nodes for later selection purposes.

The **meta attributes** are a feature of `treepathmap.TreeNodeItems`. Child *tree node items* inherit *meta attributes* from their parents.

```
>>> from treepathmap import map_tree, wh_is
>>> sample_tree = {
...     "table": {
...         "hint": "eat now",
...         "basket": [
...             {"fruit": "apple", "color": "red"},
...             {"fruit": "apple", "color": "green"},
...             {"fruit": "banana", "color": "yellow"},
...         ],
...     },
...     "shelf": [
...         {"fruit": "apple", "color": "red"},
...         {"fruit": "banana", "color": "yellow"},
...         {"fruit": "banana", "color": "brown"},
...     ],
... }
```

Mapping of a tree

Either provide a completely defined `treepathmap.TreeNodeItems` or use the default mapping method and customize the received *tree node items* of type `treepathmap.TreeNodeItem` by an own method. In this example the items 'color' and 'hint' will be used as meta attributes of the item and an additional path will list the current items by fruit types.

Hint: Meta attributes don't need to origin from the collection.

```
>>> counters = {}
>>> from pandas import Series
>>> META_ATTRIBUTE_KEYS = ["color", "hint"]
>>> def get_meta_attributes(potential_collection):
...     if not isinstance(potential_collection, dict):
...         return None
...     return {
...         key: potential_collection[key]
...         for key in META_ATTRIBUTE_KEYS
...         if key in potential_collection
...     }
...
>>> def add_path_and_meta_attributes(a_tree_node_item):
...     global META_ATTRIBUTE_KEYS
...     # Add meta attributes if exist
...     original_item = a_tree_node_item.prime_value
...     meta_attributes = get_meta_attributes(original_item)
...     if meta_attributes is not None and meta_attributes:
...         a_tree_node_item.add_meta_attributes(meta_attributes)
```

(continues on next page)

(continued from previous page)

```
...     if not isinstance(original_item, dict):
...         return a_tree_node_item
...     # Add a different view
...     if "fruit" not in original_item:
...         return a_tree_node_item
...     fruit = original_item["fruit"]
...     global counters
...     if fruit not in counters:
...         counters[fruit] = 0
...
...     first_additional_path = 1
...     path_parts = (fruit+"s", counters[fruit])
...     a_tree_node_item.set_tree_path(first_additional_path, *path_parts)
...     counters[fruit] += 1
...     return a_tree_node_item
...
```

After the tree (nested collections) is mapped lets take a look on all tree nodes (and leaves) within a table.

Note: The direct representation of the *path map* is more detailed, than the str() representation invoked by print(), which resembles a table.

```
>>> mapped_tree = map_tree(
...     sample_tree, modify_default_path_map_item=add_path_and_meta_attributes
... )
>>> print(mapped_tree)
                          additional_path_1                 meta_attributes
->table                                                    //hint/eat now//
->table->hint                                              //hint/eat now//
->table->basket                                            //hint/eat now//
->table->basket->0              ->apples->0     //color/red//hint/eat now//
->table->basket->0->fruit                       //color/red//hint/eat now//
->table->basket->0->color                       //color/red//hint/eat now//
->table->basket->1              ->apples->1   //color/green//hint/eat now//
->table->basket->1->fruit                     //color/green//hint/eat now//
->table->basket->1->color                     //color/green//hint/eat now//
->table->basket->2             ->bananas->0  //color/yellow//hint/eat now//
->table->basket->2->fruit                    //color/yellow//hint/eat now//
->table->basket->2->color                    //color/yellow//hint/eat now//
->shelf                                                                ////
->shelf->0                      ->apples->2                  //color/red//
->shelf->0->fruit                                            //color/red//
->shelf->0->color                                            //color/red//
->shelf->1                     ->bananas->1                //color/yellow//
->shelf->1->fruit                                          //color/yellow//
->shelf->1->color                                          //color/yellow//
->shelf->2                     ->bananas->2                 //color/brown//
->shelf->2->fruit                                           //color/brown//
->shelf->2->color                                           //color/brown//
```

Different views

The added *additional path* can be used to specify a different view on the collection than it is originally structured.

```
>>> other_view_map = mapped_tree["additional_path_1"]
>>> print(other_view_map)
                   additional_path_1                 meta_attributes
->table->basket->0       ->apples->0     //color/red//hint/eat now//
->table->basket->1       ->apples->1   //color/green//hint/eat now//
->table->basket->2      ->bananas->0  //color/yellow//hint/eat now//
->shelf->0               ->apples->2                   //color/red//
->shelf->1              ->bananas->1                //color/yellow//
->shelf->2              ->bananas->2                 //color/brown//
```

Selection of items

From any map selections can be done by either searching for parts of paths using unix file pattern.

```
>>> apple_map = other_view_map.select("apples", "*")
>>> print(apple_map)
                   additional_path_1                meta_attributes
->table->basket->0       ->apples->0    //color/red//hint/eat now//
->table->basket->1       ->apples->1  //color/green//hint/eat now//
->shelf->0               ->apples->2                   //color/red//
```

```
>>> apple_map = other_view_map.select("apples", "[02]")
>>> print(apple_map)
                   additional_path_1              meta_attributes
->table->basket->0       ->apples->0  //color/red//hint/eat now//
->shelf->0               ->apples->2                //color/red//
```

The *meta* attribute of the path map leads to the selection via the *meta attributes*, which is invoked by the *where* method.

Note: The helper method *wh_is* (where is) combines both items to the correct search pattern for a where <key> is <value> statement.

```
>>> yellow_fruits = mapped_tree.meta.where(wh_is("color", "yellow"))
>>> print(yellow_fruits)
                         additional_path_1                 meta_attributes
->table->basket->2            ->bananas->0  //color/yellow//hint/eat now//
->table->basket->2->fruit                   //color/yellow//hint/eat now//
->table->basket->2->color                   //color/yellow//hint/eat now//
->shelf->1                    ->bananas->1                //color/yellow//
->shelf->1->fruit                                         //color/yellow//
->shelf->1->color                                         //color/yellow//
```

Since the prior view shows every tree node/leaf related to the *where* selection the *additional path* view can reduce the selection additionally, making it more human readable.

```
>>> yellow_fruits = mapped_tree[1].meta.where(wh_is("color", "yellow"))
>>> print(yellow_fruits)
                   additional_path_1                 meta_attributes
->table->basket->2      ->bananas->0  //color/yellow//hint/eat now//
->shelf->1              ->bananas->1                //color/yellow//
```

The *where* method used at the *path map level* requests arguments by groups of two which are *path part-value* pairs. It searches for path with the path part and selects them, if the have an equal value.

```
>>> apples = mapped_tree.where("fruit", "apple")
>>> print(apples)
                          additional_path_1                meta_attributes
->table->basket->0->fruit                       //color/red//hint/eat now//
->table->basket->1->fruit                     //color/green//hint/eat now//
->shelf->0->fruit                                          //color/red//
```

While the *where* method of *tags* (e.g. meta attributes) also allows single statements. In the current version *select* is reserved for selection of *tree node paths* in which the order of the arguments is taken into account. *where* selections doesn't need to provide any order or rather the order is ignored.

```
>>> red_apples = apples.meta.where("red")
>>> print(red_apples)
                          additional_path_1              meta_attributes
->table->basket->0->fruit                     //color/red//hint/eat now//
->shelf->0->fruit                                          //color/red//
```

2.1.3 Tagging

```
>>> fruits = mapped_tree["additional_path_1"]
>>> fruits.tags["tag_group"].tag({"foo": 1, "bar": "a"})
>>> print(fruits)
                   additional_path_1  ...         tag_group
->table->basket->0       ->apples->0  ...  //bar/a//foo/1//
->table->basket->1       ->apples->1  ...  //bar/a//foo/1//
->table->basket->2      ->bananas->0  ...  //bar/a//foo/1//
->shelf->0               ->apples->2  ...  //bar/a//foo/1//
->shelf->1              ->bananas->1  ...  //bar/a//foo/1//
->shelf->2              ->bananas->2  ...  //bar/a//foo/1//

[6 rows x 3 columns]
```

2.1.4 Limitations

```
>>> map_tree("Something not being a collection of Sequence or Mapping.")
Traceback (most recent call last):
TypeError: Expected a Sequence or Mapping, got '<class 'str'>' instead.

>>> map_tree({})
Traceback (most recent call last):
    MINIMUM_POSSIBLE_PATH_COUNT
ValueError: A path count lower than 1 is not supported.

>>> map_tree({"one": "item"})
->one
```

In the current scope **treepathmap** does not features tracking of added tree nodes to the origin collection. It's main purpose is to get selections and relations of many nested entries.

In this example a smaller tree will be used.

```
>>> smaller_sample_tree = {
...     "shelf": [
...         {"fruit": "apple", "color": "red"},
...         {"fruit": "banana", "color": "yellow"},
...         {"fruit": "banana", "color": "brown"},
...     ],
... }
>>> smaller_sample_map = map_tree(
...     smaller_sample_tree,
...     modify_default_path_map_item=add_path_and_meta_attributes
... )
>>> fruits = smaller_sample_map[1]
>>> print(fruits)
           additional_path_1   meta_attributes
->shelf->0       ->apples->3     //color/red//
->shelf->1      ->bananas->3  //color/yellow//
->shelf->2      ->bananas->4   //color/brown//
```

By using the *tree_items* attribute of *treepathmap.PathMap* you get access to the origin collections. Any changed here are reflected within the origin, but not in the PathMap.

```
>>> yellow_fruits = smaller_sample_map[1].meta.where("color/yellow")
>>> print(yellow_fruits)
           additional_path_1   meta_attributes
->shelf->1      ->bananas->3  //color/yellow//
>>> for fruit in yellow_fruits.tree_items:
...     fruit["eatable"] = True
>>> from doctestprinter import doctest_print
>>> doctest_print(smaller_sample_tree, max_line_width=70)
{'shelf': [{'fruit': 'apple', 'color': 'red'}, {'fruit': 'banana', 'color':
'yellow', 'eatable': True}, {'fruit': 'banana', 'color': 'brown'}]}
>>> print(yellow_fruits)
           additional_path_1   meta_attributes
->shelf->1      ->bananas->3  //color/yellow//
```

```
>>> fruits.tree_items[1:] = {"fruit": "banana", "color": "green", "eatable": False}
>>> fruits_reselected = fruits[1]
>>> print(fruits_reselected)
           additional_path_1   meta_attributes
->shelf->0       ->apples->3     //color/red//
->shelf->1      ->bananas->3  //color/yellow//
->shelf->2      ->bananas->4   //color/brown//
```

```
>>> doctest_print(smaller_sample_tree, max_line_width=70)
{'shelf': [{'fruit': 'apple', 'color': 'red'}, {'fruit': 'banana', 'color':
'green', 'eatable': False}, {'fruit': 'banana', 'color': 'green', 'eatable':
False}]}
```

Remapping is necessary if the origin changed severly.

```
>>> smaller_sample_map = map_tree(
...     smaller_sample_tree,
...     modify_default_path_map_item=add_path_and_meta_attributes
... )
>>> print(smaller_sample_map)
                    additional_path_1  meta_attributes
```

(continues on next page)

(continued from previous page)

```
->shelf                                          ////
->shelf->0                ->apples->4    //color/red//
->shelf->0->fruit                        //color/red//
->shelf->0->color                        //color/red//
->shelf->1               ->bananas->5  //color/green//
->shelf->1->fruit                      //color/green//
->shelf->1->color                      //color/green//
->shelf->1->eatable                    //color/green//
->shelf->2               ->bananas->6  //color/green//
->shelf->2->fruit                      //color/green//
->shelf->2->color                      //color/green//
->shelf->2->eatable                    //color/green//
```

2.2 concept of treepathmap

The basic task of *treepathmap* is to create a map of nested collections and support selection of items via the path (parts) or attached meta attributes.

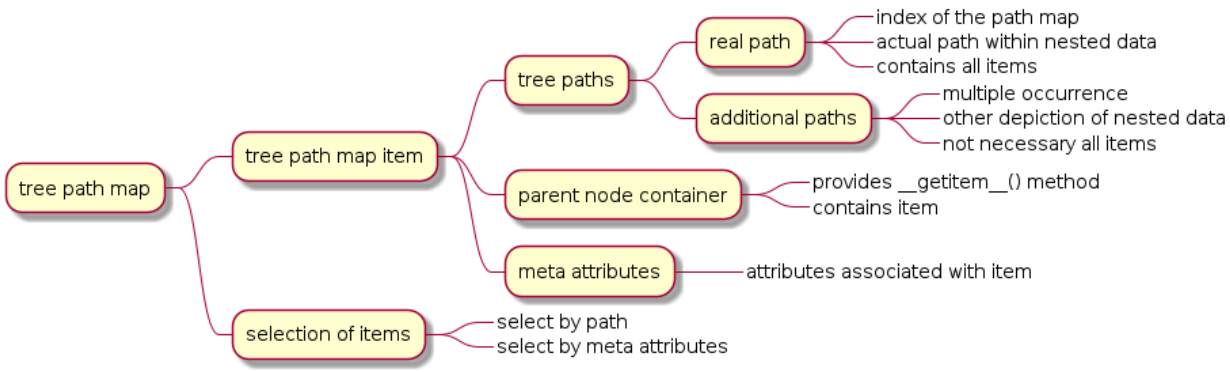
```
nested_sample_data = {
    "shelf": [
        {"banana": {"color": "red", "weight": 123}},
        {"banana": {"color": "blue", "weight": 113}},
    ],
    "table": [
        {"apple": {"color": "green", "weight": 80}},
        {"banana": {"color": "green", "weight": 113}},
        {"apple": {"color": "red", "weight": 81}},
    ]
}

sample_map = a_map_method(nested_sample_data)

# selection of items
bananas = sample_map.select("banana")

# read access to of attributes
banana_weights = bananas.select("weight")
total_weight_of_all_bananas = numpy.sum(banana_weights.to_list())
print("Total weight of bananas: {}g".format(total_weight_of_all_bananas))

# write access of attributes
bananas.tree_items["color"] = "yellow"

pretty_print(nested_sample_data)
```

```
Total weight of bananas 349g.
{
  'shelf': [
    {'banana': {'color': 'yellow', 'weight': 123}},
    {'banana': {'color': 'yellow', 'weight': 113}},
  ],
  'table': [
    {'apple': {'color': 'green', 'weight': 80}},
    {'banana': {'color': 'yellow', 'weight': 113}},
```

(continues on next page)

(continued from previous page)

```
    {'apple': {'color': 'red', 'weight': 81}},
  ]
}
```



The `PathMapItem` points at a specific item within a collection. It attributes are

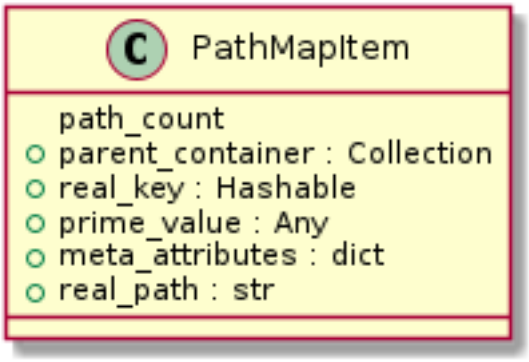
`PathMapItem.parent_container -> Collection`
The parent collection of the item, the PathMapItem points to.

`PathMapItem.real_key -> Hashable`
The index of a Sequence or hashable key of a Mapping of the item this PathMapItem points to.

`PathMapItem.prime_value : Any`
This is the value of the item

`PathMapItem.real_path : str`
The sequence of indexes and keys pointing at the item's location within the nested root collection as a path-like string.

`PathMapItem`**`.meta_attributes`**
Attributes associated with the item.



2.3 API reference

treepathmap.map_tree(potential_tree[, ...])	Maps a nested collection to a PathMap.
treepathmap.wh_is(attribute_name, value)	Makes a 'where is' search pattern.

2.3.1 map_tree

treepathmap.**map_tree**(*potential_tree: Union[Sequence, Mapping]*, *parent_path_map_item: Optional[treepathmap.selectables.TreeNodeItem] = None*, *item_is_a_leaf: Optional[Callable[[Any], bool]] = None*, *modify_default_path_map_item: Optional[Callable[[treepathmap.selectables.TreeNodeItem], treepathmap.selectables.TreeNodeItem]] = None*) → treepathmap.maps.PathMap

Maps a nested collection to a PathMap.

Parameters

- **potential_tree** – The potential nested tree to be mapped.
- **parent_path_map_item** (*Optional[TreeNodeItem]*) – The path map item of the parent container, the *potential tree* is located in. The default option *None* means the *potential tree* is the root container.
- **item_is_a_leaf** (*Optional[DetectATreeLeaf]*) – The custom Callable *item_is_a_leaf* defines if an item is a leaf or a node. By default treenodedefinition.this_item_is_a_leaf is used.
- **modify_default_path_map_item**(*Optional[Callable[[TreeNodeItem], TreeNodeItem]]*) – Defines a Callable, which enables an additional declaration of tree paths and meta potential_tree of the default real path TreeNodeItem directly after its creation.

Raises **TypeError** – if *tree_item_to_map* is not a Sequence or Mapping.

Returns PathMap

Examples

```
>>> from treepathmap.maps import map_tree
>>> sample_tree = {"1st": [["a set", "of"], ["items"]]}
>>> sample_map = map_tree(sample_tree)
>>> print(sample_map)
         meta_attributes
->1st               ////
->1st->0            ////
->1st->1            ////
>>> def add_path_and_meta_attributes(a_path_map_item):
...     a_path_map_item.add_meta_attributes({"some": "metadata"})
...     # don't give this additional path example to much credit.
...     a_nonsense_path = 0
...     the_value_of_this_item = a_path_map_item.prime_value
...     for char in str(the_value_of_this_item):
...         a_nonsense_path += ord(char)
...     # the additional path is set to location 1, first place after
...     # the real path generated by the default mapping method.
```

(continues on next page)

(continued from previous page)

```
...     a_path_map_item.set_tree_path(1, a_nonsense_path)
...     return a_path_map_item
...
>>> extended_path_map_items = map_tree_items(
...     sample_tree,
...     modify_default_path_map_item=add_path_and_meta_attributes
... )
...
>>> extended_path_map_items.print_full_items()
TreePath:
    path-0: ->1st
    path-1: ->2158
    metadata: {'some': 'metadata'}
    parent container type: dict
TreePath:
    path-0: ->1st->0
    path-1: ->1090
    metadata: {'some': 'metadata'}
    parent container type: list
TreePath:
    path-0: ->1st->1
    path-1: ->808
    metadata: {'some': 'metadata'}
    parent container type: list
```

2.3.2 wh_is

treepathmap.**wh_is**(*attribute_name: str*, *value: Union[int, str]*) → str

Makes a 'where is' search pattern. Using this method ensures the correct key-value delimiter within the search pattern.

Parameters

- **attribute_name** (*str*) – The attribute of which the value is choosen.
- **value** (*Union[int, str]*) – The value of the attribute to choose.

Returns str

Examples

```
>>> from treepathmap import wh_is
>>> wh_is("a", "b")
'a/b'
```

2.3.3 treepathmap.PathMap

treepathmap.PathMap(path_map_table, ...)	A map of a nested collection.

PathMap

class treepathmap.**PathMap**(*path_map_table: Optional[treepathmap.maps.PathMapTable] = None*, *selected_real_paths: Optional[pandas.core.indexes.base.Index] = None*, *selection_path_name: Optional[str] = None*, *path_mapping_behavior: Optional[treepathmap.maps.APathMappingBehavior] = None*)

A map of a nested collection.

Parameters

- **path_map_table** (*Optional[PathMapTable]*) – The table of all real paths and *path map items*, which is the basis of each *path map*.
- **selected_real_paths** (*Optional[pandas.Index]*) – The selected real paths (indexes) of this instance.
- **selection_path_name** (*Optional[str]*) – The name of the paths (real or additional paths) this path map points to.
- **path_mapping_behavior** (*Optional[APathMappingBehavior]*) – The mapping behavior of this path map, by which new items are mapped.

Examples

In this example the map is build from scratch instead using *treepathmap.map_tree()*, which is the recommend way.

```
>>> from treepathmap import (
...     TreeNodePaths,
...     TreeNodeItem,
...     TreeNodeItems,
...     PathMapTable
... )
```

The nested sample collection is kept simple.

```
>>> sample_tree = {"a": {"b": {"d": "leaf-1"}, "c": {"e": "leaf-2"}}}
```

The *tree node paths* are pointing at items of the collection. *Meta attributes* are inherited, what is taken into account here.

```
>>> tree_node_paths = [
...     TreeNodePaths([["a"], ["x"]], {"k1": 1}),
...     TreeNodePaths([["a", "b"], [""]], {"k1": 2, "k2": "n"}),
...     TreeNodePaths([["a", "b", "d"], ["y"]], {"k1": 2, "k2": "m"}),
...     TreeNodePaths([["a", "c"], [""]], {"k1": 3, "k2": "n"}),
...     TreeNodePaths([["a", "c", "e"], ["y"]], {"k1": 3, "k2": "m"})
... ]
```

The prior block is identical to the following one, which shows the joining method of TreeNodePaths.

```
>>> a_root_path = TreeNodePaths([RootNodePath()])
>>> path_1 = a_root_path.join([["a"], ["x"]], {"k1": 1})
>>> path_11 = path_1.join([["b"], [""]], {"k1": 2, "k2": "n"})
>>> path_111 = path_11.join([["d"], ["y"]], {"k2": "m"})
>>> path_21 = path_1.join([["c"], [""]], {"k1": 3, "k2": "n"})
>>> path_211 = path_21.join([["e"], ["y"]], {"k2": "m"})
>>> tree_node_paths = [path_1, path_11, path_111, path_21, path_211]
```

The *tree node items* resembles the core of the *PathMapTable*, which is the basis of the final *PathMap*. Using *treepathmap.map_tree()* is the recommended way to get a *PathMap*.

```
>>> sample_node_items = TreeNodeItems(
...     TreeNodeItem(tree_node_paths[0], sample_tree),
...     TreeNodeItem(tree_node_paths[1], sample_tree["a"]),
...     TreeNodeItem(tree_node_paths[2], sample_tree["a"]["b"]),
...     TreeNodeItem(tree_node_paths[3], sample_tree["a"]),
...     TreeNodeItem(tree_node_paths[4], sample_tree["a"]["c"]),
... )
...
>>> sample_table = PathMapTable(tree_node_items=sample_node_items)
>>> sample_map = PathMap(sample_table)
>>> print(sample_map)
          additional_path_1 meta_attributes
->a                     ->x        //k1/1//
->a->b                          //k1/2//k2/n//
->a->b->d               ->y  //k1/2//k2/m//
->a->c                          //k1/3//k2/n//
->a->c->e               ->y  //k1/3//k2/m//
```

The choosen path column defines the active tree nodes. The default column are the *real paths*. In this example the *additional paths* has one blank path, which is removed from the view.

```
>>> sample_map = PathMap(sample_table)
>>> sample_map.real_paths
Index(['->a', '->a->b', '->a->b->d', '->a->c', '->a->c->e'], dtype='object')
>>> other_view = sample_map[1]
>>> other_view.real_paths
Index(['->a', '->a->b->d', '->a->c->e'], dtype='object')
```

The choosen paths define the selection and iteration behavior of the *path map*. In the following case the paths '->a->b' and '->a->c' are omitted, due to a *blank* path in the *additional paths*.

```
>>> rows = {
...     real_path: row.to_list()
...     for real_path, row in other_view.iter_rows()
... }
...
```

(continues on next page)

(continued from previous page)

```
>>> from dicthandling import print_tree
>>> print_tree(rows)
->a: ['->a', '->x']
->a->b->d: ['->a->b->d', '->y']
->a->c->e: ['->a->c->e', '->y']
>>> selected_map = other_view.select("y")
>>> print(selected_map)
          additional_path_1 meta_attributes
->a->b->d               ->y  //k1/2//k2/m//
->a->c->e               ->y  //k1/3//k2/m//
```

Since the *additional paths* are active only '->a->c->e' should be selected using the where statement, although '->a->c' also is tagged with {k1: 3}.

```
>>> reduced_map = selected_map.meta.where(wh_is("k1", "3"))
>>> print(reduced_map)
          additional_path_1 meta_attributes
->a->c->e               ->y  //k1/3//k2/m//
>>> reduced_map.select("->a->b->d")
<empty map>
>>> reduced_map.real_path_exists("->not->existing")
False
>>> reduced_map.real_path_exists("->a->b")
False
>>> reduced_map.real_path_exists("->a->c->e")
True
```

```
>>> for item in reduced_map.tree_node_items:
...     print(item)
TreeNodeItem(->a->c->e: in a dict)
```

```
>>> list(reduced_map.tree_items)
['leaf-2']
```

The reduced map is switched back to the real paths. Selections from the reduced map should not exceed the current selection.

```
>>> reduced_map = reduced_map["real_path"]
>>> reduced_map.selected_indexes
Index(['->a->c->e'], dtype='object')
>>> reduced_map.select("a", "*")
->a->c->e
    additional_path_1: ->y
    meta attributes: {'k1': 3, 'k2': 'm'}
```

By default the first tag group are the meta_attributes, which are an instance of `treepathmap.IrregularTags`. The **tags** attribute of the *path map* gives access to all *tag groups*. In this example all items with the *meta attributes k2* being *m* are selected. The new tag group *'ids'* is assigned, which can be selected by this tags from the whole map afterwards.

```
>>> items_with_k2_is_m = sample_map.tags["meta_attributes"].where("k2/m")
>>> items_with_k2_is_m.tags["ids"].tag({"category": "foo", "name": "bar"})
>>> items_with_k2_is_m.tags["ids"]
          category name                          ids
->a->b->d      foo  bar  //category/foo//name/bar//
```

(continues on next page)

(continued from previous page)

```
->a->c->e      foo  bar  //category/foo//name/bar//
>>> sample_map.tags["ids"].where("category/foo")
->a->b->d
    additional_path_1: ->y
    meta attributes: {'k1': 2, 'k2': 'm'}
->a->c->e
    additional_path_1: ->y
    meta attributes: {'k1': 3, 'k2': 'm'}
```

Properties

treepathmap.PathMap.meta	
treepathmap.PathMap.tags	
treepathmap.PathMap.real_paths	
treepathmap.PathMap.selection_path_name	States the current name of the paths from which the item selection via *select* is performed.
treepathmap.PathMap.tree_node_items	
treepathmap.PathMap.tree_items	

meta

PathMap.**meta**

tags

PathMap.**tags**

real_paths

PathMap.**real_paths**

selection_path_name

PathMap.**selection_path_name**

States the current name of the paths from which the item selection via *select* is performed.

Returns str

tree_node_items

PathMap.**tree_node_items**

tree_items

PathMap.**tree_items**

Methods

treepathmap.PathMap.is_empty()
treepathmap.PathMap.from_selection([...])

is_empty

treepathmap.PathMap.**is_empty**(*self*)

from_selection

treepathmap.PathMap.**from_selection**(*self*, *selected_real_paths: Optional[pandas.core.indexes.base.Index] = None*, *selection_path_name: Optional[str] = None*)

Related to real paths (indexes)

treepathmap.PathMap.get_sub_paths_of_real_path(...)
treepathmap.PathMap.real_path_exists(real_path)
treepathmap.PathMap.get_indexes()
treepathmap.PathMap.selected_indexes

get_sub_paths_of_real_path

treepathmap.PathMap.**get_sub_paths_of_real_path**(*self*, *parent_real_path: str*) → List[str]

real_path_exists

treepathmap.PathMap.**real_path_exists**(*self*, *real_path: str*) → bool

get_indexes

treepathmap.PathMap.**get_indexes**(*self*) → pandas.core.indexes.base.Index

selected_indexes

PathMap.**selected_indexes**

Selecting

treepathmap.PathMap.select(*search_parts)	Selects tree items on base of the supplied w*search_parts*, which are parts of the *augmented_paths* within the tree.
treepathmap.PathMap.where(*where_search_parts)	**param *search_parts** Tree path parts which are parts of the requested tree items

select

treepathmap.PathMap.**select**(*self*, **search_parts*) → treepathmap.maps.PathMap

Selects tree items on base of the supplied w*search_parts*, which are parts of the *augmented_paths* within the tree. All parts are considered with an *and* condition in between them. Multiple parts within a part are considered with an *or* condition in between.

Examples

select("this", "and_that", ["this", "or_this", "or_that"])

Parameters ***search_parts** – Tree path parts which are parts of the requested tree items paths.

Returns Selection of augmented tree items.

Return type PathMapSelection

where

treepathmap.PathMap.**where**(*self*, **where_search_parts*) → treepathmap.maps.PathMap

Parameters ***search_parts** – Tree path parts which are parts of the requested tree items paths.

Returns Selection of augmented tree items.

Return type PathMapSelection

Path map items

treepathmap.PathMap.get_path_map_item_by_real_path(...)	Retrieves the tree node item of a requested real path.
treepathmap.PathMap.iter_rows()	

get_path_map_item_by_real_path

treepathmap.PathMap.**get_path_map_item_by_real_path**(*self*, *real_path: str*) → treepathmap.selectables.TreeNodeItem

Retrieves the tree node item of a requested real path.

Parameters **real_path** (*str*) – real path of the requested tree item.

Returns TreeNodeItem

iter_rows

treepathmap.PathMap.**iter_rows**(*self*) → Generator[Tuple[str, pandas.core.series.Series], None, None]

PYTHON MODULE INDEX

t

INDEX