itertree Documentation

Release 1.0.5

B.R.

Jul 01, 2023

CONTENTS

1 Changelog 3

2 Tutorial 7

3 itertree package 89

4 itertree examples 143

5 Comparison 151

6 Background information about itertree 177

7 itertree - Introduction 183

Python Module Index 191

Index 193

- *Introduction* - Short introduction to the itertree package
- *Tutorial* - A tutorial with examples and an ordered reference of the main functions of itertree
- *API Reference* - API Description of all containing classes and methods of itertree
- *Usage Examples* - itertree usage examples
- *Comparison* - Compare itertree with other packages
- *Background information* - Some background information about itertree and the target of the development

CHAPTER

ONE

CHANGELOG

1.1 Version 1.0.5

Minor bugfix (escapes).

And correct issues related wrong commit in 1.0.3

1.2 Version 1.0.3

This version contains minor changes related to comments and the test setup.

Deleting of items targeted via *slice* are improved. E.g.: *del mytree[10:100]*

We appended a new version of blist which can be used in python 3.10 and 3.11 environments.

Issues #21,#22 solved.

1.3 Version 1.0.1

Full released

After the whole functionality was implemented in the previous versions we made a review of the interfaces of the *iTree* class and we came to the decision that we should align it more with the standard interfaces in python (especially related to list and dict standard methods). Finally we updated a lot of methods to a more clearer naming and a more standardized behavior. We apologize that the changes leads into adaptions of already existing implementations of the users. But we hope that you understand after some tries that the new interface is much clearer and easier to use.

The functionalities related to the nested (in-depth) structure are now moved in an internal helper-class which is reachable via *itree.deep*.

Furthermore we saw in practice that item access is most often made via absolute index or tag,index pair (key). Therefore we changed the paradigm of targeting those kind of targets easier and with higher priority. In case of conflicts with the index or tag-index pair the user must give the lower prioritized family-tags in a specific way. The number of possible targets is increased especially a level-filter is now available too. As a side effect the limitation related to integer keys we had is no more there. Integers can now be used as tags too. In general with this release any hashable object can be used as tag.

In case the user instance an *iTree*-object without a tag or without a value. We have new default values (*NoTag* and *NoValue*) which are used automatically in this case. This is made implicit and allows the build of very simple trees without any overhead anymore. The append of values to a tree with implicit instancing the related *iTree*-object is made based on the *NoTag* definition available.

The equal check == operator is now checking for same content and no more on the identical instance (as it is in *list`s too). For identical instance checks the user must use the `is* build-in statement. But please check on the side effects of this change (read here the changed behavior of the *index()* command which is now the same like in lists (first match is delivered).

We deleted the find-functions from the object because we first thought they were too confusing and second the filter possibilities in all the methods are largely extended. We do not see any case (from old find functions) that can not be covered by the method-parameter-set we have. The filter functions are also simplified in a way that any filter-method can be used now, we do not need any more a special filter-object to be used.

Finally we uncoupled a lot of functionalities, especially the usage of the data property is changed here. *iTree* can now be used without any limitations related to the stored data. We do not expect here any more a *dict*-like-object. The provided data models can still be used if required but there is no more coupling anymore. To align with the standard *dict*-class we renamed the related attribute from *data* to *value*.

As a side effect the performance of the *iTree* could be improved again. We eliminated the different classes of *iTree* related to read_only behavior. We now use a set of methods and flags. The advantage is that the objects can now change their behavior without changing the instance of the original object (in-place-operation).

iTree objects can now be pickled (if the trees are deeper than 200 levels RecursionError will be raised (std. recursion-limit)). The serializers and rendering is updated too.

The MIT licence was extended by a “human protect patch”.

To symbolize the stability and also the final fix of the interface we decided to create the first full released version. The testsuite is largely expanded for this step.

1.4 Version 0.8.2

We reworked the itertree data module so that iData class behaves much better like a dict. All overloaded methods are improved to match the dict interface. Also *iTDataModel* is changed and is now a class that must be overloaded.

The value *validator()* raises now an *iTDataValueError* or *iTDataTypeError* exception directly. This behavior match from our point of view much better to the normal Python behavior compaired with the old style were we delivered a tuple containing the error information.

->Please consider this interface change in your code.

Second we focused for this release on the extension of functionalities related to linked iTrees:

- create internal links (reference to another tree part of the current tree)
- *localize* and *cover* of linked elements
- an example file related to the usage of links is available now

Beside this we started to extend the unit testing for the package and we fixed a lot of smaller bugs.

Because of some internal simplifications in *iTree* class the overall performance is again improved a bit.

The documentation was reviewed and improved.

No new features are planned at the moment and we just wait to complete the unit test suite, before we will do an official 1.0.0 release.

Still Beta SW -> but release candidate!

1.5 Version 0.7.3

Bugfixes in repr() and render()

Extended examples

Still Beta SW -> but release candidate!

1.6 Version 0.7.2

Improved Interval class (dynamic limits in all levels)

Adapted some tests and the documentation

Still Beta SW -> but release candidate!

1.7 Version 0.7.1

Bigger bugfix on 0.7.0 which was really not well tested!

Still Beta SW -> but release candidate!

1.8 Version 0.7.0

Recursive functions are rewritten to use an iterative approach (recursion limit exception should be avoided)

Access to the deeper structures improved (find_all, new getitem_deep() and max_depth_down() method.

New *iTree* classes for Linked, Temporary or ReadOnly items

performance improved again

Examples regarding data models added

Still Beta SW -> but release candidate!

1.9 Version 0.6.0

Improved interface and performance

Documentation is setup

Testing is improved

Examples still missing

Beta SW!

1.10 Version 0.5.0

First released version

Contains just the base functionalities of itertree. Interface is is finished by 80%

Documentation and examples are missing

testing is not finished yet.

Beta SW!

CHAPTER

TWO

TUTORIAL

In this part of the documentation we try to dive in the functions of itertree in a clear structured way. The user might look in the class description of the modules too. But the huge number of methods in the *iTree* class might be very confusing. We hope these chapters orders the things in a much better way so that the user get's used to the class as quick as possible.

To understand the functionality of itertree in practice the user might have a look on the related examples which can be found in the example folder of itertree.

Status and compatibility information:

The original implementation is done in python 3.9 and it is tested under python 3.5 and 3.9. It should work for all Python-versions >= 3.4.

From version 1.0.0 on we see the package as released and stable. The unit and integration test suite should target a huge amount of functionalities and use cases. We will try to keep the interface stable too.

2.1 Quick start - the basics

We really hope that the usage of the itertree package is intuitive. If the user is familiar with *list* and *dict* objects the basic functionality should be easy to understand. So don't have any fears about all the details described in this tutorial you can start quite quick and simple.

2.1.1 Build the object

Each tree item contains two sub-elements the value (data-object) that can be stored in the item and the subtree of children. The base class that must be instanced to build the trees is *iTree* and you can simply append sub-items.

```
>>> # Instance an iTree object by giving a tag, value and two subtree items␣
↪(children):
>>> root = iTree('root', value=0, subtree=[iTree('item0', value=0), iTree('item1',␣
↪value=1)])
>>> # append additional child with same tag!
>>> root.append(iTree('item1', value={'value1':2,'value2':3})) # any object can be␣
↪used as values
iTree('item1', value={'value1': 2, 'value2': 3})
>>> # list like operations are supported; e.g. insert():
>>> root.insert(2,iTree((1,2), value=3)) # any hashable object can be used as tag
iTree((1, 2), value=3)
>>>  # extend the tree by one more level
>>> root[1].append(iTree('sub_item0',0.1))
iTree('sub_item0', value=0.1)
```

(continues on next page)

(continued from previous page)

```
>>> root[-1].append(iTree('sub_item0',4.1))
iTree('sub_item0', value=4.1)
>>> root.render()
iTree('root', value=0)
 > iTree('item0', value=0)
 > iTree('item1', value=1)
 .   > iTree('sub_item0', value=0.1)
 > iTree((1, 2), value=3)
 > iTree('item1', value={'value1': 2, 'value2': 3})
 .   > iTree('sub_item0', value=4.1)
```

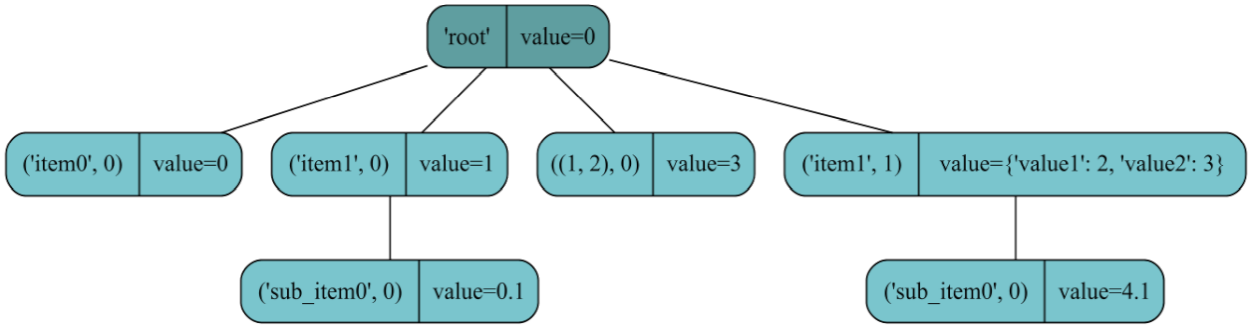


Figure representing the resulting *iTree*-object each item represented by a rounded box (left-side: tag-idx; right-side: value object)

Note: IMPORTANT: In itertree you can append items with the same tag multiple times. Those items are collected in a "tag-family". As tag you can use any hashable object.

2.1.2 Access the items

Item access is possible via *__getitem__(target)* (usage via: *my_tree[target]*). The method supports different types of targets and delivers returns related to those.

You can target a **single** item via absolute index or you can target it via tag-idx-key (this key is unique).

Note: The tag-idx-key is a tuple: *(tag, family-index)* . The family-index is the relative index of the item inside the tag-family. Inside the *iTree*-object the children are ordered and they keep the same order inside their tag-family.

In case the target is only the tag (without the tag-family-index) the method will deliver the whole tag-family as a list (multi-items-target).

```
>>> # Target a child in the tree via absolute index:
>>> root[1]
iTree('item1', value=1, subtree=[iTree('sub_item0', value=0.1)])
>>> # Target a child in the tree via tag-idx-key:
>>> root[('item1',0)]
iTree('item1', value=1, subtree=[iTree('sub_item0', value=0.1)])
>>> item=root[('item1',1)] # given index is the tag-family index in this case
>>> item.idx # delivers absolute index of the item
3
```

(continues on next page)

(continued from previous page)

```
>>> item.tag_idx # delivers tag-index-key of the item
('item1', 1)
>>> item.parent # delivers the parent object of the item
iTree('root', value=0, subtree=[iTree('item0', value=0),...,iTree('item1', value={
↪'value1': 2, 'value2': 3}, subtree=[iTree('sub_item0', value=4.1)])])
>>> # if you give just the family tag without index the whole tag-family is given as␣
↪a list
>>> root['item1']
[iTree('item1', value=1, subtree=[iTree('sub_item0', value=0.1)]), iTree('item1',␣
↪value={'value1': 2, 'value2': 3}, subtree=[iTree('sub_item0', value=4.1)])]
```

2.1.3 Iterate over the items

As the name of the package implies we have multiple iterators available.

```
>>> # Standard iterator over the children:
>>> [i.value for i in root]
[0, 1, 3, {'value1': 2, 'value2': 3}]
>>> # iteration over items (like in dicts):
>>> [i for i in root.items()]
[(('item0', 0), iTree('item0', value=0)), (('item1', 0), iTree('item1', value=1,␣
↪subtree=[iTree('sub_item0', value=0.1)])), (((1, 2), 0), iTree((1, 2), value=3)), ((
↪'item1', 1), iTree('item1', value={'value1': 2, 'value2': 3}, subtree=[iTree('sub_
↪item0', value=4.1)]))]
```

2.1.4 Copy and Compare

A copy of an *iTree*-objects implies a copy of all children. The compare operation == is an in-depth operation too (compare all children and sub-children inside (same tags, values and order?)). But a match means "just" that we have an **equal** object and not the **same** object-instance as we see:

```
>>> # Copy the iTree:
>>> new_tree=root.copy()
>>> # compare:
>>> new_tree==root
True
>>> # and see we have different objects:
>>> new_tree is root
False
>>> # and all sub-items are copied too:
>>> new_tree[0] is root[0]
False
>>> new_tree[1][0] is root[1][0]
False
```

2.1.5 In-depth operations

The itertree is a nested tree-structure and it supports in-depth operations out of the box. As we have already seen some functions in the base-class contains direct in-depth support (we saw already *copy()*, == and now follows the important function *get()*).

Additional in-depth functionalities (especially deep-iterators) can be found in the sub-class *iTree.deep*.

```
>>> # To access items in-depth target_paths can be given as parameters to get()
>>> target_item=root.get(('item1',1),0) # target types can be mixed (e.g. tag-idx and␣
↪absolute index)
>>> # Get method delivers flatten lists in case multiple items are targeted (even in␣
↪higher levels)
>>> root.get('item1',0) # delivers all matches in deepest level!
[iTree('sub_item0', value=0.1), iTree('sub_item0', value=4.1)]
>>> # other in-depth operation are found via .deep:# contains (target-item of first␣
↪get operation):
>>> target_item in root # item is not a level 1 child!
False
>>> target_item in root.deep # but item is part of the tree (in-depth)
True
>>> # size:
>>> len(root)
4
>>> len(root.deep)
6
>>> # flatten iterators over all in-depth items:
>>> [i for i in root.deep] # up-down order
[iTree('item0', value=0), iTree('item1', value=1, subtree=[iTree('sub_item0', value=0.
↪1)]), iTree('sub_item0', value=0.1), iTree((1, 2), value=3), iTree('item1', value={
↪'value1': 2, 'value2': 3}, subtree=[iTree('sub_item0', value=4.1)]), iTree('sub_
↪item0', value=4.1)]
>>> [i for i in root.deep.tag_idx_paths(up_to_low=False)] # tag_idx related iterator;␣
↪down-up order
[((('item0', 0),), iTree('item0', value=0)), ((('item1', 0), ('sub_item0', 0)), iTree(
↪'sub_item0', value=0.1)), ((('item1', 0),), iTree('item1', value=1, subtree=[iTree(
↪'sub_item0', value=0.1)])), ((((1, 2), 0),), iTree((1, 2), value=3)), ((('item1',␣
↪1), ('sub_item0', 0)), iTree('sub_item0', value=4.1)), ((('item1', 1),), iTree(
↪'item1', value={'value1': 2, 'value2': 3}, subtree=[iTree('sub_item0', value=4.
↪1)]))]
```

2.1.6 Save and load

The itertree package delivers a standard serializer which stores the *iTree*-object in a JSON formatted file. It supports the serialization of more complex value-objects (e.g. numpy-arrays).

```
>>>  # save tree to file
>>> root.dump('dt.itz',overwrite=True) # returns the sha1 hash of the tree stored in␣
↪the file
fb2a60c29acc2119363831ad1039c00836e55d15eb36955617d1c913f86dc8eb
>>>  # load tree from file
>>> loaded_tree=iTree().load('dt.itz')
>>> loaded_tree==root
True
```

Note: The *iTree*-class uses iterative and no recursive algorithms. The advantage is that the object will not raise Re-

cursionErrors even if user defines very deep trees (e.g. see the performance-analysis with a tree depth of 1000 levels). To keep the functionality for the stored data the serializer creates a flat list of entries (which avoids RecursionErrors related to the JSON parser).

2.1.7 Next steps

After those basic functions are learned you may be motivated to dive deeper. E.g. learn more about possible targets related to item access, linking trees and branches, search/filter in the trees and store more advanced datatypes in the tree.

In the tutorial you can find a large table which compares *iTree* with *dict* and *list* objects (link can be found in next chapter).

2.2 Introduction to the iTree object

As a starting point the *iTree*-class should be seen as a *list* (the object inherits his functions from a *list* or *blist*). All typical *list* like methods are available. But *iTree*-objects supports also in-depth access and iterations over different levels of the nested tree structure. Different than in normal lists the *iTree*-class supports the more *dict*-like access functions related to keys too.

For a functional comparison in between *iTree*, *list* and *dict* the table in the chapter *Comparison of the iTree object with lists and dicts* might be interesting for the reader.

2.2.1 Same tags and tag-families

The children items of a *iTree*-object with the same tag are collected in a related tag-family. Inside the family each item contains a related index (relative index). The items can be targeted by giving the family-tag and the **family-index** as a tuple. This *tag_idx*-pair is a unique *key* inside the children of a parent. Each item in a nested *iTree*-structure contains a unique *tag_idx_path* from the root object (or any parent (relative path)). The *tag_idx_path* property of an item contains all *tag_idx*'s from the root item over all parents to the item itself (the *tag_idx_path* is represented by a *tuple* of *tag_idx* items).

Beside this more key-like targeting we can target an item via the **absolute** index too (*idx* or *idx_path*). The access is made here like it is known from lists. The *idx_path* is again represented as a tuple of index numbers.

It's important that the user understands the difference between the **absolute** index and the **family-index**.

The things might getting clearer if we look into the order structure of an *iTree*-object:

The tree items of one level are ordered globally like in a *list* and the same order of items will be found in the tag related family too. The order is not independent because an item which is a predecessor of another item in the tag-family will be found before the item in the global order too. But from the global/absolute view there might be other items (with other tags) inbetween. They are not seen in the family because they have other tags!

abs-order	family “a”	family “b”
iTree(tag=’a’,value=1)	iTree(tag=’a’,value=1)	
iTree(tag=’b’,value=2)		iTree(tag=’b’,value=2)
iTree(tag=’a’,value=3)	iTree(tag=’a’,value=3)	
iTree(tag=’b’,value=4)		iTree(tag=’b’,value=4)

Normally the tag must be given to the item when it is instanced. As tag-objects the user can give any hashable object (e.g. tuples, int, float, str, bytes). If no tag is given the *iTree*-object will use the default *NoTag*-object as tag. In *iTree* exists a *rename()* method to change the tag of an item, but if possible this should be avoided because it implies a reordering of the items inside the effected tag-families (removed tag and new tag).

2.2.2 Unique parent principle

We have one important limitation related to *iTree* objects, each one can only be the child of **ONE PARENT ONLY!**

If the users tries to append an *iTree*-object that is already a child of an *iTree* to another *iTree* a *RecursionError* will be raised.

Only if the *iTree* referencing feature *iLink()* is utilized the share of same objects in different tree-sections is possible.

To avoid issues in some multi-item-functions implicit copies are created automatically (e.g.: *my_tree.extend(itree)* or rearrangements via *itree[1],itree[2]==itree[2],itree[1]* or multiplications like *my_tree= itree * 10*).

Note: The terms *itree* and *my_tree* are used as examples of instanced objects in this tutorial.

In case of implicit copies the objects *copy()*-method will be used. The method is an in-depth copy of all sub-items (required because of one parent only principle) and the method creates also a copy of the stored value object (top-level-only). It is an iterative equivalent to the operation:

```
new_itree=iTree(itree.tag,copy.copy(itree.value), subtree=[i.copy() for i in itree])
```

Warning: If it is required to keep the original objects the operations:

- multiplication of *iTree*-objects
- build *iTree*-object based of children of another *iTree* (e.g. *new_tree=iTree(subtree=old_tree)*)
- rearrangements like *itree[1],itree[2]==itree[2],itree[1]*

must be avoided!

2.2.3 Naming conventions

In the itertree package and this tutorial the following naming convention is used:

- **item** An item is an *iTree* object that is a child (sub-element) of an *iTree* parent object somewhere inside the nested tree structure.
- **parent** The current object can be the child of a specific parent or it has no parent. A child can have only one parent. All parent related properties will deliver *None* in case no parent is coupled to the object (e.g. *itree.idx*, *itree.key*,`itree.parent`, . . .).
- **child** An *iTree* object that has a parent. This object is part of the parents children and it is related to the absolute order of them and to its family siblings.
- **root** For nested children in sub-sub-trees the root is the top level parent. Any *iTree* object that has no parent is a root object itself.
- **family** The group (*list*) of children in an *iTree* that have the same tag (The children have same order in the family as in *iTree*-object (absolute order)).
- **tag** The tag is a object that defines that the item is part of a specific family. If no tag is given automatically the *NoTag* object will be used as tag. The user can use any hashable object as a tag for an *iTree*-object.
- **idx** Specific (unique) index of a children related to the absolute order of the *iTree*'s children (list like access)
- **tag-idx** Specific (unique) tuple of family-tag and family-index of an ´iTree´ child (sometimes named tag-idx-key).
- **idx_path** Specific (unique) tuple of indexes (index per level) describe the path from the root parent object to the specific nested child somewhere deep in the *iTree* object. E.g *(0,1,0)* targets:
  - 0. element (level 0) ->
  - 1. element (level 1) ->
  - 0. element (level 2)

  In access function the relative idx_path from the current object to the sub-item must be given (not the absolute path (might be used if you target via *itree.root.get(*idx_path)*)).
- **tag_idx_path** List of tag-idx-keys (unique tuples of family-tag,family-index) describe the path from the root object to the specific nested child somewhere deep in the *iTree* object. E.g *(('tag1',0),(NoTag,1),(1.6,0))* targets:
  - 0. element in tag-family 'tag1' (level 0) ->
  - 1. element in tag-family NoTag (level 1) ->
  - 0. element in tag-family 1.6 (level 2)

  In access function the relative tag_idx_path from the current object to the sub-item must be given (not the absolute path (might be used if you target via *itree.root.get(*tag_idx_path)*)).
- **target** Is an object that targets one or multiple items in an *iTree* the target is used related to one level only. But to reach deeper levels the user can create based on targets target_paths (list of targets).

  The common access methods *__getitem__()* , *get()* are sensitive related to the given target and a related object will be returned:
  - Single target definitions deliver a single item.
  - Multi target definitions deliver a *list* (or *blist*) of items.

  Possible target definitions are:
  - index - absolute target index integer (fastest operation) -> unique/single result

  - key - key tuple (family_tag, family_index) -> unique/single result
  - tag-set - {family_tag} object targeting a whole family -> list result
  - tag-sets - {family_tag,family-tag2,...} object targeting multiple families -> list result
  - target-list - indexes or keys or other targets (mixed lists support). Selects items in **same level** based given target-list -> list result
  - index slice - slice of absolute indexes -> list result
  - key slice - tuple of of (family_tag, family_index_slice) -> list result
  - filter_method - a filtering method that delivers True/False related to an analysis of item properties -> list result
  - iter_method - if build-in *iter* is given a list of all children will be delivered (same like *list(itree.__iter__())*
  - Ellipsis - if Ellipsis ... is given a list of all children will be delivered (same like *itree[:]*)

- **target-path** The target-path is a list of targets and it is used for in-depth operations over the different nested levels of the tree. Most often (e.g. *get(*target_path)*) the target-path is given as a pointer argument to the method.

Note: Please understand the difference in between a target-list and a target_path.

  - target-list -> targets items in the **same level** (siblings)
  - target-paths -> targets items in **different nested levels**, this is an in-depth access

In the related methods (e.g. *get()*) target-list are given as one parameter but target_paths are given as multiple parameters.

  - *itree.get([1,2,3])~[itree[1],itree[2],itree[3]]* -> targets the children [1][2][3] in level 1
  - *itree.get(*[1,2,3])~itree[1][2][3]* -> targets the item [1] in level 1, [2] in level 2 and [3] in level 3

If the user defines a target-path like *my_path=[[1,2],[0,1]]* the object will be seen as a target_path of target_list-targets. E.g. such a list can be used in *my_tree.get(*my_path))* (give pointer). The input is the same like *get([1,2,3,4],[9,10])*. The result of the request is a flatten iterator over all matches in the deepest requested level but it will considering all multi-matches in the levels inbetween too.

```
>>> root = iTree('root')
>>> root.append(iTree('a', value={'mykey': 1}, subtree=[iTree('a1'),␣
↪iTree('a2')]))
iTree('a', value={'mykey': 1}, subtree=[iTree('a1'),iTree('a2')])
>>> root.append(iTree('a', value={'mykey': 1}, subtree=[iTree('a1'),␣
↪iTree('a2')]))
iTree('a', value={'mykey': 1}, subtree=[iTree('a1'),iTree('a2')])
>>> root.get([0, 1], [0, 1])
[iTree('a1'), iTree('a2'), iTree('a1'), iTree('a2')]
```

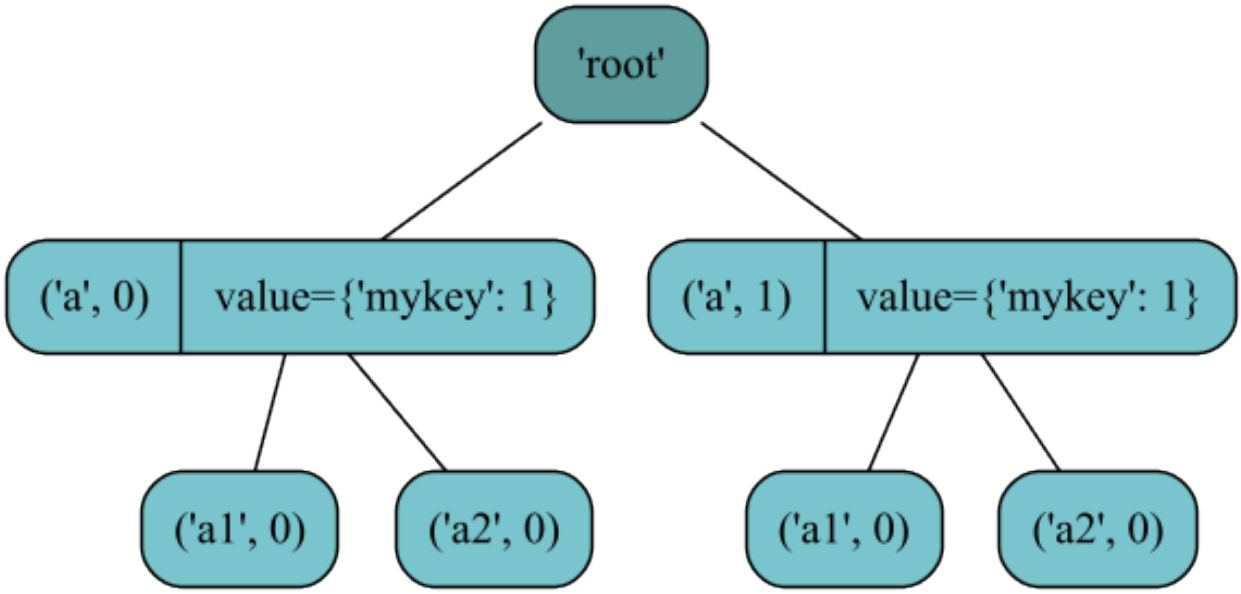


Fig. 1: Figure showing the resulting iTree

- **value** The value is the a data-object that can be stored in a *iTree*-object

Name extensions:

- **s** If plural is used in method names this is a hint that the method return will be an iterator: e.g.: *itree.keys()*; *itree.values()*; *itree.items()*; *itree.deep.tag_idx_paths()*; *itree.deep.idx_paths()*
- **_path** The extension is used for parameters and properties. This means that the parameter is an iterable that targets the different levels of the nested structure (in-depth access). e.g. *get(*target_path)*
- **filter_method** A method that check the match of a *iTree*-item related to a property and the method delivers True/False if an *iTree*-item is given as parameter. Therefore the method can be used for the filtering of items.

Internal helper classes:

- **.deep** Helper class contains the in-depth functions that targets all elements inside the *iTree*-object. E.g. the class contains different flatten iterators that iterates over all nested items of the *iTree*-object. The class contains no *__getitem__()* method for in-depth item access because the function is already covered by the standard *get()* and *get_single()* methods. The available *get()*-method is the same as the *get()*-method in the base class. (in detail: *iTree full overview over the in-depth functionalities*)
- **.getitem** Helper class that contains a lot of specific getitem methods f<or the different possible targets. (in detail: *Item Access*)

2.3 Construction of an itertree

The first step in the construction of a itertree is to instance the main itertree class: *iTree*.

class itertree.**iTree**(*tag=<class 'itertree.itree_helpers.NoTag'>*, *value=<class 'itertree.itree_helpers.NoValue'>*, *subtree=None*, *link=None*, *flags=0*)

Instance the *iTree* object:

```
>>> item1 = iTree('item1')  # itertree item with the tag 'item1'
>>> item2 = iTree('item2', 2)  # instance a iTree-object with value content integer 2
>>> item2b = iTree('item2', {'mykey': 2})  # instance a iTree-object with a dict as␣
↪value content
>>> item3 = iTree()  # instance an iTree-object with the default tag (==NoTag) and no␣
↪data content (==NoValue)
>>> root = iTree('root', subtree=[item1, item2, item2b, item3])
>>> root.render()
iTree('root')
 > iTree('item1')
 > iTree('item2', value=2)
 > iTree('item2', value={'mykey': 2})
 > iTree()
```

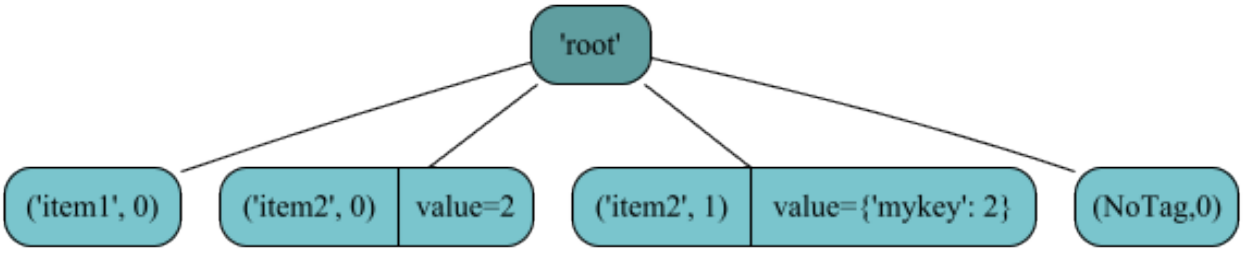


Fig. 2: Figure showing the resulting iTree

To include *iTree*-objects as a children in a parent object we have several possibilities, those functionalities are comparable to the same methods you find in *list*-objects.

```
>>> root = iTree('root')
>>> root.append(iTree('child'))  # append a child
iTree('child')
>>> # The append operation delivers the appended object back
>>> root += iTree('child')  # alternative way to append a child
>>> root.append('value_content')  # append a child with implicit iTree(tag=NoTag,
↪value='value_content')
iTree(value='value_content')
>>> root.insert(1, iTree('child','inserted'))  # insert the item in the given target␣
↪position (the insert is done in this target (index)
iTree('child', value='inserted')
>>> # the old item with given target (index) will be moved in next position
>>> root.render()
iTree('root')
 > iTree('child')
 > iTree('child', value='inserted')
 > iTree('child')
 > iTree(value='value_content')
>>> root[0] = iTree('newchild')  # replace the child with index 0
>>> root.render()
```

(continues on next page)

(continued from previous page)

```
iTree('root')
 > iTree('newchild')
 > iTree('child', value='inserted')
 > iTree('child')
 > iTree(value='value_content')
>>> del root[('newchild', 0)]  # deletes the child with key=('newchild',0) family-tag=
↪'newchild' and family-index=0
>>> root.render()
iTree('root')
 > iTree('child', value='inserted')
 > iTree('child')
 > iTree(value='value_content')
>>> del root[1]  # deletes the child with absolute index 1
>>> root.render()
iTree('root')
 > iTree('child', value='inserted')
 > iTree(value='value_content')
>>> # The tag can be any hashable type!
>>> root.append(iTree(1))  # append a child with tag 1
iTree(1)
>>> root.append(iTree((1, 2, 3)))  # append a child with tag (1,2,3)
iTree((1, 2, 3))
>>> root.append(iTree((1, 2, 3), 1))  # append a child with tag (1,2,3) and data␣
↪content 1
iTree((1, 2, 3), value=1)
>>> root.render()
iTree('root')
 > iTree('child', value='inserted')
 > iTree(value='value_content')
 > iTree(1)
 > iTree((1, 2, 3))
 > iTree((1, 2, 3), value=1)
>>> new_itree = iTree()
>>> root.append(new_itree)
iTree()
>>> root.append(new_itree)  # appending same object again will not work because␣
↪parent is already set
Traceback (most recent call last):
...
RecursionError: Given item has already a parent iTree!
```

Remember if a tag is appended in an object where already exists a child with same tag this/those child/children will not be overwritten! Furthermore all items with same tags are collected in the same tag-family:

```
>>> family=root[{(1,2,3)}] # target the family with a set(): {(1,2,3)}
>>> family # is represented as a list of the related items (with same tag)
[iTree((1, 2, 3)), iTree((1, 2, 3), value=1)]
>>> family=root.get.by_tag((1,2,3)) # target via the s?ecial tag access function
>>> family # is represented as a list of the related items (with same tag)
[iTree((1, 2, 3)), iTree((1, 2, 3), value=1)]
```

Additionally a huge set of methods is available for structural manipulations related to the children of a item.

itertree.iTree.**append**()

Append the given *iTree*-object to the *iTree* (new last child) The *append()* method is the fastest way to add a single item to the end of the tree.

Except In case *iTree*-object has already a parent a *RecursionError* will be raised Other exceptions

might come up in case the *iTree* is protected (tree read-only mode).

Parameters **item** (*Union[iTree,object]*) – *iTree*-object to be appended

> **Warning:** In case the given item-object is not a *iTree*-object the item is interpreted as a value and the *iTree* will be created implicit (with tag-family *NoTag*) in the way:
>
> *iTree(tag=NoTag, value=item)* ~ ìTree(value=item) If no item is given an empty iTree is created tag=`NoTag`; value=`NoValue`.

```
>>> root=iTree('root')
>>> root.append('myvalue')
iTree(value='myvalue')
>>> root.append() # append an empty iTree-object
iTree()
```

Return type *iTree*

Returns Delivers the appended item itself (it might be useful for the user to get the updated information of the object).

itertree.iTree.**__iadd__**()

append the given item to the iTree (short form of *append()*)

Except In case *iTree*-object has already a parent a *RecursionError* will be raised Other exceptions might come up in case the *iTree* is protected (tree read-only mode).

Parameters **other** (*Union[iTree,object]*) – *iTree*-object to be appended.

> **Warning:** As in *append()* in case the given item-object is not a *iTree*-object the item is interpreted as a value and the *iTree* will be created implicit (with *NoTag* tag).

Return type *ìTree*

Returns self

itertree.iTree.**appendleft**()

Append the given *iTree*-object to the left of the parent-tree (new first child) The *appendleft()* method is the recommended method to add a new first item to iTree (quicker than *insert(0,item)*). Compared to *append()* the method is slower and the cache index information gets invalid after the operation (will be automatically updated later on if required).

Except In case *iTree*-object has already a parent a *RecursionError* will be raised. Other exceptions might come up in case the *iTree* is protected (tree read-only mode).

Parameters **item** (*Union[iTree,object]*) – *iTree*-object to be appended as first item.

> **Warning:** As in *append()* in case the given item-object is not a *iTree*-object the item is interpreted as a value and the *iTree* will be created implicit.

Return type *iTree*

Returns Delivers the appended item itself (it might be useful for the user to get the updated information of the object).

itertree.iTree.**extend**()

We extend the *iTree* with given items (multi append). The function is high performant and if you have to append a large number of items it is recommended to create an iterator of the items and feed them into this method. This is quicker compared to a loop doing multiple normal *append()* operations.

Note: In case the to be extended items have already a parent an implicit copy will be made. We do this because the internal copy can be created more effective. We accept also iTree-objects as extend_items parameter and the children which have a parent will be automatically copied to be integrated in this second tree. We have the same situation with a filtered iterator which might be used to extend this *iTree* too.

Parameters **items** (*Iterable*) – iterable-object that contains *iTree*-objects as items it can be:

- iterator or generator of *iTree*-objects (using next)
- *iTree*-object (children will be copied and extended in this tree)
- iterable of *iTree*-objects (list, tuple, ...)
- argument list for *iTree*-instance (´__init__()´) (created by ´get_init_args()´ or ´get_init_args_deep()´) -> this is most often an internal functionality.
- iterator or generator of value-objects (using next) - implicit *iTree*-objects created
- iterable of value-objects (list, tuple, ...)- implicit *iTree*-objects created

itertree.iTree.**extendleft**()

Multy item append on left hand-side (at the beginning) of the ´iTree´.

The operation is slower than ´extend()´ because it requires a reordering of all items in the *iTree*.

Note: The order of extended items is kept in the operation. It's comparable with: ´[1,2,3]+[4,5,6]=[1,2,3,4,5,6]´ but the result is not a new instance, self is kept.

Note: In case the to be extended items have already a parent an implicit copy will be made. We do this because the internal copy can be created more effective. We accept also iTree-objects as extend_items parameter and the children which have a parent will be automatically copied to be integrated in this second tree. We have the same situation with a filtered iterator which might be used to extend this *iTree* too.

Parameters **items** (*Iterable*) – iterable-object that contains *iTree*-objects as items it can be:

- iterator or generator of *iTree*-objects (using next)
- *iTree*-object (children will be copied and extended in this tree
- iterable of *iTree*-objects (list, tuple, ...)
- argument list for *iTree*-instance (´__init__()´) (created by ´get_init_args()´ or ´get_init_args_deep()´)
- iterator or generator of value-objects (using next) - implicit *iTree*-objects created
- iterable of value-objects (list, tuple, ...)- implicit *iTree*-objects created

itertree.iTree.**insert**()

Insert an item **before** a given target-position. The insertion works like in lists.

The insertion operation is slower as the append operations.

If *target=None* is given the operation inserts in the last position (== *append()*).

Except In case *iTree*-object has already a parent a *RecursionError* will be raised Other exceptions might come up in case the *iTree* is protected (tree read-only mode).

Parameters

- **target** (*Union[Integer,tuple,iTree,None]*) – target position definition; **target must target a single/unique item!** Possible targets:
  - index - absolute target index integer, negative values supported too (count from the end).
  - key - key-tuple (family_tag, family_index) pair
  - item - *iTree*-item that is already a children (future successor)
  - None - if *None* is given we will append the item in the last position of the ´iTree´-object
- **item** (*Union[iTree,object]*) – *iTree*-object to be inserted in the tree.

> **Warning:** As in *append()* in case the given item-object is not a *iTree*-object the item is interpreted as a value and the *iTree* will be created implicit.

Return type *iTree*

Returns Delivers the inserted item itself (it might be useful for the user to get the updated information of the object).

itertree.iTree.**move**()

Move this item in given target position (item will be positioned **before** the given target). The given target must be a unique item! If None is given the item will be moved in the last position of the *iTree*. If an ìTree`-object is given as target it must be a children of the same parent (sibling).

Except LookupError in case the target is not found or not unique!

Parameters **target** (*Union[Integer,tuple,iTree,None]*) – target-object defining the replacement target; possible types are:

- index - absolute target index integer, negative values supported too (count from the end).
- key - key-tuple (family_tag, family_index) pair
- item - *iTree*-item that is already a children (future successor)
- None - if *None* is given we will move the item to the last position in the ´iTree´-object

Returns self (with updated indexes)

itertree.iTree.**rename**()

give the item a new family tag

The renaming of the item implies a reordering of the items in the tree because the family order depends on the global/absolute order of items.

Parameters **new_tag** (*Hashable*) – new tag (any kind of hashable object)

Return type *iTree*

Returns Delivers the renamed item itself (it might be useful for the user to get the updated information of the object).

itertree.iTree.**pop**()

pop the item out of the tree, if no key is given the last item will be popped out

We do not have the method popleft because *pop(0)* does the same.

Parameters **target** (*Union[int,tuple,Hashable,Iterable,slice,iTree]*) – target of popped item(s):

- *index* - absolute target index integer (fastest operation)
- *key* - key tuple (family_tag, family_index)
- *tag* - Tag(family_tag) object targeting a whole family
- *target-list* - absolute indexes or keys to be replaced (indexes and keys can be mixed)
- *index-slice* - slice of absolute indexes
- *key-slice* - tuple of (family_tag, family_index_slice)
- *itree_filter* - method (callable) for filtering the children of the object

Returns popped out item(s) (parent will be set to None). In case multiple items are removed an iterator over the removed items is given.

2.4 iTree other structure related commands

itertree.iTree.**__setitem__**()

Replace an item with the given new item given in the *value*-parameter. The method handles also multiple replaces (rearrangements) like:

```
>>> mytree[1],mytree[0]=mytree[0],mytree[1]
```

Warning: Because of the parent only principle in rearrangements operations an implicit copy might be created.

Note: Linked items cannot be changed. If changes are required The user must change the link source tree items and afterwards actively rerun *load_links()* to reload the linked tree.

Except In case the target is not found or the *iTree* is protected (read-only tree).

Parameters

- **target** – target object defining the replacement target; possible types are:
  - index - absolute target index integer (fastest operation)
  - key - key tuple (family_tag, family_index)
  - tag - Tag(family_tag) object targeting a whole family
  - target-list - absolute indexes or keys to be replaced (indexes and keys can be mixed)
  - index slice - slice of absolute indexes
  - key slice - tuple: (family_tag, family_index_slice)

  For multi targets the given value must have a matching structure (item list with same length).

  We have two special targets which are used for placing/replacing single items in the iTree:

  - Ellipsis ... - new_items tag-family will be deleted and the new-item is placed in families first item position

  - items_tag - new_items tag-family will be delted and the new-item is placed in families last item position

  If those two special targets are used and the new-items family does not exist yet, the method will just append the new item, no exception will be raised.

- **value** – iTree object that should replace the target or in case of multi targets a tuple of items that should be used for replacements

Returns value added items (only for internal usage)

itertree.iTree.**__delitem__**()
The function deletes the targeted item in the tree.

Except In case the target is not found or the *iTree* is protected (read-only tree).

Parameters **target** (*Union[int,tuple,Hashable,Iterable,slice]*) – target object defining the replacement target; possible types are:

- *index* - absolute target index integer (fastest operation)
- *key* - key tuple (family_tag, family_index)
- *tag* - Tag(family_tag) object targeting a whole family
- *target-list* - absolute indexes or keys to be replaced (indexes and keys can be mixed)
- *index-slice* - slice of absolute indexes
- *key-slice* - tuple of (family_tag, family_index_slice)
- *itree_filter* - method (callable) for filtering the children of the object

Returns deleted item

itertree.iTree.**clear**()
deletes all children and the value!

All flags stay unchanged, except the load_links flag!

Parameters

- **keep_value** (*bool*) –
  - True - value is not deleted
  - False - value will be replaced with NoValue
- **local_only** (*bool*) –
  - True - clear only the local items
  - False - clear whole object (The object is reset to the no links loaded state and locals are deleted)

itertree.iTree.**copy**()
create a copy of this item

The difference in between *copy()* and *deepcopy()* for *iTree* is just that we do in *deepcopy()* a deepcopy of all value items. In *copy()* we just copy the value object not the items inside, the pointers to the original objects are kept (for immutable objects there is no difference).

Returns copied iTree object

itertree.iTree.**copy_keep_value**()
Create a copy of this item.

The difference in between normal *copy()* and this method is that the value objects are completely untouched in this operation (for immutable objects there is no difference in between the two copy operations).

Returns copied iTree object

itertree.iTree.**deepcopy**()
create a deepcopy of this item

The difference in between *copy()* and *deepcopy()* for *iTree* is just that we do in *deepcopy()* a deepcopy of all value items. In *copy()* we just copy the value object not the items inside, the pointers to the original objects are kept (for immutable objects there is no difference).

Returns deep copied new iTree object

The copy operations are automatically in-depth operations this means the items in the subtree will be copied too. This is required because of the one parent only principle. The available copy operations making a difference in the treatment of the *itree.value*-object:

- *copy()* - creates a top-level copy of the value object
- *copy_keep_values()* - copies just the *iTree* object but keep the value
- *deepcopy()* - creates a deepcopy of the value object

The methods of the *copy* package use the same functionalities *copy.copy(itree) ~ itree.copy()* and *copy.deepcopy(itree) ~ itree.deepcopy()*.

```
>>> import copy
>>> itree = iTree('root',value={'a':[1,2,3]})
>>> copied_itree=itree.copy()
>>> iTree(itree.tag,value=copy.copy(itree.value)) # root only copy (subtree␣
↪eliminated)
iTree('root', value={'a': [1, 2, 3]})
>>> copied_itree.value is itree.value
False
>>> copied_itree.value['a'] is itree.value['a']
True
>>> deepcopied_itree=itree.deepcopy() # Inner values objects will be copied too
>>> deepcopied_itree_extern=iTree(itree.tag,value=copy.deepcopy(itree.value))
>>> deepcopied_itree.value is itree.value
False
>>> deepcopied_itree.value['a'] is itree.value['a']
False
>>> itree_only_copy=itree.copy_keep_value() # values will be taken over without copy
>>> itree_only_copy_extern=iTree(itree.tag,value=itree.value)
>>> itree_only_copy.value is itree.value
True
```

Some of the structural manipulation commands can be utilized also as an in-depth variant which will run over the nested *iTree*-structure. Use the helper class *.deep* for this propose.

itertree.iTree.**rotate**()
Rotate children of the *iTree*-object n times (n positions) (rotate 1 times means move last item to first position)

If no parameter is given we rotate by one position only.

The rotation can be made in negative direction too (give negative numbers).

In case zero is given the operation is neutral and nothing will be changed.

Note: There is no in-depth counterpart of this method available.

Parameters **n** (*integer*) – number of positions the items should be rotated

itertree.iTree.**reverse**()
Reverse the order of all children in the *iTree*.

If you do not want to change the object itself (in place operation) you might use the iterator *reversed()* instead.

itertree.iTree.deep.**reverse**()

coded in helper-class:

itertree.itree_indepth._iTreeIndepthTree.**reverse**()
Call via **iTree().deep.reverse()**

In-depth reverse of the order of all children in the *iTree*. Same as method *reverse()* but this is the in-depth version of the method. This method dives deeper and the sub-children, sub-sub-children, ... orders are reversed too.

Note: The implementation of this method is recursive for deep trees recursion limit might be reached.

---

itertree.iTree.**sort**()
Sorting operation -> same behavior as sort of lists (parameter description is taken from list documentation).

Note: This is an "in place" operation which changes the content of the object the build-in *sorted()* might be use instead (if the original object should not be changed):

```
>>> a=iTree(subtree=[iTree(3),iTree(2),iTree(4),iTree(1)])
>>> a.render()
iTree()
> iTree(3)
> iTree(2)
> iTree(4)
> iTree(1)
>>> b=iTree(subtree=(a[i] for i in sorted(a.keys())))
iTree()
> iTree(1)
> iTree(2)
> iTree(3)
> iTree(4)
```

Internally in this operation a copied sorted list is created, and afterwards the whole structure is cleared and rebuild based on the sorted list.

The default-operation is to the sort based on the list of keys (tag-family, family_index) pair of the items. The base of the sorting can be modified by changing the *target_type* parameter.

Parameters

- **key** – specifies a function of one argument that is used to extract a comparison key from each list element (for example, key=str.lower). The key corresponding to each item in the list is calculated once and then used for the entire sorting process. The default value of None means that list items are sorted directly without calculating a separate key value.
- **reverse** – is a boolean value. If set to True, then the list elements are sorted as if each comparison were reversed.

itertree.iTree.deep.**sort**()

coded in helper-class:

itertree.itree_indepth._iTreeIndepthTree.**sort**()

Call via **iTree().deep.sort()**

sort operation running also over the deeper levels of the tree -> same behavior as sort of lists (parameter description is taken from list documentation)

In this operation internally a copied sorted list is created, the structure is cleared and rebuild based on the sorted list. The default-operation is to the sort based on the list of keys (tag-family.family_index) pair of the items. This might be modified by changing the target_type.

Warning: In case of really deep *iTree`s (depth >100) the sorting might take a lot of time. We made a test with an `iTree* containing ~2500 items and a depth of 9000. Result was: *itree.all.sort() time: 83.772834 s* (Python 3.9).

Note: The implementation of this method is recursive for deep trees recursion limit might be reached.

Parameters

- **key** – specifies a function of one argument that is used to extract a comparison key from each list element (for example, key=str.lower). The key corresponding to each item in the list is calculated once and then used for the entire sorting process. The default value of None means that list items are sorted directly without calculating a separate key value.
- **reverse** – is a boolean value. If set to True, then the list elements are sorted as if each comparison were reversed.

Additionally we support following rearrangement functions:

```
>>> root[0], root[1], root[2] = root[2], root[0], root[1]
>>> root[0:3] = root[2], root[0], root[1]
  File "<string>", line 1
    root[0:3] = root[2], root[0], root[1]
              ^
SyntaxError: invalid syntax

During handling of the above exception, another exception occurred:

Traceback (most recent call last):
  File "E:\projects\privat\itertree\src\itertree\examples\itree_docu_examples.py",␣
↪line 130, in exec_and_print
    exec(command)
  File "<string>", line 1, in <module>
  File "E:\projects\privat\itertree\src\itertree\itree_main.py", line 1441, in __
↪setitem__
    return [it_setitem(old_items[i].idx, new) for i, new in enumerate(value)]
  File "E:\projects\privat\itertree\src\itertree\itree_main.py", line 1441, in
↪<listcomp>
    return [it_setitem(old_items[i].idx, new) for i, new in enumerate(value)]
  File "E:\projects\privat\itertree\src\itertree\itree_main.py", line 1473, in __
↪setitem__
    old_item_idx = family[0].idx
IndexError: list index out of range

>>> root[2], root[0], root[1] = root[0:3]
```

There might be cases where those in-place rearrangements might not work (We have not tested all possible combinations here) and be aware that in this kind of operations it can be that there are implicit copies (same as *itree.copy()*) of the original object-instances created.

In the following pseudo mathematical operations the result will always be a new *iTree* instance. Flags are not considered in those operations. Addition and multiplication is not permutable because the first object gives the tag,value for the resulting object!

The addition of *iTree*'s is possible the result contains always the properties of the first added item and the children of the second added item are appended to the items of the fiorst one by creating a copy.

```
>>> a = iTree('a', value={'mykey': 1}, subtree=[iTree('a1'), iTree('a2')])
>>> b = iTree('b', subtree=[iTree('b1'), iTree('b2')])
>>> itree = a + b
>>> repr(itree) # repr() is required to get the un-shorten representation of iTree␣
↪(str() shortens the subtree-parameter)
iTree('a', value={'mykey': 1}, subtree=[iTree('a1'), iTree('a2'), iTree('b1'), iTree(
↪'b2')])
```

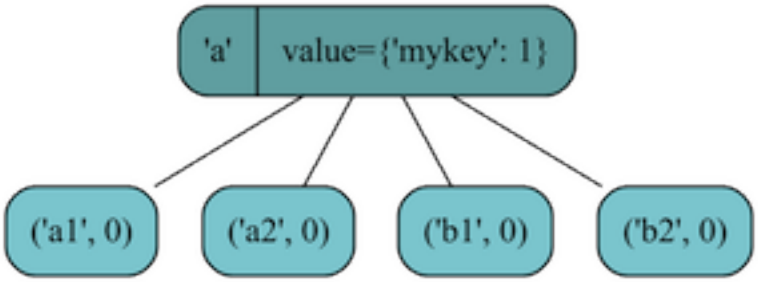


Fig. 3: Figure showing the resulting iTree

Multiplication of a *iTree* is possible too the result is a list of *iTree* copies of the original one.

```
>>> itree_list = iTree('a') * 1000  # creates a list of 1000 copies of the original␣
↪iTree
>>> itree_list[0]==itree_list[1] # items are equal
True
>>> itree_list[0] is itree_list[1] # but we have different instances
False
>>> root = iTree('root')
>>> root.extend(iTree('a') * 10000) # append all 10000 items as children to root
>>> len(root)
10000
```

In case two *iTree*-objects are multiplied in the result each children of first will be mixed with the children of the second in the scheme: child1_0,child2_0,child1_0,child2_1,...child1_1,child2_0,child1_1,child2_1...

```
>>> itree1=iTree('one',1,[iTree(1.0),iTree(1.1),iTree(1.2)])
>>> itree2=iTree('two',1,[iTree(2.0),iTree(2.1),iTree(2.2)])
>>> itree_mul=itree1*itree2
>>> itree_mul.render()
iTree('one', value=1)
 > iTree(1.0)
 > iTree(2.0)
 > iTree(1.0)
 > iTree(2.1)
 > iTree(1.0)
```

(continues on next page)

(continued from previous page)

```
> iTree(2.2)
> iTree(1.1)
> iTree(2.0)
> iTree(1.1)
> iTree(2.1)
> iTree(1.1)
> iTree(2.2)
> iTree(1.2)
> iTree(2.0)
> iTree(1.2)
> iTree(2.1)
> iTree(1.2)
> iTree(2.2)
```

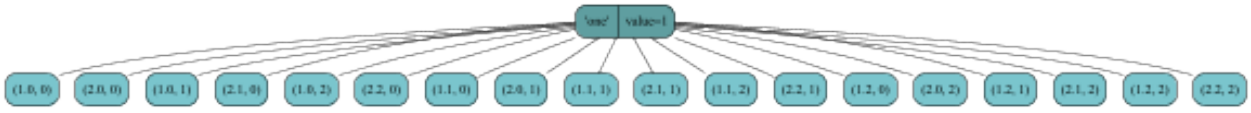


Fig. 4: Figure showing the resulting iTree after multiplication

The subtraction of two iTrees is supported too. The base of operation is the tag_idx of the items. Items with same tag_idx are eliminated (only in case they have same value too). With different values we try to calculate the the difference of the value objects if this is not possible the value will kept unchanged (value of the minuend is kept).

```
>>> itree1=iTree('one',1,[iTree('a',1.0),iTree('a',1.1),iTree('a','str')])
>>> itree1[0]-itree1[1] # same tage different value -> diff of value is calculated␣
↪(if possible)
iTree(value=-0.10000000000000009)
>>> itree1[0]-itree1[2] # same tage different value -> diff not possible minuend is␣
↪kept
iTree(value=1.0)
>>> sub_tree=itree1-itree1 # minus same object
>>> sub_tree.tag # tag eliminated
<class 'itertree.itree_helpers.NoTag'>
>>> sub_tree.value # value eliminated
<class 'itertree.itree_helpers.NoValue'>
>>> sub_tree.render() # subtree eliminated
iTree()
```

Subtraction of same iTree delivers an empty iTree object (tag=NoTag; value=NoValue).

2.5 Item Access

In this chapter we will dive in the "magic" of the *iTree.get* object.

The user can choose in between the common and the specific target access. The common access is more flexible related to the possibility of giving mixed target_paths and it is a bit more "lazy". The specific access should be used if the quickest possible access is required (depending on the given target type it is ~2-6 times quicker compared to the common access). And it can be that the specific access is needed because of conflicting target content (e.g. if an integer tag is used in iTree, it cannot be reached via common access because the target will be interpreted as an absolute index access (higher priority the tag access))

Note: The common target access is also used when ever a item must be targeted in other functionalities like *move()*

or *insert()*!

For common target access we have the following methods:

itertree.iTree.**__getitem__**()
Main common get method for children (first level items).

In case the given targets is a absolute index or a key (tag,family-index) pair the method will deliver a unique item back. This operation is prioritized over the other operations.

For all other targets the method will deliver a list with the targeted items as result.

In some cases an empty list might be delivered and no exception might be raised (e.g. filter query delivers no match).

In case user likes to have other return-types he might check the other available get methods (*get()*, *get.single()*, *get.iter()*) or he might also use the itertree helper method *getter_to_list()* to convert any of the possible results into a list.

Except In case of no match (even if a part is not matching (e.g. one index in an index-list) the method will raise a KeyError (no matching target given); IndexError (no matching index given) or ValueError (no valid type of target given).

Parameters **target** (*Union[int,tuple,list,slice]*) – target object targeting a child or multiple children in the ´iTree´. Possible types are:

- *index* - absolute target index integer (fastest operation)
- *key* - key tuple (family_tag, family_index)
- *index-slice* - slice of absolute indexes
- *key-index-slice* - tuple of (family_tag, family_index_slice)
- *target-list* - absolute indexes or keys to be replaced (indexes and keys can be mixed)
- *key-index-list* - tuple of (family_tag, family_index_list)
- *tag* - family_tag object targeting a whole family
- *tag-set* - a set of family-tags targeting the items of multiple families
- *itree_filter* - method (callable) for filtering the children of the object
- *all-children* - if build-in *iter* or . . . *`(Ellipsis) is given a list of all children will be given (same like list(itree.__iter__())*)

Return type Union[*iTree*,list]

Returns Target was *index* or *key* -> one *iTree* item will be given; for all other targets a list will be delivered.

itertree.iTree.**get**()

coded in helper-class:

itertree.itree_getitem._iTreeGetitem.**__call__**()
Call via **iTree().get()**

Main get method for items that supports in-depth level-wise access too.

If only one parameter is given get behaves like *__getitem__()* except that a default parameter can be given so that it will be delivered (the normal method would raise an exception in this case). In case no default is given the exception will be raised too.

Warning: The default parameter must be given as a keyword argument only e.g.:*get(1,default=None)*. All unnamed arguments given will always be interpreted as a target definitions!

In case the method got more than one unnamed argument an in-depth target access will be performed. Each parameter will target in this case the next nested level of the tree.

The method can be seen as a replacement of the operation *self[target_deep[0]][target_deep[1]]...[target_deep[-1]]*

Note: But be aware that the results in the different levels might not be unique and therefore in detail the method will behave different as the simple direct targeting (which will raise an exception in this case). This method will create an iterator of all (branched) findings in the deepest targeted level instead.

In this case the method will deliver an iterator of all the findings in the mostlowest level targeted. The iterator is always flatten even that in higher levels we might have multiple findings.

E.g. the user might have build a tree like this:

```
>>> root_tree.render()
iTree('root', value=0)
 > iTree('sub', value=1)
 .  > iTree('subsub', value=5)
 > iTree('sub1', value=2)
 .  > iTree('subsub', value=6)
 > iTree('sub2', value=3)
 .  > iTree('subsub', value=7)
 > iTree('sub', value=4)
 .  > iTree('subsub', value=8)
 >>> get('sub','subsub')
 [iTree('subsub', value=5), iTree('subsub', value=8)]
```

The reason for this result is that the first match is not unique and so the sub-items in the target levels are combined into on flatten result.

The return of this method can be the following:

1. Pure index and key list is given -> single target -> iTree object should be delivered
2. list of all found items
3. No match found an KeyError or IndexError will be raised

Except In case no matching item is found a KeyError or IndexError is raised. In case of invalid targets TypeError or ValueError will be raised.

Parameters

- **target** (*Union[int,tuple,list,slice]*) – level 0 target object targeting a child or multiple children in the ´iTree´. Possible types are:
  - *index* - absolute target index integer (fastest operation)
  - *key* - key tuple (family_tag, family_index)
  - *index-slice* - slice of absolute indexes
  - *key-index-slice* - tuple of (family_tag, family_index_slice)
  - *target-list* - absolute indexes or keys to be replaced (indexes and keys can be mixed)

- *key-index-list* - tuple of (family_tag, family_index_list)
- *tag* - family_tag object targeting a whole family
- *tag-set* - a set of family-tags targeting the items of multiple families
- *itree_filter* - method (callable) for filtering the children of the object
- *all-children* - if build-in *iter()* or ... (Ellipsis) is given a list of all children will be given (same result as *list(itree.__iter__())*)

- ***target_path** – in-depth targets iterable of targets for the different levels 1-n The supported targets in each level are (same like *__getitem__()*:
  - *index* - absolute target index integer (fastest operation)
  - *key* - key tuple (family_tag, family_index)
  - *index-slice* - slice of absolute indexes
  - *key-index-slice* - tuple of (family_tag, family_index_slice)
  - *target-list* - absolute indexes or keys to be replaced (indexes and keys can be mixed)
  - *key-index-list* - tuple of (family_tag, family_index_list)
  - *tag* - family_tag object targeting a whole family
  - *tag-set* - a set of family-tags targeting the items of multiple families
  - *itree_filter* - method (callable) for filtering the children of the object
  - *all-children* - if build-in *iter()* or ... (Ellipsis) is given a list of all children will be given (same result as *list(itree.__iter__())*)
- **default** – The parameter must be given as keyword parameter! The object given will be delievred in case of issues. If the parameter is not set (==*Exception*) exceptions will be raised in case of issues.

Return type Union[*iTree*,list]

Returns iTree object or list of objects

itertree.iTree.get.**single**()

coded in helper-class:

itertree.itree_getitem._iTreeGetitem.**single**()

Call via **iTree().get.single()**

In general the methods does same like the "normal" *get()* but the method delivers only single (unique) results. In case *get()* delivers multiple items this method will raise an Exception or delivers the default value (if defined).

Note: In case the match contains a list with only one element the result is unique too. The method will unpack the unique item from the iterable and return it in this case.

Except If default parameter is not set an KeyError or IndexError will be raised. If result is not unique a ValueError will be raised

Parameters

- **target** (*Union[int,tuple,list,slice]*) – level 0 target object targeting a child or multiple children in the ´iTree´. Possible types are:

- *index* - absolute target index integer (fastest operation)
- *key* - key tuple (family_tag, family_index)
- *index-slice* - slice of absolute indexes
- *key-index-slice* - tuple of (family_tag, family_index_slice)
- *target-list* - absolute indexes or keys to be replaced (indexes and keys can be mixed)
- *key-index-list* - tuple of (family_tag, family_index_list)
- *tag* - family_tag object targeting a whole family
- *tag-set* - a set of family-tags targeting the items of multiple families
- *itree_filter* - method (callable) for filtering the children of the object
- *all-children* - if build-in *iter()* or ... (Ellipsis) is given a list of all children will be given (same result as *list(itree.__iter__())*)

***target_path** – in-depth targets iterable of targets for the different levels 1-n The supported targets in each level are (same like *__getitem__()*:

- *index* - absolute target index integer (fastest operation)
- *key* - key tuple (family_tag, family_index)
- *index-slice* - slice of absolute indexes
- *key-index-slice* - tuple of (family_tag, family_index_slice)
- *target-list* - absolute indexes or keys to be replaced (indexes and keys can be mixed)
- *key-index-list* - tuple of (family_tag, family_index_list)
- *tag* - family_tag object targeting a whole family
- *tag-set* - a set of family-tags targeting the items of multiple families
- *itree_filter* - method (callable) for filtering the children of the object
- *all-children* - if build-in *iter()* or ... (Ellipsis) is given a list of all children will be given (same result as *list(itree.__iter__())*)

default (*object*) – If parameter is set in case of no match the default object will be delivered. If parameter is not set an Exception will be raised

Return type Union[*iTree*,object]

Returns found single item or default (in case default is set)

itertree.iTree.get.**iter**()

coded in helper-class:

itertree.itree_getitem._iTreeGetitem.**iter**()

Method call via **iTree().get.iter()**

In general the methods does same like the "normal" *get()* but the method delivers an iterator results. In case *get()* delivers a single items this method will deliver [item].

If no match is found will be delivered the default value (if defined).

If no target is given *[self]* will be delivered.

Warning: It can be that an empty iterator is delivered and no Exception is raised in this case!

Note: In case the target item should be iterated afterwards this method is recommended because some operations are quicker then the standard *get()*.

Except If default parameter is not set an *KeyError* or *IndexError* will be raised. If result is not unique a *ValueError* will be raised.

Parameters

- **target** (*Union[int,tuple,list,slice]*) – level 0 target object targeting a child or multiple children in the ´iTree´. Possible types are:
  - *index* - absolute target index integer (fastest operation)
  - *key* - key tuple (family_tag, family_index)
  - *index-slice* - slice of absolute indexes
  - *key-index-slice* - tuple of (family_tag, family_index_slice)
  - *target-list* - absolute indexes or keys to be replaced (indexes and keys can be mixed)
  - *key-index-list* - tuple of (family_tag, family_index_list)
  - *tag* - family_tag object targeting a whole family
  - *tag-set* - a set of family-tags targeting the items of multiple families
  - *itree_filter* - method (callable) for filtering the children of the object
  - *all-children* - if build-in *iter()* or ... (Ellipsis) is given a list of all children will be given (same result as *list(itree.__iter__())*)
- ***target_path** – in-depth targets iterable of targets for the different levels 1-n The supported targets in each level are (same like *__getitem__()*:
  - *index* - absolute target index integer (fastest operation)
  - *key* - key tuple (family_tag, family_index)
  - *index-slice* - slice of absolute indexes
  - *key-index-slice* - tuple of (family_tag, family_index_slice)
  - *target-list* - absolute indexes or keys to be replaced (indexes and keys can be mixed)
  - *key-index-list* - tuple of (family_tag, family_index_list)
  - *tag* - family_tag object targeting a whole family
  - *tag-set* - a set of family-tags targeting the items of multiple families
  - *itree_filter* - method (callable) for filtering the children of the object
  - *all-children* - if build-in *iter()* or ... (Ellipsis) is given a list of all children will be given (same result as *list(itree.__iter__())*)
- **default** (*object*) – If parameter is set in case of no match the default object will be delivered. If parameter is not set an Exception will be raised

Return type Union[list,blist,Iterator]

Returns An iterator or a list with a single item will be delivered

The first method *__getitem__()* targets first level only (access via "brackets-operation" *itree[]*). All other methods are capable to target via in-depth access (realized via multiple parameters that can be given to the method).

Warning: The usage of target_paths are just supported by the *get*-subclass. The following methods supporting target-paths containing mixed target-items (different types):

- *get()*
- *get.single()*
- *get.iter()*

The other methods in *get*-subclass support only target-paths with unique targets (matching to the specific method).

The method *__getitem__()* does not support target-paths it just takes targets targeting the level 1 children only!

The return type of the common access functions *__getitem__()`and `get()* depends on the given target-type:

- absolute index, key (family tag-index pair) -> unique *iTree*-item will be delivered
- all other targets (multi target operations) -> *list* of matching items (in some case a *blist* object might be delivered)

The *get.single()* method delivers only single *iTree*-objects and *get.iter()* delivers an iterator of the matches found.

For the specific access the following methods are available:

itertree.iTree.get.**by_idx**()

coded in helper-class:

itertree.itree_getitem._iTreeGetitem.**by_idx**()

Call via **iTree().get.by_idx()**

Get items by absolute index.

This is the quickest getter function we have in *iTree* . As parameters the user can give just integers.

For in-depth operations the user can give an index-path (pointer).

Parameters

- **idx** (*int*) – first item index
- ***idx_path** – in case we have a in-depth operation we use index path and first given idx will be integrated in the operation (give level 1- n index)
- **default** (*object*) – This is a named parameter only! If default is given the default objet will be returned in case of internal exceptions. If default is Exception an exception is raised

Return type *iTree*

Returns target item

itertree.iTree.get.**by_idx_slice**()

coded in helper-class:

itertree.itree_getitem._iTreeGetitem.**by_idx_slice**()

Call via **iTree().get.by_idx_slice()**

Get items by absolute index slice.

For in-depth operations the user can give multiple parameters (a slices per level). The findings are combined to a final flatten list.

The operation can be mixed with normal indexes.

Note: If the user likes to target all items in a level he can give the slice(None) object which will iterate over all children of the level

To target a single item slice(n,n+1) must be given.

Parameters

- **idx_slice** (*slice*) – absolute index slice for level 0 access (a slice object must be given!)
- ***idx_path** – Give multiple parameters (one slice per level)
- **default** (*object*) – This is a named parameter only! If default is given the default objet will be returned in case of internal exceptions. If default is Exception an exception is raised

Return type list

Returns list of target iTree-items

itertree.iTree.get.**by_idx_list**()

coded in helper-class:

itertree.itree_getitem._iTreeGetitem.**by_idx_list**()

Call via **iTree().get.by_idx_list()**

Get items via absolute index lists.

For in-depth operations the user can multiple parameters (one parameter per level) each parameter must be an absolute index list.The findings are combined to a final flatten list.

Note: The user can give ... (Ellipsis) to target all children in a specific level

Parameters

- **idx_list** (*list*) – list of absolute indexes targeting level 0
- ***idx_list_path** – Give multiple parameters (one index list per level)
- **default** (*object*) – This is a named parameter only! If default is given the default objet will be returned in case of internal exceptions. If default is Exception an exception is raised

Return type list

Returns list of targeted iTree-items

itertree.iTree.get.**by_tag_idx**()

coded in helper-class:

itertree.itree_getitem._iTreeGetitem.**by_tag_idx**()

Call via **iTree().get.by_tag_idx()**

Get items by tag-idx-key (tag,family-index) tuple.

This is the quickest getter function available for tag-idx access (comparable to keys in dicts) we have in iTree. The parameters must be (tag, family-idx) tuples.

For in-depth operations the user can give a tag_idx_path. In this case the methods dives into the tree and extracts the matching items in the different levels

Parameters

- **tag_idx** (*tuple*) – level one tag-idx-key
- ***idx_path** – In-depth parameters each additional parameter must be a tag-idx-key target the item in the specific level
- **default** (*object*) – This is a named parameter only! If default is given the default objet will be returned in case of internal exceptions. If default is Exception an exception is raised

Return type *iTree*

Returns targeted item

itertree.iTree.get.**by_tag_idx_slice**()

coded in helper-class:

itertree.itree_getitem._iTreeGetitem.**by_tag_idx_slice**()

Call via **iTree().get.by_tag_idx_slice()**

Get items via tag_idx_key containing a slice in the family index tuple(tag,family-index-slice). The user must give here a slice object.

For in-depth operation additional tag_idx_keys containing slices can be added. To target a whole family the user may give the slice(None). The results in the different levels are merged to a flatten list containing all matches in the highest targeted level.

Parameters

- **tag_idx_slice** (*tuple*) – tuple of tag and family-index-slice
- ***tag_idx_path** – Give additional tag-idx-slices per target level in-depth of the iTree
- **default** (*object*) – This is a named parameter only! If default is given the default objet will be returned in case of internal exceptions. If default is Exception an exception is raised

Return type list

Returns list of targeted iTree-items

itertree.iTree.get.**by_tag_idx_list**()

coded in helper-class:

itertree.itree_getitem._iTreeGetitem.**by_tag_idx_list**()

Call via **iTree().get.by_tag_idx_list()**

Get items by giving a tag-family-index-list tuple.

For in-depth operation the user can add more tag-family-index-list tuples as additional parameters targeting the in-depth levels of the iTree object.

To target all family items of a specific level the „,-object`(Ellipsis) can be placed as parameter.

Parameters

- **tag_idx_list** (*tuple*) – tuple of tag and a list of family-indexes (e.g. ('my-tag',[1,2,3]))
- ***tag_idx_list_path** – Additional parameters each containing a tuple with tag and a list of indexes for each in-depth level of the iTree
- **default** (*object*) – This is a named parameter only! If default is given the default objet will be returned in case of internal exceptions. If default is Exception an exception is raised

Return type list

Returns list of targeted iTree-items

itertree.iTree.get.**by_tag**()

coded in helper-class:

itertree.itree_getitem._iTreeGetitem.**by_tag**()

Call via **iTree().get.by_tag()**

Get family items by given tag.

This is the quickest getter function for families.

For in-depth operation the user can give as additional parameters more tags (one tag per level). The findings are cumulated and delivered as a flattened item list.

Parameters

- **tag** (*hashable*) – Family tag targeting all items inside the family
- ***tag_path** – hashable tags targeting the deeper levels of iTree
- **default** (*object*) – This is a named parameter only! If default is given the default objet will be returned in case of internal exceptions. If default is Exception an exception is raised

Return type list

Returns list of targeted iTree-items

itertree.iTree.get.**by_tags**()

coded in helper-class:

itertree.itree_getitem._iTreeGetitem.**by_tags**()

Call via **iTree().get.by_tags()**

Here the user gives an iterable of tags for the to be targeted families (multiple families). The targeted items are combined in one list.

For in-depth operation the user can give additional parameters containing tag-iterables per target levels. The result is cumulated and delivers all found items in the deepest targeted level.

The user might give also single tags (but it's recommended to put them in a list -> see the warning).

Warning: Tuples are interpreted as iterables in this case! If the user likes to target a single tag which is a tuple-object he must give an additional iteration level (e.g. tag=(1,2) tags([(1,2)] must be given to target the tag-family (1,2)).

Parameters

- **tags** (*Iterable*) – Iterable of family tags
- ***tags_path** – Additional family-tag iterables for deeper levels of teh iTree
- **default** (*object*) – This is a named parameter only! If default is given the default objet will be returned in case of internal exceptions. If default is Exception an exception is raised

Return type list

Returns list of target items

itertree.iTree.get.**by_level_filter**()

coded in helper-class:

itertree.itree_getitem._iTreeGetitem.**by_level_filter**()

Call via **iTree().get.by_level_filters()**

Get items by level-filters.

For in-depth operation additional parameters can be given each is a level-filter for the next level.

In case the build-in *iter*-method is given (without parameters) all items in the level will be considered (no filtering). The level filtering is always a hierarchical filtering.

Parameters

- **filter_method** (*Method*) – filter_method analysis the itree-items and delivers *True* for a match and *False* for no match (filtered out)
- ***filter_method_path** – Additional parameters for filter_methods for the deeper levels of the iTree.
- **default** (*object*) – This is a named parameter only! If default is given the default objet will be returned in case of internal exceptions. If default is Exception an exception is raised

Return type list

Returns list of filtered iTree-items found in the deepest targeted level

2.5.1 Target description

Beside the construction of the object the access to it's items is the second core-functionality for a tree object.

In *iTree* this is one of the most complex functionalities available. The reason is the wide range of different possible targets that are supported. It's recommended that the user reads the following explanations and examples carefully to understand the full range of functionalities available related to the access of children stored in *iTree*.

But even for less experienced users the easy access via *itree[index]* (list like counterpart) or *itree[tag_idx_key]* (dict-like counterpart) will work in most cases.

Lets build a small example *iTree*-object and let's see with which target definitions we can access the children in this object:

```
>>> root = iTree('root')
>>> root += iTree('child', value=0)
>>> root += iTree('child', value=1)
>>> root += iTree('child', value=2)
>>> root += iTree('child', value=3)
>>> root += iTree('child', value=4)
>>> root += iTree(1, value=5)
>>> root += iTree(('child',1), value='tag conflict')
>>> # any hashable object can be used as tag!
>>> root += iTree((1, 2, 3), value=6)  # any hashable object can be used as tag!
>>> root.render()
iTree('root')
 > iTree('child', value=0)
 > iTree('child', value=1)
 > iTree('child', value=2)
 > iTree('child', value=3)
 > iTree('child', value=4)
 > iTree(1, value=5)
 > iTree(('child', 1), value='tag conflict')
 > iTree((1, 2, 3), value=6)
```

Fig. 5: Figure showing the resulting iTree

In the following examples have a special look on the result types delivered (single-targets -> *iTree*-child and multi-targets -> *list* of matching children in *iTree*-order):

- Target via **absolute index**:

  The absolute index is like the index in lists and targets the children counting from 0. And as in lists negative values are supported too (count index from the last index down).

  This operation is the fastest way to target a item in *iTree*-objects.

  This operation has highest priority in common access. It will "cover" the tag access to families (based on integer-type tags).

  The specific access method *get.by_idx()* is faster and can be used too.

  This is a single/unique target therefore it delivers directly the targeted *iTree*-child-object.

```
>>> # Common index access:
>>> root[0] # absolute index access
iTree('child', value=0)
>>> root[-1] # absolute index access (negative values)
iTree((1, 2, 3), value=6)
>>> root[5] # This child is not targeted in the next step even that it's␣
↪tag==1!
iTree(1, value=5)
>>> root[1] # The absolute index access has higher priority than access␣
↪via tags
iTree('child', value=1)
>>> # Specific index access:
>>> root.get.by_idx(0) # absolute index access
iTree('child', value=0)
>>> root.get.by_idx(-1) # absolute index access (negative values)
iTree((1, 2, 3), value=6)
>>> root.get.by_idx(5) # This child is not targeted in the next step␣
↪even that it's tag==1!
iTree(1, value=5)
>>> root.get.by_idx(1) # The absolute index access has higher priority␣
↪than access via tags
iTree('child', value=1)
```

- Target via **absolute index-slice**:

  As in lists the slicing of the absolute index is supported too.

  But the result is no more unique, therefore the operation will return a *list* or *blist*.

  The specific access method for this target is *get.by_idx_slice()* but the method parameter(s) must be slice object(s).

```
>>> # Common index-slice access:
>>> root[1:3]
blist([iTree('child', value=1), iTree('child', value=2)])
>>> # Specific index-slice access:
```

(continues on next page)

(continued from previous page)

```
>>> root.get.by_idx_slice(slice(1,3))
blist([iTree('child', value=1), iTree('child', value=2)])
```

- Target via **absolute index-list**:

  We can target multiple children by giving a list of indexes. The resulting list represents the order of indexes the user gave.

  > **Warning:** Duplicated indexes will deliver duplicated items in the result. Especially in case of in-depth access this should be avoided, because the results can be very confusing.

  No unique result, a *list* will be returned.

  The specific access method for this target is *get.by_idx_list()*.

```
>>> # Common index-list access:
>>> root[[0, 2]]
[iTree('child', value=0), iTree('child', value=2)]
>>> # same as:
>>> [root[0],root[2]]
[iTree('child', value=0), iTree('child', value=2)]
>>> root[[2, 0, 2]]  # The target-order is kept (even multiple same␣
↪items are kept)
[iTree('child', value=2), iTree('child', value=0), iTree('child',␣
↪value=2)]
>>> # Specific index-list access:
>>> root.get.by_idx_list([0, 2])
[iTree('child', value=0), iTree('child', value=2)]
```

- Target via **tag-idx** (key):

  This tag-idx-key (family-tag, family-index) is unique for any child. The second item in the *tuple* is the family-index. This gives the position of the child in the related tag-family-list (negative values supported too -> count from the end). A tag-idx-key is internally identified via the given *tuple* of length 2. (For downward compatibility the *TagIdx*-helper-object is still available and can be used for this case too).

  This operation has highest priority and covers tag access to families based on tuples and this operation is the second fastest way (after absolute index access) to target a object in iTrees.

  The key is unique therefore the operation delivers a single *iTree*-object.

  The specific access method for this target is *get.by_tag_idx()*.

```
>>> # Common tag-idx-key access (given as tuple)
>>> # and how it must be used for targeting in other commands e.g.␣
↪`insert()` or `move()`:
>>> root[('child', 0)]
iTree('child', value=0)
>>> root['child', 0]  # lazy way to give the tag-idx-key
iTree('child', value=0)
>>> root[('child', -1)]  # negative family-index, is supported too
iTree('child', value=4)
>>> root[('child',1), 0] # This child is not targeted in the next step␣
↪even that it's tag==('child',1)!
iTree(('child', 1), value='tag conflict')
```

(continues on next page)

(continued from previous page)

```
>>> root[('child', 1)] # The key access has higher priority than access␣
↪via tags
iTree('child', value=1)
>>> # Specific tag-idx access (must be given as tuple)
>>> root.get.by_tag_idx(('child', 0))  # Give the tuple; multiple␣
↪parameters would target in-depth!
iTree('child', value=0)
```

- Target via **(family-tag, family-index-slice) - pair**:

  Slice operations on family_index is supported but the slice object must be given explicit *slice(start,end,step)*.

  Note: In this case we **cannot** use the slice definition via double dots like *[0:3:2]* . We must define a *slice()*-object.

  Result is not unique a item therefore a *list* or *blist* with the selected items will be returned.

  The specific access method for this target is *get.by_tag_idx_slice()*.

```
>>> # Common tag-idx-slice access (given as tuple)
>>> root[('child',slice(0,3,2))]
blist([iTree('child', value=0), iTree('child', value=2)])
>>> root['child',slice(0,3,2)] # lazy input supported
blist([iTree('child', value=0), iTree('child', value=2)])
>>> # Specific tag-idx-slice access (must be given as tuple)
>>> root.get.by_tag_idx_slice(('child',slice(0,3,2)))
blist([iTree('child', value=0), iTree('child', value=2)])
```

- Target via **(family-tag, family-index-list) - pair**:

  Giving a index list of family indexes to target the children is supported.

  The order of the delivered items is the order of indexes given and duplicates are kept too.

  Result is a *list* of matching children.

  The specific access method for this target is *get.by_tag_idx_list()*.

```
>>> # Common tag-idx-list access (given as tuple)
>>> root[('child',[0,2])]
[iTree('child', value=0), iTree('child', value=2)]
>>> root[('child',[0,2])] # lazy input supported
[iTree('child', value=0), iTree('child', value=2)]
>>> # Specific tag-idx-list access (must be given as tuple)
>>> root.get.by_tag_idx_list(('child',[0,2]))
[iTree('child', value=0), iTree('child', value=2)]
```

- Target a whole **tag-family**:

  Here we target all items that have the same tag (same family).

  As already shown this object has lower priority, in case of conflicts (with idx or tag_idx) the user should use the specific access method or he puts the tag as a single value in a set *itree[{tag}]* but the access is much slower as the specific one.

  Result is a *list* with all children having the target tag (whole tag-family).

  The specific access method for this target is *get.by_tag()*.

```
>>> root['child'] # In case of no conflicts a given family tag delivers␣
↪the family directly
blist([iTree('child', value=0), iTree('child', value=1), iTree('child',␣
↪value=2), iTree('child', value=3), iTree('child', value=4)])
>>> # specific tag-family access
>>> root.get.by_tag('child')
blist([iTree('child', value=0), iTree('child', value=1), iTree('child',␣
↪value=2), iTree('child', value=3), iTree('child', value=4)])
>>> root.get.by_tag(('child',1)) # target ('child',1) tag-family with␣
↪root[('child',1)] the tag-idx is targeted!
[iTree(('child', 1), value='tag conflict')]
>>> # The tag=('child',1) is a family tag not a tag-idx-key!
>>> root.get.by_tag(1) # target again an item which cannot be reached␣
↪via root[1]
[iTree(1, value=5)]
>>> root[{1}] # In case of conflicts the user can use a tag-set with one␣
↪item too (slower as specific access)
[iTree(1, value=5)]
>>> # The tag=1 is a family tag not an absolute index!
```

- Target multiple tag-families **tag-families-set**:

  If a set of multiple tags is given the children of the different families are combined in the output list.

  Result is a *list* with all children having the target tag that were targeted. The order of the items is the order of the families in the set.

  The specific access method for this target is *get.by_tags()*. Different to the common access we can give here also lists or tuples as parameter(s) the order will be kept but duplicates will be delivered as given too.

```
>>> root[{(1,2,3),1,('child',1)}] # order of tags in the set is kept in␣
↪the result
[iTree(1, value=5), iTree((1, 2, 3), value=6), iTree(('child', 1), value=
↪'tag conflict')]
>>> root[{1,('child',1),(1,2,3),}]
[iTree(1, value=5), iTree((1, 2, 3), value=6), iTree(('child', 1), value=
↪'tag conflict')]
>>> root.get.by_tags([1,('child',1),(1,2,3),]) # here the order of th␣
↪tags in the list is kept; duplicates will be delivered too
[iTree(1, value=5), iTree(('child', 1), value='tag conflict'), iTree((1,␣
↪2, 3), value=6)]
```

- Target children via a **filter-method**:

  A filter-method is a function that analysis the children object related to the properties, attributes, etc. and that generates at the end a True/False (match/ no match) return per item. By this the children are filtered and only the matching ones will be integrated into the result.

  We have multiple items in the result a *list* will be returned.

  The specific access method for this target is *get.by_level_filter()*

```
>>> # The following EXCEPTION is expected:
>>> root[lambda i: i.value%2==0] # filters all children which contains␣
↪an even value, but we have an exception:
Traceback (most recent call last):
...
TypeError: lambda: raised an exception in filter-calculation, the 6.␣
↪child iTree(('child', 1), value='tag conflict') is incompatible with␣
↪the calculation
```

(continues on next page)

(continued from previous page)

```
>>> root[lambda i: type(i.value) is int and i.value%2==0] # ensure that␣
↪the filter-calculation matches to any child!
[iTree('child', value=0), iTree('child', value=2), iTree('child',␣
↪value=4), iTree((1, 2, 3), value=6)]
>>> root[(lambda i: i.value==2)] # This filter targets in our case one␣
↪value only
[iTree('child', value=2)]
>>> root.get.by_level_filter(lambda i: type(i.value) is int and i.value
↪%2==0) # ensure that the filter-calculation matches to any child!
[iTree('child', value=0), iTree('child', value=2), iTree('child',␣
↪value=4), iTree((1, 2, 3), value=6)]
>>> root.get.by_level_filter(lambda i: i.value==2) # This filter targets␣
↪in our case one value only
[iTree('child', value=2)]
```

- Target all children via a build-in **iter** or **...** (Ellipsis):

  The user can target all children of the *iTree*-object if he gives the *iter* or ... build-in function as a target.

  This function may make no sense from the first view because it's equivalent to the main children iterator *__iter__()*. But we will see that the option is very helpful in target_paths.

  This results in multiple items and a list is returned.

```
>>> root[iter] # give build in iter to target all children
blist([iTree('child', value=0), iTree('child', value=1), iTree('child',␣
↪value=2), iTree('child', value=3), iTree('child', value=4), iTree(1,␣
↪value=5), iTree(('child', 1), value='tag conflict'), iTree((1, 2, 3),␣
↪value=6)])
>>> list(root) # is the recommended equivalent function for this but␣
↪here we need must create the list explicit from the iterator
[iTree('child', value=0), iTree('child', value=1), iTree('child',␣
↪value=2), iTree('child', value=3), iTree('child', value=4), iTree(1,␣
↪value=5), iTree(('child', 1), value='tag conflict'), iTree((1, 2, 3),␣
↪value=6)]
>>> root[(lambda i: True)] # Delivers also the same result but is much␣
↪slower
[iTree('child', value=0), iTree('child', value=1), iTree('child',␣
↪value=2), iTree('child', value=3), iTree('child', value=4), iTree(1,␣
↪value=5), iTree(('child', 1), value='tag conflict'), iTree((1, 2, 3),␣
↪value=6)]
```

- Use different targets to target children in the first level via a **target-list**:

  In a target list (instead of a absolute index only list) the user can combine the different targets already explained (cumulate the targets).

  The result is a flatten list that combines all those targeted children. The order of the children is defined by the order of given targets and duplicates will be kept!

  Mixed target lists can only be used via common access methods.

```
>>> # Here we target absolute index, absolute index, tag-idx-key,family-
↪set,filter
>>> root[[0,1,('child', 1),{1},lambda i: type(i.value) is int and i.
↪value>4]] # in result the iTree children order is kept and duplicates␣
↪are deleted
```

(continues on next page)

(continued from previous page)

```
[iTree('child', value=0), iTree('child', value=1), iTree('child',␣
↪value=1), iTree(1, value=5), iTree(1, value=5), iTree((1, 2, 3),␣
↪value=6)]
>>> root[[{1},('child', 1),lambda i: type(i.value) is int and i.value>4,
↪0,1]] # same targets in other order delivers same result
[iTree(1, value=5), iTree('child', value=1), iTree(1, value=5), iTree((1,
↪ 2, 3), value=6), iTree('child', value=0), iTree('child', value=1)]
```

- Finally KeyError, IndexError, ValueError or TypeError Exceptions will be raced in case we have no match (output is shortened in these examples):

  ::

```
>>> root['child',slice(1,1)] # slice delivers no match
blist([])
>>> root[{'child2'}] # invalid tag
Traceback (most recent call last):
...
KeyError: 'child2'
>>> root[100] #  Index access out of rangeroot['child',100] # family␣
↪index out of range
Traceback (most recent call last):
...
IndexError: Given abs-idx in target 100 is out of range
>>> root[('child',100,1)] # Invalid family tag
Traceback (most recent call last):
...
ValueError: Given target ('child', 100, 1) is invalid
>>> root[lambda i: i.value>2] # invalid calculation for child with␣
↪value 'tag conflict'
Traceback (most recent call last):
...
TypeError: lambda: raised an exception in filter-calculation, the 6.␣
↪child iTree(('child', 1), value='tag conflict') is incompatible␣
↪with the calculation
```

2.5.2 In-depth Item Access

In general all get methods can be used for in-depth access too (The only exception is the *__getitem__()*-method that targets first level only).

In the get-methods the levels are addressed by multiple parameters:

get(target_level1, target_level2, . . . ,target_leveln).

To check the in-depth access we append our example with an item in level2 of the tree:

```
>>> root[0].append(iTree('sub_child',value=0)) # prepare one level deeper item
iTree('sub_child', value=0)
```

For sure the deeper levels can be accessed via multiple *__getitem__()* too. But in case of multiple matches the results can be very confusing.

Imagine in the first level you target a tag-family with multiple items the second index targets in this case the items in the delivered level1 list only and does not dive in the tree as the user might expect:

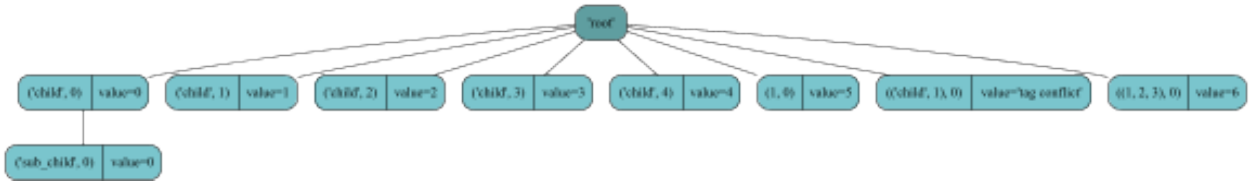


Fig. 6: Figure shows the tree with additional level in first item

```
>>> root[0][0] # access nested (deeper) items
iTree('sub_child', value=0)
>>> root['child'][0] # If the result of first operation is not a single item this␣
↪will deliver the first item in the result-list
iTree('child', value=0, subtree=[iTree('sub_child', value=0)])
>>> # See that the result is in the first and not in the second level of the iTree!!
```

To avoid such failures it's recommended to use the more advanced in-depth get-methods. E.g: usage of *get()*:

```
>>> root.get(0,0)
iTree('sub_child', value=0)
>>> root.get(0,('sub_child',0))  # access nested (deeper) items via target-path-list␣
↪(mixed target types)
iTree('sub_child', value=0)
>>> target_path=[0,0]
>>> root.get(*target_path) # targets deep
iTree('sub_child', value=0)
>>> root.get(*[0,0]) # targets deep -> single item arguments given will deliver␣
↪single item only
iTree('sub_child', value=0)
>>> # be CAREFUL because:
>>> root.get(*[0,0]) # gives empty list because target single item has no subtree␣
↪(type cast to list)
iTree('sub_child', value=0)
>>> root.get(target_path) #target first level only (absolute index-list given)
[iTree('child', value=0, subtree=[iTree('sub_child', value=0)]), iTree('child',␣
↪value=0, subtree=[iTree('sub_child', value=0)])]
>>> root.get([0,0]) #target first level only (absolute index-list given)
[iTree('child', value=0, subtree=[iTree('sub_child', value=0)]), iTree('child',␣
↪value=0, subtree=[iTree('sub_child', value=0)])]
```

The functionality of *get()* is to handle multiple results in higher levels and combine them in an internal iterator. The result is at the end a flattended list that considers all findings in the **final** target level from all branches that were matching.

In case one level only is given the method behaves like *__getitem__()* except that in case of issues a default might be returned (if defined as named parameter).

The method *get.single()* enforces the delivery of unique items. The user can be sure that just a single item will be delivered. In case of multi-target parameters given the method analysis the result and shrink a list with a unique element to the element itself. If the list contains more items this is handled as no match and a ValueError will be raised (or default value will be delivered if defined).

The method *get.iter()* delivers always an iterator over the items targeted. In case of unique findings it delivers a list *[unique_item]* that is iterable and can be easy identified by a type check.

For the in-depth get-methods a level filter functionality is available. The user can define level filters by giving filtering methods for the different levels (see *level-filtering*).

```
>>> root.get(lambda i: i.value==0,lambda i: i.value==0) # level filtering
[iTree('sub_child', value=0)]
```

2.6 Comparing iTrees

In case *iTree*-items should be compared the difference in between the == operator and the *is* keyword should be understood. An *iTree* object is equal (==) if the following statement delivers *True* :

```
>>> itree.tag and itree.data and all(sub_i==sub_o for sub_i,sub_o in zip(itree,other))
True
```

To check if the item is really the same (instance) the user must use *is*.

itertree.iTree.**__eq__**()
compares if the tag, value and children content of another item matches with this item

---

Note: If you like to check if it is really the same object you should use ´is´ instead of ´==´ operator

---

Parameters **other** – other iTree

Returns boolean match result (True match/False no match)

itertree.iTree.**equal**()
compares if the data content of another item matches with this item

Parameters

- **other** – other iTree
- **check_coupled** – check the couple object too? (Default False)
- **check_flags** – check the flags of the objects? (Default False)

Returns boolean match result (True match/False no match)

itertree.iTree.**__hash__**()
The hash operation is available

Returns integer hash

The explicit *equal()* method allows the check of additional properties (e.g. flags or the *itree.coupled_object*), which are not considered in the normal *__eq__()* method.

The difference inbetween == and *is* is also important in case of the *in* operation where the operation == is used. Same for the *index()* and *deep.index()* method. The *index()* function behaves here like in lists and the start parameter can be used to target multiple searches.

To get the index of a specific item it is recommended just to use the *iTree* property *itree.idx* or *itree.idx_path* which delivers the absolute index/index-path of the specific item directly.

property iTree.**idx**
Index of this object in the iTree (related to the absolute order)

Method is very important for internal functionalities

Note: In general the item index is cached but in case of deleted items or reorder operations the cache might be outdated. In this case the index update based on a search might take longer.

Return type Union[int, None]

Returns unsigned integer representing the index (related to absolute order of iTree)

property iTree.**idx_path**
delivers a list of absolute indexes from the root to this item

For items with no parent (root_item) an empty tuple will be delivered

Note: We deliver here a tuple because it might be helpful if the object is hashable (usage as a dict key)

Return type tuple

Returns tuple of index integers (here we do not deliver an iterator!)

Methods checking if a item is a child of the *iTree*-object:

itertree.iTree.**__contains__**()
Checks if an ´iTree´ object is part of the ´iTree´ for comparison == -> ´__eq__()´ is used. For finding a specific object use ´is_parent()´ or ‘is_in()` instead.

In case no ´iTree´ object is given the function uses ´__getitem__´ to check if matching item(s) exists.

Note: There is no coresponding in-depth function available the user can easy search via: >>> # Let itree be the iTree object the target should be searched in >>> any(tag == i.tag for i in itree.deep) >>> any(searchkey == i[0][-1] for i in itree.deep.tag_idx_paths()) >>> s=len(index_list) >>> any(len(i[0])>s and index_list == i[0][(-s+1):] for i in itree.deep.idx_paths())

Parameters **target** – iTree object searched for or a target used by ´__getitem__()´ method

Returns

- True - matching child is found
- False - no matching item found

itertree.iTree.deep.**__contains__**()

coded in helper-class:

itertree.itree_indepth._iTreeIndepthTree.**__contains__**()
Call via **x in iTree().deep**

Checks if given ´iTree´ is child or sub-child of the ´iTree´ (inside). For comparison == -> ´__eq__()´ is used. For finding the exact object instance use ´is_in()´ instead.

Parameters **item** (iTree) – iTree object to be searched for

Return type bool

Returns

- True - matching child is found

- False - no matching item found

itertree.iTree.**is_in**()

Checks if the given object is child of the iTree. Different to ´__contains__()´ we check here for the instance (specific) object (is) and not based on ´__eq__()´.

Parameters **item** – iTree object to be searched for

Returns

- True - matching child is found
- False - no matching item found

itertree.iTree.deep.**is_in**()

coded in helper-class:

itertree.itree_indepth._iTreeIndepthTree.**is_in**()

Call via **iTree().deep.is_in()**

Checks if the given object is in thee iTree. Different to ´__contains__()´ we check here for the instance (specific) object (*is*) and not based on ´__eq__()´.

Parameters **item** (iTree) – iTree object to be searched for

Return type bool

Returns

- True - matching child is found
- False - no matching item found

itertree.iTree.**index**()

The index method allows to search for the absolute index of a matching item in the *iTree*. The item must be a iTree object and the index will deliver the first match. The comparison is made via == operator.

If item is not found a IndexError will be raised

Note: To get the index of a specific item instance the *.idx-* property should be used.

Parameters

- **item** (iTree) – iTree object to be searched for
- **start** (*Union[iTree,target_path]*) – iTree item or start target_path where index search should be started (start item is included in search)
- **stop** (*Union[iTree,target_path]*) – iTree item or stop target_path where index search should be stopped (stop item is not included in search)

;rtype: int :return: absolute index of the found item

itertree.iTree.deep.**index**()

coded in helper-class:

itertree.itree_indepth._iTreeIndepthTree.**index**()

Call via **iTree().deep.index()**

The index method allows to search for the index_path of a matching item in the *iTree*. The item must be a iTree object and the index will deliver the first match. The comparison is made via == operator.

> **Warning:** If the user gives the *start* or *stop* argument not as an *iTree*-item but as a *target_path* he must give a list (or iterable) for targeting each level in the tree! The arguments are interpreted as the arguments for *iTree.get()*.
>
> This means if the user targets an element in first level by an absolute index he must give it as *index(item,[index])* giving just the integer value will not work in this case!

If item is not found a IndexError will be raised

Note: To get the index of a specific item instance in his parent tree the *.idx_path-* property should be used.

Parameters

- **item** (*iTree*) – iTree object to be searched for
- **start** (*Union[iTree,target_path]*) – iTree item or start target_path where index search should be started (start item is included in search)
- **stop** (*Union[iTree,target_path]*) – iTree item or stop target_path where index search should be stopped (stop item is not included in search)

;rtype: list :return: index_path of the found item

itertree.iTree.**count**()
Counts how many equal (==) children are in the *iTree*-object.

Parameters **item** (*iTree*) – The *iTree*-items will be compared with this item

Return type int

Returns Number of matching items found

itertree.iTree.deep.**count**()

coded in helper-class:

itertree.itree_indepth._iTreeIndepthTree.**count**()
Call via **iTree().deep.count()**`

Counts (in-depth) how many equal (==) items are inside the *iTree*-object.

Parameters **item** (*iTree*) – The *iTree*-items will be compared with this item

Return type int

Returns Number of matching items found

itertree.iTree.**is_tag_in**()
Checks if a iTree contains the given family-tag (first-level only) :param tag: family tag :return: True/False

itertree.iTree.deep.**is_tag_in**()

coded in helper-class:

itertree.itree_indepth._iTreeIndepthTree.**is_tag_in**()
Call via **iTree().deep.is_tag_in()**

Checks if a iTree contains the given family-tag (in_depth (all levels)) :param tag: family tag :return: True/False

iTree`s can also be compared with each other the criteria here is the size `__len__() of the objects. Based on this comparison operators < ; <= ; > ; >= are available. The methods exists in the level 1 children related variant (base-class) or in in-depth variant (use *deep*-sub-class).

For length calculations the following methods exists:

itertree.iTree.**__len__**()

itertree.iTree.deep.**__len__**()

coded in helper-class:

itertree.itree_indepth._iTreeIndepthTree.**__len__**()
Call via **len(iTree().deep)**

Delivers number of all items (in-depth) inside the *iTree*-object

Return type int

Returns number of children and sub-children in *iTree*-object

itertree.iTree.**filtered_len**()
Calculates the number of filtered children.

Parameters **filter_method** (*Callable*) – filter method that checks for matching items and delivers *True*/*False*. The filter_method targets always the *iTree*-child-object and checks a characteristic of this object for matches (see *filter_method*)

Return type int

Returns Number of matching items found

itertree.iTree.deep.**filtered_len**()

coded in helper-class:

itertree.itree_indepth._iTreeIndepthTree.**filtered_len**()
Call via **iTree().deep.filtered_len()**`

Calculates in-depth the number of filtered items.

Parameters

- **filter_method** (*Union[Callable, None]*) – filter method that checks for matching items and delivers *True*/*False*. The filter_method targets always the *iTree*-child-object and checks a characteristic of this object for matches (see *filter_method*)
- **hierarchical** (*bool*) –
  - **True - hierarchical filtering if a parent does not match to the filter** the children are taken out too, and they are not considered
  - **False - non-hierarchical filtering (all items are checked against the** filter and considered in the result)

Return type int

Returns Number of matching items found

2.7 iTree properties

As we will see later on some properties of the *iTree* object can be modified by the related methods.

The *iTree* object contains the following general properties:

property iTree.**root**
property delivers the root-item of the tree

In case the item has no parent it will deliver itself

Return type *iTree*

Returns iTree root item

property iTree.**is_root**
Is this item a root-item (has no parent)?

Return type bool

Returns

- *True* - is root
- *False* - is not root

property iTree.**parent**
Property delivers current items parent-object.

Return type Union[*iTree*, None]

Returns iTree parent-object or None (in case no parent exists)

property iTree.**pre_item**
Delivers the pre-item (predecessor) of this object in the parent-tree. If self is first item or there is no parent *None* will be delivered.

Return type Union[*iTree*,None]

Returns iTree predecessor or None (no match)

property iTree.**post_item**
Delivers the post-item (successor) of this object in the parent-tree. If self is first item or there is no parent *None* will be delivered.

Return type Union[*iTree*,None]

Returns *iTree* successor or *None* (no match)

property iTree.**level**
Delivers the distance (number of levels) to the root-item of the tree. Or in other words how deep in tree the item is positioned. In case item has no parent (is a root-item) this method will deliver 0.

Return type int

Returns integer - number of levels (outer direction)

property iTree.**max_depth**
Relative from this item the method measures the maximum depth of the tree and delivers the maximum number of levels that are found in this object.

If the user wants to now the maximum depth of the whole tree ensure that the property of the root-item is read. The user might use *my_tree.root.max_depth* to ensure this.

Return type int

Returns integer maximal number of levels that exists in the tree (inner direction)

property iTree.**is_tree_read_only**
Is the tree protection flag set? In this case the tree structure cannot be changed

This property targets the tree structure not the value!

Return type bool

Returns

- False - subtree can be changed (writeable)
- True - subtree is protected (read-only)

property iTree.**is_value_read_only**
Is iTree value read_only? Is the value protection flag iTFLAG.READ_ONLY_VALUE is set?

Return type bool

Returns True - read-only protection of value active False - value is writeable

property iTree.**is_linked**
In contrast to iTreeLinked class this is False

Return type bool

Returns True/False

property iTree.**is_link_root**
property that marks the iTree item as an item that contains a link

Returns

- True - is a link root item
- False is no iTree link item

property iTree.**is_link_cover**
If the item is local and covers a linked item the property is True

Return type bool

Returns True/False

property iTree.**is_placeholder**
Property shows that item is a placeholder class

Normally there should be no placeholder class in the iTree but in case a loaded link does no more contain the expected items it might happen that such a class artifact is still in the tree. In placeholders the value contains the family index in the linked class.

Return type bool

Returns True/False

Item identification properties:

property iTree.**idx**
Index of this object in the iTree (related to the absolute order)

Method is very important for internal functionalities

Note: In general the item index is cached but in case of deleted items or reorder operations the cache might be outdated. In this case the index update based on a search might take longer.

Return type Union[int, None]

Returns unsigned integer representing the index (related to absolute order of iTree)

property iTree.**tag_idx**
The tag_idx is a unique identification of the item. It is represented by a tuple containing the family-tag and the family related index of the item.

If the item is not part of a parent-tree (root-item) in this case the result will be *None*.

Return type Union[tuple, None]

Returns tuple (family-tag, family-index) or None (if item has no parent)

property iTree.**idx_path**
delivers a list of absolute indexes from the root to this item

For items with no parent (root_item) an empty tuple will be delivered

Note: We deliver here a tuple because it might be helpful if the object is hashable (usage as a dict key)

Return type tuple

Returns tuple of index integers (here we do not deliver an iterator!)

property iTree.**tag_idx_path**
The path is a tuple of tag_idx tuples from root to this item. Each tag_idx is a tuple containing the pair family-tag and family-index.

For items with no parent (rooot_item) an empty tuple will be delivered

Note: We deliver here a tuple because it might be helpful if the object is hashable (usage as a dict key)

Return type tuple

Returns tuple of key tuples containing family-tag and family-index

The following examples shows how some of the *iTree*-properties are read out.

```
>>> root = iTree('root', subtree=[iTree('child', 0), iTree((1, 2), 'tuple_child0'),␣
↪iTree('child', 1), iTree('child', 2),iTree((1, 2), 'tuple_child1')])
>>> root[0] += iTree('subchild')
>>> root.render()
iTree('root')
 > iTree('child', value=0)
 .  > iTree('subchild')
 > iTree((1, 2), value='tuple_child0')
 > iTree('child', value=1)
 > iTree('child', value=2)
 > iTree((1, 2), value='tuple_child1')
>>> root[0][0].root
iTree('root', subtree=[iTree('child', value=0, subtree=[iTree('subchild')]),...,
↪iTree((1, 2), value='tuple_child1')])
>>> root[0][0].idx
0
>>> root[0][0].tag_idx
('subchild', 0)
```

(continues on next page)

(continued from previous page)

```
>>> root[0][0].idx_path
(0, 0)
>>> root[0][0].tag_idx_path
(('child', 0), ('subchild', 0))
>>> root[1].value
tuple_child0
>>> root[1].tag_idx
((1, 2), 0)
>>> root[-1].value
tuple_child1
>>> root[-1].tag_idx
((1, 2), 1)
>>> len(root) #  level 1 only
5
>>> len(root.deep) # all in-depth items
6
>>> root2=root.copy()
>>> root2[-1].append(iTree('subitem')) # we append one item in depth
iTree('subitem')
>>> root2>root # level 1 only size-compare
False
>>> root2.deep>root.deep # all items size-compare
True
```

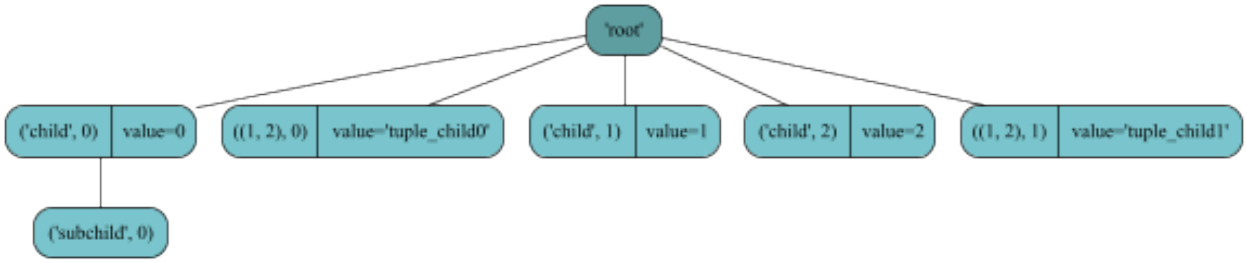


Fig. 7: Figure showing iTree used in example

As shown in the last example hashable objects can be used as tags for the itertree items to be stored in the *iTree* object. Even for those kind of tag objects it is possible to store multiple items with the same tag. In the example the enumeration inside the tag family can be seen in the index enumeration (tag_idx).

Beside those structural properties the *iTree* objects contains a property that can be used to "link" the ìTree`-object to another Python object.

property iTree.**coupled_object**

The *iTree*-object can be coupled with another Python-object. The pointer to the object is stored and can be reached via this property. (E.g. this can be helpful when connecting the *iTree* with a visual item (hypertree-list item) in a GUI)

Returns pointer to coupled-object or None if no object is stored

itertree.iTree.**set_coupled_object**()

Couple another Python-object with this *iTree*-object.

Compared with the *value* the coupled-object is not tracked by any internal functions. We do not consider it in any relation (e.g. *__contains__()* and do not dump it in files, etc. Even in linked items the coupled-object is not protected. And in copies it is ignored and not taken over.

Note: E.g. The coupled-object might be an object in a GUI that is related to this item.

Parameters **coupled_object** – object pointer to the object that should be coupled with this iTree item

Different than the data the coupled_obj the idea is here to have just a pointer to another Python object. The only operations considering those objects is in the link root were during reload or if a linked item is converted in a local item the couple object will be taken over. The *equal()* compare function can also target the coupled-object.

Note: Behind this objects is the following idea: E.g. The user might couple the *iTree* to a graphical user interface object. Connect it with an item in a hypertree-list. Or it can be used to couple the *iTree* object to an item in a mapping dictionary. The property coupled-object is not actively managed by the *iTree* object it's just a place to store a pointer. E.g. If *iTree* is stored in a file or standard compares this information will not be considered.

There can be cases where it is helpful to use this additional possibility to store information in the *iTree* too. E.g. in the attached calendar.example.py we use the coupled-object to store the day-name.

2.8 iTree value related methods

Compared with the previous versions (0.8.0) the handling of the data/value property is simplified a lot.

First we renamed the *data*-property to *value*-property to be compatible with the naming of items in dicts. Second we came to the conclusion that the management of the value content is not the core function of *iTree* and so we made it more independent as it was in the previous versions.

Now it is in the hand of the user if he stores a more complex object or e.g. just a simple integer value in the *iTree*-object.

The old *iData`class is still available for downward compatibility. But the object is no more placed automatically in the value of a ìTree* item. To utilize it the user must put the object manually in. As explained we do not expect anymore that the object stored in value is a dictionary like object (*iData*). We uncoupled here the functionalities.

If required we can recommend one of the data-models available in the itertree package. They can be used to store specific types of data (including checks). Other data models might be used too but the user must ensure that the external data models are serialized correctly if he wants to store the *iTree* and his data in files.

In case a *iTree* object is created without a *value* parameter the default value object will be the *NoValue* class.

These are the value related methods available in *iTree*.

property iTree.**value**
Delivers the full value object stored in the *iTree*-object

Return type object

Returns value-object of the item

itertree.iTree.**get_value**()
Delivers the value-object of the item or a sub-value in case key_index parameter is used and a matching object is stored in the *iTree* .

Note: If *iTValueModel* is stored in *iTree* the method will not target the model it will target the value inside. If the model itself is required the *value*-property of *iTree* must be used.

Except In case a key_index is given but the object is not a *dict* or a *list* like object an *AttributeError* will be raised (*__getitem__()`required). If no matching item is found an `IndexError* or *KeyError* will be raised.

Return type object

Returns value object the *iTree* or *iTValueModel* (in case a model is stored in the *iTree*)

itertree.iTree.**set_value**()

Set/replace the value content of the *iTree*-object.

The method returns the previous stored value object that was replaced by the operation.

Note: If an *iTValueModel* is stored as value in the *iTree* by default the set_value() method will target the value which is stored inside the model. If the model itself should be exchanged the user must give the new model as value parameter of this method. To replace the model with another Python object the user must first delete the model via *del_value()* command and afterwards set the new value.

Parameters **value** (*object*) – data-object that should be placed as value or in case we have a *iTValueModel* already as value it is placed inside the model.

Return type object

Returns old value object that was stored in iTree before

itertree.iTree.**del_value**()

Deletes the full value-object stored in ´iTree´ (´NoValue´ is stored in iTree).

This method will always delete the whole object stored in *iTree* even *iTValueModel*-objects are deleted. To delete the value content of a model *mytree.value.clear()* or 'set_value(NoValue)' might be used.

Returns deleted value

```
>>> my_tree = iTree('root')
>>> my_tree.set_value(1)
<class 'itertree.itree_helpers.NoValue'>
>>> repr(my_tree.get_value())
1
>>> my_tree.set_value(Data.iTInt8Model())  # store a model limiting the matching␣
↪values
1
>>> my_tree.set_value(1)  # store the value in the model
<class 'itertree.itree_helpers.NoValue'>
>>> repr(my_tree.value)  # delivers the whole object stored in value
iTInt8Model(1)
>>> repr(my_tree.get_value())  # again we take the value out of the model
1
>>> my_tree.set_value(1024)  # value out of the valid range
Traceback (most recent call last):
...
ValueError: Given value does not match to given filter_method (out of range)
>>> repr(my_tree.del_value())  # delete the model
iTInt8Model(1)
>>> my_tree.value
<class 'itertree.itree_helpers.NoValue'>
```

In case a *iTValueModel* based object is stored in the *iTree`value the methods `get_value()* and *set_value()* will not target the model itself. Furthermore the value inside the models will be read or exchanged. If the model itself should be exchanged *set_value()* can be used too the method will automatically identify that the new value is a model and the

old model will be replaced by the new one. Beside this the *del_value()* targets always the value object and replaces it with *NoValue*. Even a model will be deleted in this case. To delete the value in the model the user must use *get_value(NoValue)* or *my_tree.value.clear()*.

In addition to the normal get and set we have the key related methods for value access:

itertree.iTree.**get_key_value**()

Delivers the value-object of the item or a sub-value in case key_index parameter is used and a matching object is stored in the *iTree* .

In case the stored value is a *dict*-like object the key will be used as the key of the dict. In case the stored value is a *list*-like object the keyx will be used as the index of the list.

In case the target value is a *iTValueModel* the value inside will be targeted and not the model itself.

Note: If *iTValueModel* is stored in *iTree* the method will not target the model it will target the value inside. If the model itself is required the *value*-property of *iTree* must be used.

Except In case a key_index is given but the object is not a *dict* or *list* like object an *AttributeError* will be raised (*__getitem__()*-method required). If no matching item is found an *IndexError* or *KeyError* will be raised.

Parameters **key** (*Optional[Hashable, int]*) – Optional key or index parameter

Return type object

Returns value object the *iTree* or *iTValueModel* (in case a model is stored in the *iTree*)

itertree.iTree.**set_key_value**()

Depending on the already stored object this operation is a sub-replacement of a part only.

The method returns the previous stored value object that was replaced by the operation.

The user can influence the behavior by giving the *key* parameter. And it depends on the already stored value object (e.g. a *list* or *dict*). Only the value of the related item will be replaced or in case the item did not exist yet the might object will be extended by the given value (*dict* only).

Depending on given key parameter and the already stored object we have the following possible behaviours:

- dict stored in value -> store the value in the dict with the key given in key_index
- dict stored in value and matching item-value is a *iTValueModel* -> replace value inside the model
- list stored in value -> key_index must be an index and replace the related item in the list with the value given
- list stored in value and matching (index) item-value is a *iTValueModel* -> replace value inside the model
- key == *INF* and list stored in value -> append given value in the list

---

Note: If an *iTValueModel* is stored as value in the *iTree* by default the *mytree.set_value()*-method will target the value which is stored inside the model. If the model itself should be exchanged the user must give a new model as value parameter of this method. To replace the model with another Python object the user must first delete the model via *del mytree.value[key]* command and afterwards set the new value or he sets the value directly via *mytree.value[key]==new_value* .

Parameters

- **key** (*Optional[Hashable,int]*) – key or index of the value object (depends on the object already stored in *iTree*). if *key==INF* the value will be appended in case a list-like object is already stored in the *iTree*-object.
- **value** (*object,*) – value object that should be placed as value or in case a key is given the sub-value in the *iTree* or in case we have a *iTValueModel* is used inside the model.

Return type object

Returns old value object that was stored in iTree before

In general these methods behave like the normal counter part (model objects are handled the same way). The only difference is that these methods targeting sub_values in *dict* or *list* like objects (using *__getitem__()*). For *dict`s key is used like a key and for `list* key is used as an integer index. If the key does not exists in a *dict`like object the key-value pair will be added. For `list* an append via *INT=float('int')* as index is possible too. By default for *list* like objects no matching indexes will raise an *IndexError* exception.

2.9 iTree iterations

As the name itertree suggests we have a lot of possibilities to iterate over the items in the tree-structure. In the class the we use generators (*yield*-statement) to create the output for the iterations.

Note: The class doesn't contain a *__next__()*-method. This means if the given iteration methods are used (generators inside) the user must cast those generators for functions targeting the *__next__()* via the build-in *iter()*-statement. But most often this is not required because by most functionalities the supported *__iter__()* method is targeted.

In *iTree* we have iteration-generators which are more related to list-like functionalities and other which are targeting more in the direction of the dict-like iterators.

Most iteration-generators are available in diffrent level behavior:

1. The children only variant iterating only over the items in level 1 of the tree-structure
2. In the in-depth variant which iterates as a flatten iterator over all the nested children.

First we show the list like standard iterator which delivers the children in the main/absolute order of the *iTree*-object.

`itertree.iTree.__iter__()`

The more dict-like iteration-methods targeting the children (level 1) are:

`itertree.iTree.keys()`
Iterates over all children and deliver the children tag-idx tuple (family-tag,family_index)

Note: This is a dict like iterator that delivers the unique keys for all children.

Parameters **filter_method** (*Union[Callable,None]*) – filter method that checks the item and delivers *True*/*False*. The filter_method targets always the *iTree*-child-object and checks a characteristic of this object for matches

If *None* is given filtering is inactive.

Return type Iterator

Returns iterator over the tag-idx of the children

itertree.iTree.**values**()

Iterates over all children and deliver the children values

Parameters **filter_method** (*Union[Callable,None]*) – filter method that checks for matching items and delivers *True*/*False*. The filter_method targets always the *iTree*-child-object and checks a characteristic of this object for matches (see *filter_method*)

If *None* is given filtering is inactive.

Return type Iterator

Returns iterator over the values stored in the children

itertree.iTree.**items**()

Iterates over all children and deliver the children item-tuples (key,item) or (key,value). As key we use the unique tag-idx: (tag-family,family-index).

The function is comparable with dicts *items()* function.

Parameters

- **filter_method** (*Union[Callable,None]*) – filter method that checks for matching items and delivers *True*/*False*. The filter_method targets always the *iTree*-child-object and checks a characteristic of this object for matches (see *filter_method*)

  If *None* is given filtering is inactive.

- **values_only** (*bool*) –
  - *False* (default) - in the key,value tuple the iterator put the iTree object as value in
  - *True* - in the key,value tuple the iterator put "only" the value object of the *iTree*-object in

Return type Generator

Returns iterator over the target keys and item value of the children

To make the delivered generator-content visible we use the *list()*-cast in the following examples:

```
>>> # create a small nested iTree:
>>> root = iTree('root', subtree=[iTree('one', 1, subtree=[iTree('subone', 1.1),␣
↪iTree('subtwo', 1.2)]), iTree('two', 2), iTree('three', 3)])
>>> list(root)  # __iter__()
[iTree('one', value=1, subtree=[iTree('subone', value=1.1), iTree('subtwo', value=1.
↪2)]), iTree('two', value=2), iTree('three', value=3)]
>>> list(root)
[iTree('one', value=1, subtree=[iTree('subone', value=1.1), iTree('subtwo', value=1.
↪2)]), iTree('two', value=2), iTree('three', value=3)]
>>> list(root.values())
[1, 2, 3]
>>> list(root.tag_idxs())
Traceback (most recent call last):
...
AttributeError: 'iTree' object has no attribute 'tag_idxs'
>>> list(root.items())
[(('one', 0), iTree('one', value=1, subtree=[iTree('subone', value=1.1), iTree('subtwo
↪', value=1.2)])), (('two', 0), iTree('two', value=2)), (('three', 0), iTree('three',
↪ value=3))]
>>> list(root.items(values_only=True))
[(('one', 0), 1), (('two', 0), 2), (('three', 0), 3)]
```

We have some special iteration-methods related to the item access based on the groups created by tag-families. The delivered items are ordered by the first item (or the last - if parameter is set) in the family and the iteration runs over

all items of the first familly then all items of the next and so on.

itertree.iTree.**tags**()
iters over all family-tags in level 1 (children). The order is based on first or last item in the family.

Parameters **order_last** (*bool*) –

- False (default) - The tag-order is based on the order of the first items in the family
- True - The tag-order is based on the order of the last items in the family

Return type Iterator

Returns tag iterator

itertree.iTree.**iter_families**()
This is a special iterator that iterates over the families in *iTree*. It delivers per family the tag and a list of the containing items. The order is defined by the absolute index of the first item in each family

Method will be reached via *iTree.Families.iter()*

Parameters

- **filter_method** (*Union[Callable, None]*) – filter method that checks for matching items and delivers *True*/*False*. The filter_method targets always the *iTree*-child-object and checks a characteristic of this object for matches (see *filter_method*)

  If filter_method is None no filtering is performed

  Note: An internal filtering is available because this may change the order of the delivered items. An external filter with same method might deliver a different result!

- **order_last** (*bool*) –
  - False (default) - The tag-order is based on the order of the first items in the family
  - True - The tag-order is based on the order of the last items in the family

Return type Generator

Returns iterator over all families delivers tuples of (family-tag, family-item-list)

itertree.iTree.**iter_family_items**()
This is a special iterator that iterates over the families in *iTree*. It iters over the items of each family the ordered by the first or the last items of the families.

Parameters **order_last** (*bool*) –

- False (default) - The tag-order is based on the order of the first items in the family
- True - The tag-order is based on the order of the last items in the family

Return type Generator

Returns iterator over all families delivers tuples of (family-tag, family-item-list)

Note: The family structure inside *iTree* cannot be made available directly because this would give the user the possibility of corrupting manipulations. But the user can use those family related iteration functions if he wants to create a representation of the family structure.

Most in-depth iteration-methods have additional parameters:

- *filter_method* filter parameter which allows the hierarchical-filtering inside the iteration loops.

- *up_to_low* allows to select the direction of the iteration top->down or bottom-> up (default: *up_to_low=True*).

All the in-depth iteration-methods are reached via the helper class *iTree.deep*:

itertree.iTree.deep.**__iter__**()

coded in helper-class:

itertree.itree_indepth._iTreeIndepthTree.**__iter__**()

Call via: **iter(iTree().deep)**

In-depth generator (iterator) which iterates over all nested items of *iTree* top -> down direction

Return type Generator

Returns iterator over all ìTree`-items

itertree.iTree.deep.**iter**()

coded in helper-class:

itertree.itree_indepth._iTreeIndepthTree.**iter**()

Call via **iTree().deep.iter()**

In-depth iterator that iterates over all items in the nested *iTree*-structure. The iterator flattens the nested structure.

Via the parameters the user can achieve hierarchical filtering of items. He can change the iteration order up-> down or down->up.

If no parameter is given *iter()* behaves like the build in *__iter__()* method of the object.

Note: The given iteration order must not be seen like the build-in 'reversed()' function which changes the iteration direction in general! Furthermore, it means we iterate:

- *up_to_low==True*: parent-> child-> sub-child-> sub-sub-child-> ...

or we start from the most-inner nested item:

- *up_to_low==False*: item, parent, parent-parent, ..., -> root

But we always start in the right order we have in *iTree* first the root or in second case first most-inner nested item coming from the root.

Parameters

- **filter_method** (*Union[Callable, None]*) – filter method that checks for matching items and delivers *True*/*False*. The *filter_method* targets always the *iTree*-child-object and checks a characteristic of this object.

  If *None* is given no filtering will be performed.

- **up_to_low** (*bool*) –
  - True (default) - we iterate in-depth from up to the lower inner structure of the *iTree*-object
  - False - we iterate in-depth from lower to upper structure of the *iTree*-object

Return type Generator

Returns iterator over all nested ìTree`-items

As explained we can iter in two directions up-> low (default) or low->up (set parameter *up_to_low=False*):

```
>>> root = iTree('root')
>>> for i in range(2):
    item=root.append(iTree('%i'%i, i))
    for ii in range(2):
        subitem = item.append(iTree('%i_%i' % (i,ii), i*10+ii))
        for iii in range(2):
            subitem.append(iTree('%i_%i_%i' % (i, ii,iii), i * 100 + ii*10+iii))
>>> [i for i in root.deep.iter(up_to_low=True)][0:5] # show just a part
[iTree('0', value=0, subtree=[iTree('0_0', value=0, subtree=[iTree('0_0_0', value=0),␣
↪iTree('0_0_1', value=1)]), iTree('0_1', value=1, subtree=[iTree('0_1_0', value=10),␣
↪iTree('0_1_1', value=11)])]), iTree('0_0', value=0, subtree=[iTree('0_0_0',␣
↪value=0), iTree('0_0_1', value=1)]), iTree('0_0_0', value=0), iTree('0_0_1',␣
↪value=1), iTree('0_1', value=1, subtree=[iTree('0_1_0', value=10), iTree('0_1_1',␣
↪value=11)])]
>>> [i for i in root.deep.iter(up_to_low=False)][0:5] # show just a part
[iTree('0_0_0', value=0), iTree('0_0_1', value=1), iTree('0_0', value=0,␣
↪subtree=[iTree('0_0_0', value=0), iTree('0_0_1', value=1)]), iTree('0_1_0',␣
↪value=10), iTree('0_1_1', value=11)]
```

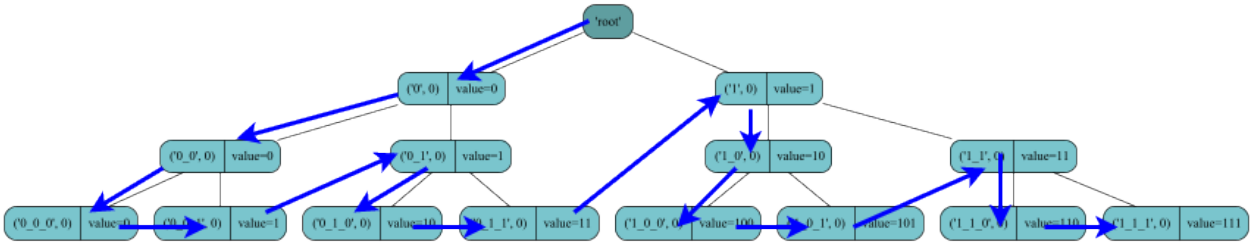


Fig. 8: Figure schema for up->down (default) iteration

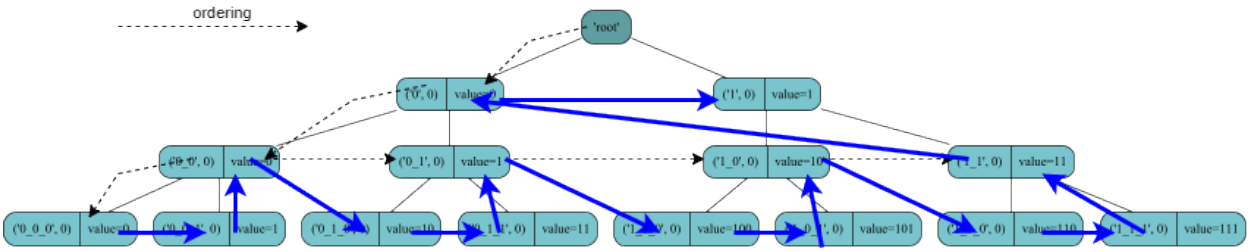


Fig. 9: Figure schema for down->up iteration

Additional we have the in-depth iteration-methods:

`itertree.iTree.deep.`**`idx_paths`**`()`

coded in helper-class:

`itertree.itree_indepth._iTreeIndepthTree.`**`idx_paths`**`()`

Call via **iTree().deep.idx_paths()**

In-depth generator (iterator) which iterates over all nested items of the *iTree*-object in top -> down direction. The iterator delivers per item the pair (relative idx_path, item).

The index path is same as in the items *.idx_path* property which contains the absolute indexes to the root-parent. But in this iterator we deliver the relative idx_path related to the element the iteration is started and not the path to the root-parent.

The iterator does exactly the same as the following code based on the main iterator and the extraction of the idx_paths:

```
>>> # Let itree be the instanced iTree in which we like to iterate over all␣
↪nested items (in-depth-iteration)
>>> s=len(itree.idx_path) # required to create relative paths
>>> idx_paths_generator=((i.idx_path[s:],i) for i in iter(itree.all))
```

But this specific iterator is much quicker because the indexes are counted up internally during the iteration which is more efficent as the calculation of the idx_path for each item in this solution.

The solution to deliver the pairs is chosen, because the user can choose by unpacking what's required for his needs and he still can filter based on item properties.

E.g.: Store the ind_paths in a list:

```
>>> my_idx_path_list=[idx_path for idx_path,_ in itree.all.idx_paths()]
```

Store the filtered idx_paths in a list (because of the delivered items a filtering is possible):

```
>>> my_idx_path_list=[idx_path for idx_path,_ in filter(lambda i: i[1].tag=='mytag
↪', itree.all.idx_paths())]
```

Convert the content of the *iTree* in a dict by using the idx_paths as keys:

```
>>> my_dict={idx_path:item for idx_path,item in itree.all.idx_paths()}
```

The user may store values only in the dict too:

```
>>> my_dict={idx_path:item.value for idx_path,item in itree.all.idx_paths()}
```

Parameters

- **filter_method** (*Union[Callable, None]*) – filter method that checks for matching items and delivers *True*/*False*. The *filter_method* targets always the *iTree*-child-object and checks a characteristic of this object.

  If *None* is given no filtering will be performed.

- **up_to_low** (*bool*) –
  - True (default) - we iterate in-depth from up to the lower inner structure of the *iTree*-object
  - False - we iterate in-depth from lower to upper structure of the *iTree*-object

Return type Generator

Returns iterator over all ìTree`-items and yields for each item the pair (relative idx_path, item)

itertree.iTree.deep.**tag_idx_paths**()

coded in helper-class:

itertree.itree_indepth._iTreeIndepthTree.**tag_idx_paths**()

Call via: **iTree().deep.tag_idx_paths()**

In-depth generator (iterator) which iterates over all nested items of the *iTree*-object in top -> down direction. The iterator delivers per item the pair (relative idx_path, item).

The index path is same as in the items *.key_path* property which contains the absolute indexes to the root-parent. But in this iterator we deliver the relative idx_path related to the element the iteration is started and not the path to the root-parent.

The iterator does exactly the same as the following code based on the main iterator and the extraction of the key_paths:

```
>>> # Let itree be the instanced iTree in which we like to iterate over all␣
↪nested items (in-depth-iteration)
>>> s=len(itree.tag_idx_path) # required to create relative paths
>>> key_paths_generator=((i.tag_idx_path[s:],i) for i in iter(itree.all))
```

But this specific iterator is much quicker because the family-indexes are counted up internally during the iteration which is more efficent as the calculation of the key_path for each item in this solution.

The solution to deliver the pairs is chosen, because the user can choose by unpacking what's required for his needs and he still can filter based on item properties (see similar examples in method *idx_paths()*).

Parameters

- **filter_method** (*Union[Callable,None]*) – filter method that checks for matching items and delivers *True*/*False*. The *filter_method* targets always the *iTree*-child-object and checks a characteristic of this object.

  If *None* is given no filtering will be performed.

- **up_to_low** (*bool*) –
  - True (default) - we iterate in-depth from up to the lower inner structure of the *iTree*-object
  - False - we iterate in-depth from lower to upper structure of the *iTree*-object

Return type Generator

Returns iterator over all ìTree`-items and yields for each item the pair (relative idx_path, item)

itertree.iTree.deep.**iter_family_items**()

coded in helper-class:

itertree.itree_indepth._iTreeIndepthTree.**iter_family_items**()

Call via: **iTree().deep.iter_family_items()**

This is a special iterator that iterates over the families in *iTree*. It iters over the items of each family the ordered by the first or the last items of the families.

Note: As an exception this in-depth iteration-method does not support level-filtering because in an iteration based on tag-family items we do not see any sense in hierarchical filtering. Only external filtering of the resulting elements makes sense.

Parameters **order_last** (*bool*) –

- False (default) - The tag-order is based on the order of the first items in the family
- True - The tag-order is based on the order of the last items in the family

Return type Generator

Returns iterator over all families delivers tuples of (family-tag, family-item-list)

Related to tag_family sorted iterations we have in-depth only the *ìter_family_items()* mathod available.

In the following example we create based on the in-depth generators lists and dicts:

```
>>> # deep iterators:
>>> list(root.deep)  # deep counterpart of level1 __iter__() iterator
[iTree('one', value=1, subtree=[iTree('subone', value=1.1), iTree('subtwo', value=1.
↪2)]), iTree('subone', value=1.1), iTree('subtwo', value=1.2), iTree('two', value=2),
↪ iTree('three', value=3)]
>>> list(root.deep.iter(up_to_low=False))  # changed iteration order bottom-> up
[iTree('subone', value=1.1), iTree('subtwo', value=1.2), iTree('one', value=1,␣
↪subtree=[iTree('subone', value=1.1), iTree('subtwo', value=1.2)]), iTree('two',␣
↪value=2), iTree('three', value=3)]
>>> list(root.deep.tag_idx_paths()) # deep counterpart of level1 items() iterator
[((('one', 0),), iTree('one', value=1, subtree=[iTree('subone', value=1.1), iTree(
↪'subtwo', value=1.2)])), ((('one', 0), ('subone', 0)), iTree('subone', value=1.1)),␣
↪((('one', 0), ('subtwo', 0)), iTree('subtwo', value=1.2)), ((('two', 0),), iTree(
↪'two', value=2)), ((('three', 0),), iTree('three', value=3))]
>>> [(k,i.value) for k,i in root.deep.tag_idx_paths()]  # deep counterpart of level1␣
↪items(values_only=True) iterator
[((('one', 0),), 1), ((('one', 0), ('subone', 0)), 1.1), ((('one', 0), ('subtwo', 0)),
↪ 1.2), ((('two', 0),), 2), ((('three', 0),), 3)]
>>> [k for k,_ in root.deep.tag_idx_paths()]  # deep counterpart level1 to keys()␣
↪iterator
[(('one', 0),), (('one', 0), ('subone', 0)), (('one', 0), ('subtwo', 0)), (('two', 0),
↪), (('three', 0),)]
>>> [k for k,_ in root.deep.idx_paths()]  # no level 1 counterpart (lists are␣
↪automatically indexed 0->n)
[(0,), (0, 0), (0, 1), (1,), (2,)]
```

2.10 iTree Filter Queries

A lot of the in-depth methods contain the parameter *filter_method* that can be used for hierarchical inside filtering of *iTree*-items. For non-hierarchical filtering the user can use the build-in *filter()*-method. In case an outside filtering is not possible (*filter()* cannot be used) the methods have an additional parameter *hierarchical* to switch in between the two ways of filtering.

As *filter_method* the user can give a callable object that analysis the given item and calculates if the item matches to the specific criteria and deliver a True/False (match/no match) for the item.

The *iTree*-class contains no more the old *find()`and `find_all()* methods because all searches can be realized easier and more clear via the *filter_method*-parameter.

Also we do not have any more a special ´iTFilter`-class, we decided that normal filtering via filtering methods is more practicable. As a help for the user we still provide some filter classes/methods under *itertree.itree_filters* that might help related to the filtering of *iTree* specifics.

```
>>> root = iTree('root', subtree=[iTree('one', 1, subtree=[iTree('subone', 1.1),␣
↪iTree('subtwo', 1.2)]), iTree('two', 2), iTree('three', 3)])
>>> filter1 = lambda i: 'one' not in i.tag
>>> list(root.deep.tag_idx_paths(filter1))
[((('two', 0),), iTree('two', value=2)), ((('three', 0),), iTree('three', value=3))]
>>> # the hierarchical filter did not consider the item iTree('subtwo',1.2) because␣
↪parent is filtered out
>>> list(filter(lambda i: 'one' not in i[1].tag, root.deep.tag_idx_paths())) # for␣
↪non-hierachical filtering use build-in
[((('one', 0), ('subtwo', 0)), iTree('subtwo', value=1.2)), ((('two', 0),), iTree('two
↪', value=2)), ((('three', 0),), iTree('three', value=3))]
```

(continues on next page)

(continued from previous page)

```
>>> # now the sub-items are considered even that parent did not match
```

A very special filtering can be realized in the *get()*-method by putting filters in the related levels of a target_path (level filter).

E.g.:

```
root.get(Filters.is_item_tag('mytag'),Filters.is_item_tag('mytag2'))
```

will filter in first level for all items with the tag 'mytag' and in next level for all items with the tag 'mytag2'.

The filter is used only at the specific level (in side one level we can just filter) but in the next level only the findings of first level will be considered. Therefore the level filtering is a hierarchical filtering which means only the matching items of the previous level are considered in the next level..

```
>>> # based on the root object we had in last example
>>> filter_a = lambda i: 'one' in i.tag  # This will filter for the first two elements
>>> filter_b = lambda i: i.value == 1.2  # First element doesn't have this level (no␣
↪match)
>>> root.get(*[filter_a, filter_b]) # level filtering level=0~filter_a; level=1~
↪filter_b
[iTree('subtwo', value=1.2)]
```

The filtering in *iTree* is very effective and quick. As an example one might execute the example script itree_usage_example1.py or calendar_example.py. It's recommended that the user uses iterator related functions to reach the expected results (e.g. see itertools package).

2.11 iTree full overview over the in-depth functionalities

We already talked about some of the features in the in previous chapters (access and iterators) but now we like to give a full overview about in-depth related functionalities.

All related methods are available in a specific *iTree*-object via the subclass *itree.deep*.

2.12 iTree formatted output and storage

The *iTree*-object can be printed out via classical *repr()* or *str()* method, the second method delivers a shorten representation of the subtree.

itertree.iTree.**__repr__**()
: Create representation string from which the object can be theoretically be reconstructed via *eval()* (might not work in case of value-objects that do not have a working *__repr()* method)

  Return type str

  Returns representation string

itertree.iTree.**__str__**()
: String repr of the item stripping the subtree to the first and last element only and giving ".." inbetween

  For full representation-string use *repr()*.

  Returns shorten representation string

A formatted multi-line tree output is available too. If the parameter *enumerate* is set the items in the printed tree are also enumerated by the absolute index.

itertree.iTree.**renders**()
render the iTree into a string

Parameters

- **filter_method** (*Union[Callable,None]*) – filter method that checks for matching items and delivers *True*/*False*. The filter_method targets always the *iTree*-child-object and checks a characteristic of this object for matches (see *filter_method*)

  If *None* is given filtering is inactive.

  The method uses the given filter always as an hierachical filter.

- **enumerate** (*bool*) –
  - True - Add an enumeration before the items
  - False (default) - Output without enumeration
- **renderer** (*class*) – Give another renderer class for different formatting

Return type str

Returns Tree representation as string

itertree.iTree.**render**()
Print the rendered string of the *iTree*-object to the console (stdout).

Parameters

- **filter_method** (*Union[Callable,None]*) – filter method that checks for matching items and delivers *True*/*False*. The filter_method targets always the *iTree*-child-object and checks a characteristic of this object for matches. If *None* is given filtering is inactive.
- **enumerate** – add an enumeration before the rendered items
- **renderer** – Render to be used (The given render is stored and will be used until another renderer is given).

Returns

(The renderer in Version 1.0.0 was improved and uses now ascii-only characters and delivers a smaller footprint).

For full serialization of the *iTree*-objects it's recommended to use the internal *dumps()* method. If the internal methods are used (file storage is possible too) the result is represented and stored as a JSON artifact.

itertree.iTree.**dumps**()
serializes the iTree object to JSON (default serializer)

Parameters

- **calc_hash** – Tell if the hash should be calculated and stored in the header of string
- **itree_serializer** – optional user defined serializer for iTree objects

Returns serialized string (JSON in case of default serializer)

itertree.iTree.**loads**()
create an iTree object by loading from a string

If not overloaded or reinitialized the iTree Standard Serializer will be used. In this case we expect a matching JSON representation.

Parameters

- **data_str** – source string that contains the iTree information

- **check_hash** – True the hash of the file will be checked and the loading will be stopped if it doesn't match False - do not check the iTree hash
- **load_links** – True - linked iTree objects will be loaded
- **itree_serializer** – optional user defined serializer for iTree objects

Returns iTree object loaded from file

itertree.iTree.**dump**()
serializes the iTree object to JSON (default serializer) and store it in a file

Parameters

- **target_path** – target path of the file where the iTree should be stored in
- **pack** – True - data will be packed via gzip before storage
- **calc_hash** – True - create the hash information of iTree and store it in the header
- **overwrite** – True - overwrite an existing file
- **itree_serializer** – optional user defined serializer for iTree obbjects

Returns True if file is stored successful

itertree.iTree.**load**()
create an iTree object by loading from a file

If not overloaded or reinitialized the iTree Standard Serializer will be used. In this case we expect a matching JSON representation.

Parameters

- **file_path** – file path to the file that contains the iTree information
- **check_hash** – True the hash of the file will be checked and the loading will be stopped if it doesn't match False - do not check the iTree hash
- **load_links** – True - linked iTree objects will be loaded
- **itree_serializer** – optional user defined serializer for iTree objects

Returns iTree object loaded from file

In the methods the serializer can be set and might be replaced by the users own serializing format.

The serializer for Version 1.0.0 is modified and the output format is not compatible with the old format version 1.1.1. New format can be created quicker and it has no more issues with recursion depth exceptions. The conversion of old files can be made via the helper script:

```
>>> from itertree.itree_serializer.itree_json_converter import Converter_1_1_1_to_2_0_
↪0
>>> new_itree=Converter_1_1_1_to_2_0_0(old_source_file_path)
```

The new storage format was required because in Version 1.0.0 we now have only one iTree class that uses the flags parameter to be switched to read-only where we used a special class in the old implementation.

But beside this we wanted to have a better performance related to the serializing of the objects. We think that the readability is improved too. Even that this was not the main target. The new format is also 100% JSON compatible and can be read in by any JSON parser.

The output looks like this:

```
[
{
  "TYPE": "itertree.iTree",
  "VERSION": "2.0.0"
  "HASH": "e7891f95dd2f2c85d4383a8772a317e11363c495dc65a278c821836846d06471",
},
[
[0,0,["root",0],[0,8]],
  [1,0,["0",0],[0,8]],
    [2,0,["0_0",0],[0,8]],
      [3,0,["0_1",0],[0,8]],
        [4,0,["0_2",0],[0,8]],
          [5,0,["0_3",0],[0,8]],
]]
```

After the well readable header the user can see that the tree is stored in a flat list structure (which avoids RecursionError exceptions in the JSON parsers).

The formatting of the output is created in a way that each iTree item has its own row and the indentation-level gives the hint about the level in the tree. Each item is coded in JSON in the following way:

[level,family-idx,[tag-value,type-code],[value-value,type-code]]

In case the item has additional parameters they are coded like the tag and the value too. The family-index is only given for better readability of the files, it's not used during the reconstruction of the object.

We have also a dot generator available which may help to create a graphical representation of the tree but this is not deeply tested there might be limits and we cannot ensure that the shown order is always correct.

Related to serialization we like to remark that *iTree*-objects can be pickled (*pickle(my_tree)*).

2.13 iTree linked sub-trees

The *iTree* objects can be merged to one main tree from different source files by using the *link* parameter. The result is a merged *iTree* that contains all the linked subtrees. Beside the linking from different files links inside a *iTree* structure (internal links) can be defined too.

The value of the *link* parameter of the *iTree*-class must be an *iTLink`object which defines the `file_path* and the *target_path*. The parameters are dependent. For links inside the same *iTree* the *file_path* must be set to *None*. For links targeting the root of a file the *target_path* parameter must be set to *None*. The *target_path* must target a unique item in the source-tree!

Additionally the user can manipulate the linked items by making them local (covering) or by appending local items. The functionalities given here are limited to operations that do not imply a reordering of the items in the tree. The reason for this is that the linked items cannot be reordered furthermore they gave the tree a fixed, static structure. E.g. mainly we have *append()* and *make_local()* functions and we cannot *appendleft()* or *insert()* because this would mean we have to reorder the other items. A change of a linked structure can only be made by manipulating the original source structure. We allow only the localization of items that are a child of the linked root item, in deeper levels this is not possible.

The local items in a linked *iTree* are integrated in the tree during the load process of the linked items. The identification is always made via the key (family-tag,family-index) of the item. The local storage of the tree contains iTree items that are merged as placeholders which will be replaced by the linked in items during the load process. Those placeholders are needed to create the matching key for the real items that should be kept after reload. In case the loaded structure is changed and and no matching item is found the placeHolder-items will remain in the *iTree*. All appended local items which are outside of the linked structure will be always positioned at the end of the tree.

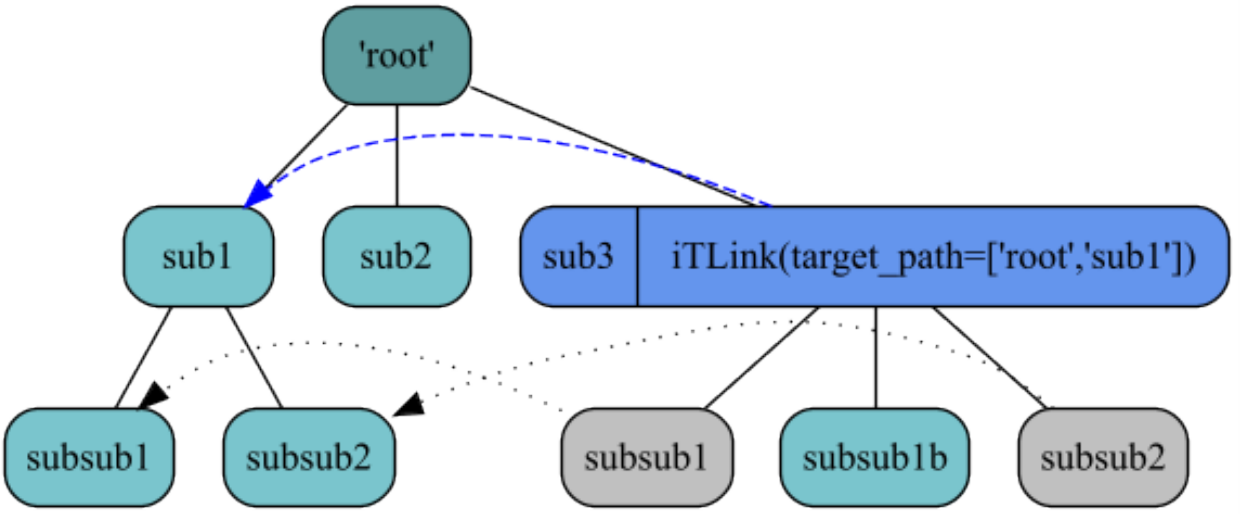


Fig. 10: Figure showing how “sub3” links to “sub1” item and “inherits” it’s subitems beside the local ones

Local items can be manipulated as normal *iTree* items with one exception. In case a local item is deleted and a matching linked item is available (was covered by the local item) the linked item will replace the local item after deletion. This means in this case a delete of an item will not reduce the numbers of the items. If the local item has no corresponding linked item the number of children will decrease as usual.

The linked items must be loaded and updated by an explicit operation. They are not loaded automatically. For this the method *load_links()* is used. The method can be executed at any level of the tree and it will start loading all links in the related subtree (use *load_links()*). By this mechanism incremental loads are possible. If the user wants to be sure that all linked items are loaded he must use the method in the root-object of the tree (load all links).

The behavior in case of load erros can be switched between Exceptions or deleting invalid items (via the *delete_invalid_items* parameter of the *load_links()*-method). In case of exceptions the *iTree* might be in an incomplete load state and if the exception is kept by the user this situation must be must be handled somehow (e.g. copy original tree before loading and replace back). The automated loading *iTree* links during instance of the object can be influenced via the *flags=iTFLAG.LOAD_LINKS* parameter that will activate the loading during instance.

Warning: The user must be aware that changing the source structure and local items in parallel might lead to unexpected results. **The identification of local items is always done via the key (family-tag,family-index)**. If we miss items during load placeholders are used to keep the key of the “real” local items. Normally those artefacts will be replaced during the load with the “real” linked items (if found) but in case of mismatches they will stay in the tree. Using wild linking in between different *iTree* items can lead into very confusing situations especially if the user removes local items. We recommend to use the feature only in special cases where the source architecture is clearly defined and remains structural relative stable. For stability reasons we have also functional limitations in linked iTree objects (e.g. we do allow only linking on not already linked items (protection for circular definitions); local items can never be linked items.

itertree.itree_main.iTree.**load_links**()

Runs ove all children and sub children in case a ITreeLink object is found the linked items are load in

In case ´iTree´ is link root: load all linked items

Parameters

- **force** –
  - False (default) - load only if not already loaded

- True - load even if already loaded (update)

- **delete_invalid_items** –
  - False (default) - in case of invalid items we will raise an exception!
  - True - invalid items will be removed from parent no exception raised
- **_items** – internal list parameter used for recursive calls of the function
- **_depth** – Internal parameter related to current item depth

Returns

- True - success
- False - load failed

property iTree.**is_linked**
In contrast to iTreeLinked class this is False

Return type bool

Returns True/False

property iTree.**is_link_root**
property that marks the iTree item as an item that contains a link

Returns

- True - is a link root item
- False is no iTree link item

property iTree.**is_link_loaded**

property iTree.**is_placeholder**
Property shows that item is a placeholder class

Normally there should be no placeholder class in the iTree but in case a loaded link does no more contain the expected items it might happen that such a class artifact is still in the tree. In placeholders the value contains the family index in the linked class.

Return type bool

Returns True/False

Beside this the following specific functions are available on linked items:

itertree.itree_main.iTree.**make_local**()
make the current linked object a local object This is only possible if the parent is a iTree object is the link root-> only the first level children in a linked iTree can be made local The operation raises an SyntaxError in case it is used on a deeper level of the linked tree

Returns None

For a better understanding please have a look in the example file examples/itree_link_example1.py in the package. That contains the following examples too.

Special functionalities related to linking of iTrees:

To link a subtree in an *iTree*-object the link=iTLink(file_path,target_path) is defined when the object is instanced. A link cannot be added later on to the object.

```
>>> # We create a small iTree:
>>> root = iTree('root')
>>> root += iTree('A')
>>> root += iTree('B')
>>> B = iTree('B')
>>> B += iTree('Ba')
>>> # we create multiple 'Bb' elements to show how the placeholders are used during␣
↪save and load
>>> B += iTree('Bb')
>>> B += iTree('Bb')
>>> B += iTree('Bc')
>>> root += B
>>> # !! Now we create a internal link (but we disable the loading (no flag set))):
>>> # (internal link -> iTLink(file_path==None,target_path= item identification)␣
↪(target_path like in get_deep())
>>> linked_element = iTree('internal_link', link=iTLink(target_path=[('B', 1)]))
>>> root.append(linked_element)
iTree('internal_link', link=iTLink(None,[('B', 1)]), flags=0b100000)
>>> root.render()
iTree('root')
 > iTree('A')
 > iTree('B')
 > iTree('B')
 .   > iTree('Ba')
 .   > iTree('Bb')
 .   > iTree('Bb')
 .   > iTree('Bc')
 > iTree('internal_link', link=iTLink(None,[('B', 1)]), flags=0b100000)
>>> root.load_links() # now we load the linked items
True
>>> root.render()  # The tree renderer marks linked items with ">>"
iTree('root')
 > iTree('A')
 > iTree('B')
 > iTree('B')
 .   > iTree('Ba')
 .   > iTree('Bb')
 .   > iTree('Bb')
 .   > iTree('Bc')
 > iTree('internal_link', link=iTLink(None,[('B', 1)]), flags=0b100100)
 .   >>iTree('Ba')
 .   >>iTree('Bb')
 .   >>iTree('Bb')
 .   >>iTree('Bc')
```

As shown in the example the internal linked item contains now the same subtree as the item ("B",1). But they are integrated as linked iTree objects which protects the items from changes (readonly). If we change the items in the "B" item the changes are only considered if we reload the links in the tree!

```
>>> root['B', 1] += iTree('B_post_append')
>>> root.render()
iTree('root')
 > iTree('A')
 > iTree('B')
 > iTree('B')
 .   > iTree('Ba')
 .   > iTree('Bb')
```

(continues on next page)

(continued from previous page)

```
 .   > iTree('Bb')
 .   > iTree('Bc')
 .   > iTree('B_post_append')
 > iTree('internal_link', link=iTLink(None,[('B', 1)]), flags=0b100100)
 .   >>iTree('Ba')
 .   >>iTree('Bb')
 .   >>iTree('Bb')
 .   >>iTree('Bc')
>>> root.load_links()  # The returning True signalizes that the tree was reloaded
True
>>> root.render()
iTree('root')
 > iTree('A')
 > iTree('B')
 > iTree('B')
 .   > iTree('Ba')
 .   > iTree('Bb')
 .   > iTree('Bb')
 .   > iTree('Bc')
 .   > iTree('B_post_append')
 > iTree('internal_link', link=iTLink(None,[('B', 1)]), flags=0b100100)
 .   >>iTree('Ba')
 .   >>iTree('Bb')
 .   >>iTree('Bb')
 .   >>iTree('Bc')
 .   >>iTree('B_post_append')
>>> root.load_links()  # If we repeat the action the command detects that the tree is␣
↪ unchanged and no update is needed
False
>>> root.load_links(force=True)  # Anyway the update can be forced
True
```

The toplevel linked *iTree*-object allow some manipulations of the subtree. We can append items and we can convert the linked sub-items into local-items that covers the linked item and that can contain different values and a different subtree. But we cannot change the order of the linked items! Therefore the commands like *insert()* or *append_left()* are not allowed.

```
>>> intern_link_item = root['internal_link', 0]  # get the linked item
>>> intern_link_item.append('new')  # append a local item
iTree(value='new')
>>> local = intern_link_item[2].make_local()  # make a linked item local (cover the␣
↪item with a local one)
>>> local.append(iTree('sublocal'))  # we change the subtree of the local item
iTree('sublocal')
>>> local.set_value('myvalue')  # we change the value of the local item
<class 'itertree.itree_helpers.NoValue'>
>>> root.render()  # see that in the linked tree we have local elements (linked items␣
↪are marked with ">>")
iTree('root')
 > iTree('A')
 > iTree('B')
 > iTree('B')
 .   > iTree('Ba')
 .   > iTree('Bb')
 .   > iTree('Bb')
 .   > iTree('Bc')
```

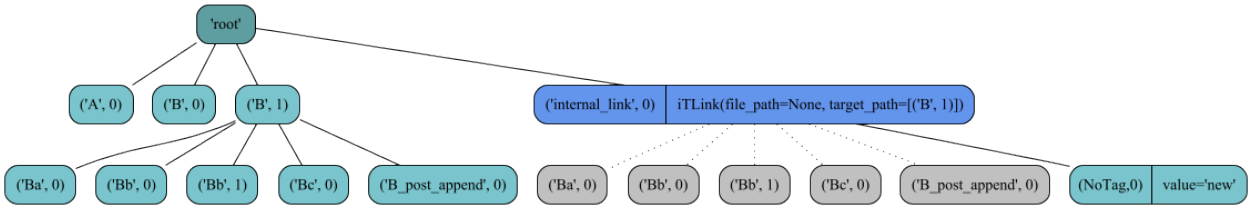
(continues on next page)

(continued from previous page)

```
 .  > iTree('B_post_append')
 > iTree('internal_link', link=iTLink(None,[('B', 1)]), flags=0b100100)
 .  >>iTree('Ba')
 .  >>iTree('Bb')
 .  > iTree('Bb', value='myvalue')
 .  .  > iTree('sublocal')
 .  >>iTree('Bc')
 .  >>iTree('B_post_append')
 .  > iTree(value='new')
```

The item 'Bb' in the linked subtree is now no more an iTreeLink object, its a normal *iTree* object. The identification of the covering item is internally always done via the TagIdx of the item. We can do all *iTree* related operations on this object. But there is one exception: if we delete the object the linked object will come back into the tree!

```
>>> del intern_link_item[('Bb', 1)]
>>> print(root.render())
iTree('root')
 > iTree('A')
 > iTree('B')
 > iTree('B')
 .  > iTree('Ba')
 .  > iTree('Bb')
 .  > iTree('Bb')
 .  > iTree('Bc')
 .  > iTree('B_post_append')
 > iTree('internal_link', link=iTLink(None,[('B', 1)]), flags=0b100100)
 .  >>iTree('Ba')
 .  >>iTree('Bb')
 .  >>iTree('Bb')
 .  >>iTree('Bc')
 .  >>iTree('B_post_append')
 .  > iTree(value='new')
None
```



The link functionality in iTrees can be understood like the overloading mechanism of classes. By linking a subtree in the tree this is like defining a superclass for a specific tree section. By making a subitem local this part of the linked *iTree* is covered (overloaded). But we should not stress this analogy to much because the functionalities in this covered data structures are much less then we have it in the class concept.

There are some quite difficult to understand aspects related to the linking of items. The ordering of loading the linked items and the mixing with the local items might be confusing. Especially if the user stores such *iTree*-objects in files and when the source is manipulated. The main order is always given by the linked elements and there keys (tag-idx-pairs). A not loaded but linked tree contains all local elements and placeholder items that mark where in the linked tree the local elements should be placed in. From the concept the local items where no linked counterpart is found will be always placed before the next linked local item (if it's a "real" one or a placeholder). All not filled local items will appended at the end during the *load_links()* process.

The most confusing things may happen if the user re orders the link source in a way that elements from the end are moved to the beginning. Original load scheme:

locals	linked	result
	iTree('link0')	iTree('link0')
iTree('tag1')		iTree('tag1')
iTree('tag2',value='new_value')	*iTree('tag2',value='link_value)*	iTree('tag2',value='new_value')
iTree('tag4')		iTree('tag4')
iTree('tag5',value='new_value')	*iTree('tag5',value='link_value)*	iTree('tag5',value='new_value')
iTree('tag6')		iTree('tag6')

Lets change the order of the source in the following way:

```
>>>root[('tag5',0)].move(('tag2',0))
```

After *load_links()* we will find the following situation

locals	linked	result
	iTree('link0')	iTree('link0')
iTree('tag4')		iTree('tag4')
iTree('tag5',value='new_value')	*iTree('tag5',value='link_value)*	iTree('tag5',value='new_value')
iTree('tag1')		iTree('tag1')
iTree('tag2',value='new_value')	*iTree('tag2',value='link_value)*	iTree('tag2',value='new_value')
iTree('tag6')		iTree('tag6')

In the linked source the *cursive* items have changed their position and the connected local items follow them.

The user might understand that the linked structure and order is somehow the main principle of ordering and the local items always follow this structure. So after the source is reordered the local items are reordered too. The local items that have no counterpart following always the anchor element afterwards (The item with 'tag4' is glued to 'tag5 and the item with 'tag1' is glued with 'tag2').

2.14 iTree - extensions

The itertree-package contains some extensions especially related to the build of data models which can be optionally used to determine the data stored in the *iTree*-value attribute.

2.14.1 Predefined Filters

As a help for filtering on *iTree*-objects the user can find the following predefined filter classes/methods under *itertree.itree_filters* :

itertree.itree_filters.**has_item_flags**()
Check the iTree flags for match to the given flag mask

Parameters

- **item** – *iTree*-item to be checked against the criteria of the method (for filtering out or not)
- **flag_mask** – flag mask E.g. can be build like: iTFLAG.READ_ONLY_TREE|iTFLAG.READ_ONLY_VALUE

Return type bool

Returns

- True -> match
- False -> no match

itertree.itree_filters.**is_item_tag**()
Check the iTree tag is equal to the given target_tag

Parameters

- **target_tag** – tag string do not give Tag() objects here! Use Tag().tag if really required
- **invert** –
  - False (default) -> unchanged result
  - True -> invert the result True->False; False->True

itertree.itree_filters.**has_item_tag_fnmatch**()
Check the iTree tag is matching to given fnmatch match_pattern

Parameters **match_pattern** – str or bytes related to fnmatch pattern definitions

itertree.itree_filters.**has_item_value**()
Check the iTree value is equal to given value

Parameters

- **target_value** – value object that should be equal with iTree.value
- **invert** –

- False (default) -> unchanged result
- True -> invert the result True->False; False->True

itertree.itree_filters.**has_item_value_dict_value**()
Check if in case the iTree value is a dict a value in the dict is equal to given value

Parameters

- **target_value** – value object that should be equal with iTree.value
- **invert** –
  - False (default) -> unchanged result
  - True -> invert the result True->False; False->True

itertree.itree_filters.**has_item_value_list_value**()
Check if in case the iTree value is a list a value in the list is equal to given value

Parameters

- **target_value** – value object that should be equal with iTree.value
- **invert** –
  - False (default) -> unchanged result
  - True -> invert the result True->False; False->True

itertree.itree_filters.**has_item_value_fnmatch**()
Check if value matches to the given fnmatch pattern

Parameters

- **target_value_pattern** – str or bytes related to fnmatch pattern definitions
- **invert** –
  - False (default) -> unchanged result
  - True -> invert the result True->False; False->True

itertree.itree_filters.**has_item_value_dict_value_fnmatch**()
Check if in case the iTree value is a dict a value in the dict matches to the given pattern

Parameters

- **target_value_pattern** – str or bytes related to fnmatch pattern definitions
- **invert** –
  - False (default) -> unchanged result
  - True -> invert the result True->False; False->True

itertree.itree_filters.**has_item_value_list_item_fnmatch**()
Check if in case the iTree value is a list a value in the list matches to the given pattern

Parameters

- **target_value_pattern** – str or bytes related to fnmatch pattern definitions
- **invert** –
  - False (default) -> unchanged result
  - True -> invert the result True->False; False->True

itertree.itree_filters.**is_item_value_in**()
Check if iTree value is in the given iTInterval object, no numeric values will be ignored

Parameters

- **target_key_interval** – msetInterval object defining the range (any object that supports "in" can be used)
- **invert** –
  - False (default) -> unchanged result
  - True -> invert the result True->False; False->True

itertree.itree_filters.**has_item_value_dict_value**()
Check if in case the iTree value is a dict a value in the dict is equal to given value

Parameters

- **target_value** – value object that should be equal with iTree.value
- **invert** –
  - False (default) -> unchanged result
  - True -> invert the result True->False; False->True

itertree.itree_filters.**has_item_value_list_value**()
Check if in case the iTree value is a list a value in the list is equal to given value

Parameters

- **target_value** – value object that should be equal with iTree.value
- **invert** –
  - False (default) -> unchanged result
  - True -> invert the result True->False; False->True

itertree.itree_filters.**has_item_value_dict_key**()
Check if in case the iTree value is a dict a key in the dict is equal with the given target_key no numeric values will be ignored

Parameters **target_key** – dict key

itertree.itree_filters.**has_item_value_list_idx**()
Check if in case the iTree value is a list the given target_key is lower than list length (inside) no numeric values will be ignored

Parameters

- **target_idx** – target-index
- **invert** –
  - False (default) -> unchanged result
  - True -> invert the result True->False; False->True

itertree.itree_filters.**has_item_value_dict_key_fnmatch**()
Check if in case the iTree value is a dict a key in the dict matches to the given key pattern (fnmatch) no numeric values will be ignored

Parameters

- **target_key_pattern** – str or bytes related to fnmatch pattern definitions

- **invert** –
  - False (default) -> unchanged result
  - True -> invert the result True->False; False->True

itertree.itree_filters.**has_item_value_dict_key_in**()

Check if in case the iTree value is a dict a key in the dict is in the given iTInterval object range no numeric values will be ignored

Parameters

- **target_key_interval** – msetInterval object defining the range (any object that supports "in" can be used)
- **invert** –
  - False (default) -> unchanged result
  - True -> invert the result True->False; False->True

2.14.2 Data Models

To control the data that can, be stored in an *iTree*-object value attribute (see **:ref:`iTree value related methods`_**) the itertree package contains some special classes related to the definition of data-models. Those models determine what kind of object can be stored inside the model. For this the user can define the target data-type, range-conditions, etc. Even the output formatting for string representations can be defined in those models.

The models might be useful and can be adapted by the user. But it's just an optional feature this is not related to the core functionality of itertree. The *iTree*-class can be used independent from this.

The main class for model definition can be found via the "Data" extension of itertree.

itertree.Data.**iTValueModel**()

This is the replacement for the old *iTDataModel()*-class.

This class can be used to define data models for the values that might be placed in the *iTree*.

The model should define min more detail which value objects are accepted or not and it defines also how not matching objects are handled:

- Deny the value and raise a ValueError exception
- Cast the value into a valid value object

But the definition of the data model allows limitations which goes far away from just data-type related topics. E.g. In the model the user can limit numerical values to a specific range (interval) or he might limit strings to specific characters.

All definitions related to checks and type casts must be defined by the user by overwriting the method *check_and_cast_single_item(value_item)* . The return of the method must e the checked and cast value. if the value does not match the method should raise a *ValueError* .

The method should always check and cast a single item, this is important. In case a list or an array like value should be stored in the model the base model will manage the required iteration over the sub-items and perform and utilize the single item check via the user defined method.

Note: In case a value like *[1,2,3,45]* is given each item in the list will be checked and/or casted.

This leads us to an important second definition functionality related to the model related to the size of dimensions (shape) of the value stored in the model. The definition of the shape is given as a parameter when the object

is instanced. For the shape we expect a tuple with the dimension information. If a value object is given the maximum shape will be calculated and this will be compared with the expected one. The maximum is used because in nested lists the user can define sub-list with different length. Strings or bytes aare also seen as arrays in this case!

We have the following possibilities to define shapes:

- shape=Any -> accept any shape of the given value (no check performed)
- shape=tuple() -> empty tuple given no dimension expected model will accept single values only!
- shape=(Any,) -> tuple containing one element which is the Any helper class; We will accept single values or any 1 dimensional object here (e.g. values like: *1*; *'abc'*; *[1,2,3,4]*)
- shape=(10,) -> tuple containing one element. We expect one dimensional values with a length lower or equal to the given integer number
- shape=(INF,) -> tuple containing one element that is INF (infinite). We expect one dimensional values of any length
- shape=(3,4) -> Two dimensions expected with fixed size (e.g. [[1],[2,3]] would match)
- shape=(INF,4) -> Two dimensions expected with first length unlimited and second length limited to 3
- shape=(4,ANY) -> Minimum one dimensions expected with first length limited to 4; here the user can also put infinite dimensions in (e.g [1]; [[1],[2]] ; [[[[888],[202,500]]]] would fit)
- shape=(4,ANY,INF) -> 1 dimension or 3 dimensions accepted, 2 dimension will not be accepted

---

Note: The model base object *iTValueModel()* contains two checking levels. First the user defined check via method definition for checks and casts of single items given. In second step the model also checks the dimension (shape) of the given value.

In case a *str* or *bytes* objects are given the behavior related to the checks will be a bit different as for the other objects. The method *check_and_cast_single_item(value_item)* will target the whole string as a single item! But the shape check will be done also on the string as an object with a length.

This means a string is a 1 dimensional object and the user might limit the size of the string via a shape. (E.g.: The object "Hello" has the shape: (5,); the object ['one', 'two','three'] has the shape: (3,5)) The user might use the method 'get_max_shape()' to measure the shape of objects that is considered in the model base class.

During the instance of the object a formatter can be defined too. This might help the user e.g. do define if an integer value should be converted to a hex or binary representation during string conversion. The build-in command str() of this model class will deliver the formatted value only. The repr() will deliver the class definition.

To use the model the user should put the instanced model object as value in the *iTree*. The real value objects can be placed during object instance via the parameter value or later on via the *set()* method of the model (value exchange too). In case the value is not matching to the model definition an ValueError exception will be raised. If the user like to test first if the value is matching he can use the *in* keyword to check this. In case of no match the exception content might be picked via the last_exception property of the model in this case (might give a hint why the value is not accepted).

Standard Parameters:

Parameters

- **`value`** – value object to be stored in the model (must match to the model). In case no value is stored in the model (empty model) the value will be NoValue.
- **`description`** – Description string

- **shape** – Define the dimensions the object should have:
  - None - shape is ignored object might have dimensions or not
  - tuple() - empty tuple or iterable - value object will have no size/dimension
  - (InfShape) - one dimensional value object with infinite size
  - (100) * one dimensional value object with max size of 100 items
  - (100,100) - two dimensional object with max size of 100 in each dimension
  - (InfShape,InfShape,InfShape) - three-dimensional object with infinite size in each dimension
  - ...

  Note: For multi-dimensional objects it's recommended to use numpy arrays or objects which have the attribute *shape* representing the size for each dimension available instead of tuples or lists. If not the object performance might be worse (internal iterations required to measure the shape).

- **formatter** – Formatter for the single item of the value object (see string formatting in python) In case no formatter is given *str()* will be used for creation of the item string representation.

To give the user an idea how this class might be used and for practical proposes we already defined a set of value models for typical data types:

itertree.Data.**iTAnyValueModel**()
Model that will take any python object without any restrictions

itertree.Data.**iTRoundIntModel**(*value=<class 'itertree.itree_helpers.NoValue'>*, *description=None*, *shape=<class 'itertree.itree_helpers.Any'>*, *contains=None*, *formatter=<class 'str'>*)
Model that would store integer values The model accepts any object that can be casted into a float and rounded to an integer to be stored as a int in the model

itertree.Data.**iTIntModel**()
This integer model allows only integers or strings containing a decimal integer to be stored in the model as int value

itertree.Data.**iTInt8Model**()
Integer model that limits the given values to int8 values

itertree.Data.**iTUInt8Model**()
Integer model that limits the given values to uint8 values

itertree.Data.**iTInt16Model**()
Integer model that limits the given values to int16 values

itertree.Data.**iTUInt16Model**()
Integer model that limits the given values to uint16 values

itertree.Data.**iTInt32Model**()
Integer model that limits the given values to int32 values

itertree.Data.**iTUInt32Model**()
Integer model that limits the given values to uint32 values

itertree.Data.**iTInt64Model**()
Integer model that limits the given values to int64 values

itertree.Data.**iTUInt64Model**()
Integer model that limits the given values to uint64 values

itertree.Data.**iTFloatModel**()
Float model that allows any float or string that can be casted to float to be stored in the model as float value

itertree.Data.**iTStrFnPatternModel**()
A model to store a string that matches to the fnmatch pattern

itertree.Data.**iTStrRegexPatternModel**()
A string model that matches to the regex pattern

itertree.Data.**iTASCIIStrModel**()
A string model that accepts only ASCII characters

itertree.Data.**iTUTF8StrModel**()
A string model that accepts only UTF-8 characters

itertree.Data.**iTUTF16StrModel**()
A string model that accepts only UTF16 characters

2.14.3 Mathsets extension

The itree package contains a extension we named mathsets which are a special kind of sets that can be used for range definitions in data-models but also for filtering a specific content.

The main check method for those kind of objects is the *__contains__*-method which is target via the build-in *in* statement.

The mathsets are a fragment of a new package that might be published in the future. The idea is mainly to extend the Python *set()* in a more mathematical way by adding for example interval sets and by allowing the user to define those sets by giving a mathematical definition string.

Therefore those classes might be interesting for the user independent from the usage related to *iTree*-objects. Especially the class *mSetInterval* is a full representation of a mathematical interval which allows also mathematical based object definitions like: *"{x| x e Z, -128<=x<128}"*

In itertree we have two main classes of mSets available:

itertree.itree_mathsets.**mSetInterval**()
Mathematical interval set object. Here the user can define a mathematical interval with closed or open boarders.

For more details related to mathematical intervals you may have a look here: https://en.wikipedia.org/wiki/Interval_(mathematics)

itertree.itree_mathsets.**mSetRoster**()
super class for all mSet objects

handles two parameters :param vars: variable names set :param complement: complement flag

Additional we have a helper classes that allows to combine those mathsets with each other or other objects (like normal Python sets).

itertree.itree_mathsets.**mSetCombine**()
class where the user can combine different sets to unions

In this class the user can combine different types of sets (all objects with *__contains__()* and a length are allowed to be added.

If the object is used to check if a value is in it is sufficient if the value is in one of the subsets to create a positive response for a match

These for classes are targeting numerical set definitions (Intervals, RosterSets, numerical domains). see https://en.wikipedia.org/wiki/Set_(mathematics) and https://en.wikipedia.org/wiki/Interval_(mathematics).

After we have presented the available classes we should give an idea of the usage. What might be the use case for this? In an application we might have the following needs:

1. We must create for the usage of the application a complex configuration
2. The configuration should be structure in a tree
3. The user should be capable to edit the value content of the attributes stored in the tree
4. The app should check if the given values are valid for the targeted attribute
5. The configuration should be shown in a GUI with a string representation
6. Some attributes contains range definition for tolerances
7. The user should be capable to define those tolerances with the help of mathematical descriptions

The most challenging attribute is in this case a tolerance definition that can be given by the user. Exactly for this case the mathset functionalities are very helpful.

2.14.4 iData

The "old" *iData*-class which was used as standard data-structure in *iTree*-objects in older versions is still available but must be added manually to the *iTree* as value object.

The object is in practice a dict-like structure which helps to manage the stored data values. But we would recommend to use normal dictionaries with data models in the items as a replacement.

2.15 Comparison of the iTree object with lists and dicts

In first case the *iTree* behaves like a *list*. Therefore all *list* related operations are supported in *iTree*-objects. Additionally in *iTree* we have the *dict* specific key related operations available too.

The following table compares the behaviors (x is always the related object)

Operation	iTree	list	dict
append	x.append(item)	x.append(item)	x[new_key]=item
appendleft	x.appendleft(item)	x.insert(0,item)	n.a.
append by +=	x+=item	x+=item	n.a.
extend	x.extend(items)	x.extend(items)	n.a. - x.update(items) goes in same direction -> (overwrites existing keys)
extendleft	x.extendleft(items)	n.a. - you migth change the target/source you are extending and use normal extend	n.a. - x.update(items) goes in same direction -> (overwrites existing keys)
insert	x.insert(target,item)	x.insert(index,item)	n.a.
delete	del x[target]	del x[index]	del x[key]
pop specific	x.pop(target)	x.pop(index)	x.pop(key)
pop last	x.pop() or x.pop(-1)	x.pop(-1)	x.popitem()
pop first	x.popleft() or x.pop(0)	x.pop(0)	n.a
remove	x.remove(value)	x.remove(value)	n.a.
move **	x[target1].move(,target2)	n.a.	n.a.
reorder	x[target1],x[target2],x[target3]= \ x[target2],x[target3],x[target1]	x[index1],x[index2],x[index3]= \ x[index2],x[index3],x[index1]	x[key1],x[key2],x[key3]= \ x[key2],x[key3],x[key1]

The target arguments used by iTree can be of different type and the result of the operations can be a single item or multiple items (iterator). Some operations which require unique targets will raise an exception if the target is not unique.

The following types/classes might be used as target parameter for the iTree related commands:

- *Tag(tag)* - this key targeting a whole family of items (can be any hashable object)
- *(tag,index)* - TagIdx object pointing to a specific item in a family
- *index* - index integer number
- *[index1,index2]* - A list of indexes targeting different items (works with iterators too)
- *iter([index1,index2])* - An iterator of indexes targeting different items (works with iterators too)
- *(tag,[index1,index2])* - A list of indexes targeting different items in a family
- *(tag,iter([index1,index2]))* - An iterator of indexes targeting different items in a family
- *slice* - slicing over items via index
- *(tag,slice)* - Slicing over the indexes of a specific family
- *method* - a method that is used for filtering of children (must deliver True/False)

** The difference in between move and reorder is that the move operation ensures that the original objects are kept. The reordering works only in case some of the objects are copied internally.

2.15.1 Special Typecasts

The iTree can be casted in lists and dicts in two ways:

1. Keep the 'iTree'-child-objects:

```
>>> root = iTree('root',subtree=[iTree('one', 1, subtree=[iTree('subone', 1.1), iTree(
↪'subtwo', 1.2)]), iTree('two', 2), iTree('three', 3)])
>>> list(root)
[iTree('one', value=1, subtree=[iTree('subone', value=1.1), iTree('subtwo', value=1.
↪2)]), iTree('two', value=2), iTree('three', value=3)]
>>> list(root.deep)  # flatten deep item list
[iTree('one', value=1, subtree=[iTree('subone', value=1.1), iTree('subtwo', value=1.
↪2)]), iTree('subone', value=1.1), iTree('subtwo', value=1.2), iTree('two', value=2),
↪ iTree('three', value=3)]
>>> dict(root.items())
{('one', 0): iTree('one', value=1, subtree=[iTree('subone', value=1.1), iTree('subtwo
↪', value=1.2)]), ('two', 0): iTree('two', value=2), ('three', 0): iTree('three',␣
↪value=3)}
>>> {k:i for k,i in root.deep.tag_idx_paths()}  # flatten deep items dict
{(('one', 0),): iTree('one', value=1, subtree=[iTree('subone', value=1.1), iTree(
↪'subtwo', value=1.2)]), (('one', 0), ('subone', 0)): iTree('subone', value=1.1), ((
↪'one', 0), ('subtwo', 0)): iTree('subtwo', value=1.2), (('two', 0),): iTree('two',␣
↪value=2), (('three', 0),): iTree('three', value=3)}
```

1. Consider just the stored values instead of the *iTree*-child-objects itself:

```
>>> root = iTree('root', subtree=[iTree('one', 1, subtree=[iTree('subone', 1.1),␣
↪iTree('subtwo', 1.2)]), iTree('two', 2), iTree('three', 3)])
>>> list(root.values())  # targets only first level children deeper hierarchy is lost
[1, 2, 3]
>>> [i.value for i in root.deep]  # flatten iterator delivering in-depth values of␣
↪items
```

(continues on next page)

(continued from previous page)

```
[1, 1.1, 1.2, 2, 3]
>>> dict(root.items(values_only=True))  # targets only first level children deeper␣
↪hierarchy is lost
{('one', 0): 1, ('two', 0): 2, ('three', 0): 3}
>>> {k:i.value for k,i in root.deep.tag_idx_paths()}  # in-depth levels are flatten␣
↪in the iterator
{(('one', 0),): 1, (('one', 0), ('subone', 0)): 1.1, (('one', 0), ('subtwo', 0)): 1.2,
↪ (('two', 0),): 2, (('three', 0),): 3}
```

In this casts to dicts the unique tag-idx-key (or relative tag_idx_path for deep operations) is used as the key in dicts (we cannot use the tags itself because they might not be unique).

2.16 Use iTree with unique tagged items

As explained collects the *iTree*-class multiple items with the same tag in a tag-family. But if the user likes to use the *iTree*-class with unique tags only (comparable to dictionaries where keys are unique) the *iTree*-class supports some specific functionalities for unique tagged trees. This should help the user in such a situation.

As user may have already read in this tutorial, we have an access function in the *get* subclass for single items:

`itertree.itree_getitem._iTreeGetitem.`**`single`**`()`

Call via **iTree().get.single()**

In general the methods does same like the "normal" *get()* but the method delivers only single (unique) results. In case *get()* delivers multiple items this method will raise an Exception or delivers the default value (if defined).

Note: In case the match contains a list with only one element the result is unique too. The method will unpack the unique item from the iterable and return it in this case.

Except If default parameter is not set an KeyError or IndexError will be raised. If result is not unique a ValueError will be raised

Parameters

- **`target`** (*`Union[int,tuple,list,slice]`*) – level 0 target object targeting a child or multiple children in the ´iTree´. Possible types are:
  - *index* - absolute target index integer (fastest operation)
  - *key* - key tuple (family_tag, family_index)
  - *index-slice* - slice of absolute indexes
  - *key-index-slice* - tuple of (family_tag, family_index_slice)
  - *target-list* - absolute indexes or keys to be replaced (indexes and keys can be mixed)
  - *key-index-list* - tuple of (family_tag, family_index_list)
  - *tag* - family_tag object targeting a whole family
  - *tag-set* - a set of family-tags targeting the items of multiple families
  - *itree_filter* - method (callable) for filtering the children of the object
  - *all-children* - if build-in *iter()* or ... (Ellipsis) is given a list of all children will be given (same result as *list(itree.__iter__())*)

- ***target_path** – in-depth targets iterable of targets for the different levels 1-n The supported targets in each level are (same like *__getitem__()*:
  - *index* - absolute target index integer (fastest operation)
  - *key* - key tuple (family_tag, family_index)
  - *index-slice* - slice of absolute indexes
  - *key-index-slice* - tuple of (family_tag, family_index_slice)
  - *target-list* - absolute indexes or keys to be replaced (indexes and keys can be mixed)
  - *key-index-list* - tuple of (family_tag, family_index_list)
  - *tag* - family_tag object targeting a whole family
  - *tag-set* - a set of family-tags targeting the items of multiple families
  - *itree_filter* - method (callable) for filtering the children of the object
  - *all-children* - if build-in *iter()* or ... (Ellipsis) is given a list of all children will be given (same result as *list(itree.__iter__())*)
- **default** (*object*) – If parameter is set in case of no match the default object will be delivered. If parameter is not set an Exception will be raised

Return type Union[*iTree*,object]

Returns found single item or default (in case default is set)

If the user gives here just the family tag as parameter and the family really contains just one item (unique tag) the method will deliver the unique item back. It's the "speciality" of the method to unpack items out of the list of family-items (normally delivered by *iTree.get()*) and in case of unique items it delivers the item directly.

But the method will raise an exception if the item is not found or what is more important if more than one item was found. (If the named parameter *default* is defined the default will be deliver instead of the exception raised.)

```
>>> root = iTree('root', subtree=[iTree('one', 1, subtree=[iTree('subone', 1.1),␣
↪iTree('subtwo', 1.2)]), iTree('two', 2), iTree('two', 2.2),iTree('three', 3)])
>>> root['one'] # targets the family and will deliver a list which contains in this␣
↪case only one item
[iTree('one', value=1, subtree=[iTree('subone', value=1.1), iTree('subtwo', value=1.
↪2)])]
>>> root.get.single('one')  # targets the same family but because we have just one␣
↪value the item inside the list is delivered directly
iTree('one', value=1, subtree=[iTree('subone', value=1.1),iTree('subtwo', value=1.2)])
>>> root.get.single('two')  # will raise an exception because we do not have a unique␣
↪result
Traceback (most recent call last):
...
ValueError: No single item found
>>> root.get('one', 'subone')  # targets in-depth and delivers the resulting list
[iTree('subone', value=1.1)]
>>> root.get.single('one', 'subone')  # Same method exists in get sub-class too
iTree('subone', value=1.1)
```

So we see that unique items can be indeed targeted only via tag if the *itree.get.single()*-method is used but how can we set single items comfortable? For the setting (and replacement) of unique items the *__setitem__()*-method with two special targets can be used. For both targets the family of the given item will be deleted before the new item is integrated.

- target: Ellipsis [...] - given *iTree*-object will be appended to the tree

- target: same family-tag as given item - given item will be inserted in the position of the first item of the deleted family.

Because the two targets ensure that the family is deleted before the item is integrated. The user can use them to integrate only unique tagged items in the tree. Different compared to the normal *__setitem__()* behavior the method will not raise an exception if the family-tag is not found in the tree. Furthermore the given new item will just be appended at the end of the already existing children.

Warning: IMPORTANT: The method will raise an exception if linked items found in the family! The reason is that we cannot reduce the number of items in the family if they are linked (original source must be modified for this). The single operation must be denied in this case.

```
>>> root['two']  # family 'two' contains two items:
[iTree('two', value=2), iTree('two', value=2.2)]
>>> root['two', 0].idx  # index of first item in 'two' family
1
>>> root['two']=(iTree('two', 'new')) # replace the two items in the family 'two'
>>> root.get.single('two')  # Now we get the unique item in this family
iTree('two', value='new')
>>> root.get.single('two').idx  # Index is same as before!
1
>>> root['two']=iTree('two', 'new2')  # replace again
>>> root.get.single('two')
iTree('two', value='new2')
>>> root.get.single('two').idx  # Index is same as before!
1
>>> root[...]=iTree('two', 'new3')  # replace and add at the end
>>> root.get.single('two')
iTree('two', value='new3')
>>> root.get.single('two').idx  # Index is now last index
2
```

As we have seen the *iTree*-class supports the handling of unique tags by some special commands. But the *iTree*-object is no not at all blocked for taking multiple tags in this case If the user utilizes other methods (e.g. standard *append()*) the object will still take multiple items with same tag! And the functions using tag_idx will still expect/deliver the tuple: *(tag,family-index)* (but family-index is always 0 for unique tags).

Related to performance we must remark that the tag_idx access (via *ìtree[(tag,idx)]* or *itree.get.tag_idx((tag,idx))* is quicker as the tag only access via *itree.get.single(tag)*. This should be considered in costly operations (e.g. loops).

If the user likes to "clean" an *iTree*-object from multiple children with same tag and only the first items in the families should resist he may run the code:

```
for tag_idx_path,item in list(itree.deep.tag_idx_paths()):
    if tag_idx_path[-1][-1]==0 and item.parent: # an already deleted items might not␣
↪have a parent
        item.parent[item.tag]=item
```

CHAPTER

THREE

ITERTREE PACKAGE

3.1 Indices and tables

- genindex
- search

3.2 Modules

3.3 The main itertree class

This code is taken from the itertree package: https://pypi.org/project/itertree/ GIT Home: https://github.com/BR1py/itertree The documentation can be found here: https://itertree.readthedocs.io/en/latest/index.html

The code is published under MIT license incl. human protect patch:

The MIT License (MIT) incl. human protect patch Copyright © 2022 <copyright holders>

Permission is hereby granted, free of charge, to any person obtaining a copy of this software and associated documentation files (the "Software"), to deal in the Software without restriction, including without limitation the rights to use, copy, modify, merge, publish, distribute, sublicense, and/or sell copies of the Software, and to permit persons to whom the Software is furnished to do so, subject to the following conditions:

Human protect patch: The program and its derivative work will neither be modified or executed to harm any human being nor through inaction permit any human being to be harmed.

The above copyright notice and this permission notice shall be included in all copies or substantial portions of the Software.

THE SOFTWARE IS PROVIDED "AS IS", WITHOUT WARRANTY OF ANY KIND, EXPRESS OR IMPLIED, INCLUDING BUT NOT LIMITED TO THE WARRANTIES OF MERCHANTABILITY, FITNESS FOR A PARTICULAR PURPOSE AND NONINFRINGEMENT. IN NO EVENT SHALL THE AUTHORS OR COPYRIGHT HOLDERS BE LIABLE FOR ANY CLAIM, DAMAGES OR OTHER LIABILITY, WHETHER IN AN ACTION OF CONTRACT, TORT OR OTHERWISE, ARISING FROM, OUT OF OR IN CONNECTION WITH THE SOFTWARE OR THE USE OR OTHER DEALINGS IN THE SOFTWARE.

For more information see: https://en.wikipedia.org/wiki/MIT_License

This part of code contains the main iTree object

class itertree.itree_main.**iTree**(*tag=<class 'itertree.itree_helpers.NoTag'>*, *value=<class 'itertree.itree_helpers.NoValue'>*, *subtree=None*, *link=None*, *flags=0*)

Bases: itertree.itree_private._iTreePrivate

__init__(*tag=<class 'itertree.itree_helpers.NoTag'>*, *value=<class 'itertree.itree_helpers.NoValue'>*, *subtree=None*, *link=None*, *flags=0*)

This is the main class related to itertree module. It represents the node in the nested tree structure.

In case the object contains a subtree this object is the parent of the children in the subtree and its inner children (children, sub-children, etc.). The ´iTree´-object itself can also be a child of a parent ´iTree´-object. If this is not the case the ´iTree´-object is the root of the tree.

Limitation: An ´iTree´-object can be integrated as a child in one ´iTree´ only (one parent only principle)!

Each ´iTree´-object contains a "tag". The objects tag can be any hashable object.

Different as dictionaries it is allowed to put multiple items with the same tag inside the ´iTree´. Those items with the same tag are placed and ordered (enumerated) in the related tag-family. The specific items can be targeted via a zag_idx tuple (family-tag,family-index) which is the items unique key.

Linked ´iTree´-objects will behave a bit special. They have a read only structure (children) and they contain the children (tree) of the linked ´iTree´. The "local" attributes like tag, value, ... can be set independent of the linked item (local properties).

To change the tree structure of such an object you can change the original link target. But an explicit reload (´load_links()´) is required to get the change active in the linked items.

Beside the linked item the user can add local items and mix them with the linked ones. But the general structure is always determined by the linked in children.

Beside the subtree the ´iTree´-object can also contain a value. The value can be any type of Python objects that is stored in the tree-node (comparable with the value of a dictionary). If it is required by the user to calculate the hash of the ´iTree´ via ´hash(item)´ some value objects might not be hashable and will raise an exception. But as long as the objects can be pickled a hash replacement will be found in the hash of ´iTree´. E.g. a dict placed as a value makes no troubles even if teh user likes to hash the tree.

As a helper the ´iTree´-object can be coupled with other objects (which might be helpful if you have a displayed tree in a GUI that is connected with the ´iTree´. Be aware that this helper function has only temporary character. It is not stored when dumping (standard dump) or considered in comparisons, etc. The coupled object is ignored by all internal functionalities. Also in linked items the coupled object is not taken over from the link and can be set independent.

The behavior of a ´iTree´ object can be influenced by specific properties or flags:

- Read-only tree: An ´iTree´ object where the subtree is protected and cannot be changed
- Read-only value: An ´iTree´ object where the value is protected and cannot be changed

The ´iTree´ object contains a large number of properties which should help the user to reach the required information as comfortable as possible. Especially the tree related information might be interesting:

- mytree.tag -> family-tag of the item
- mytree.idx -> absolute index of the object
- mytree.tag_idx -> key tuple (family-tag, family-index)
- mytree.idx_path -> tuple of absolute indexes from the root to the item
- mytree.tag_idx_path -> tuple of key-tuples from the root to the item
- mytree.parent -> parent item of the item
- mytree.root -> root item of the item (highest level parent)
- mytree.pre_item -> pre item (the children in the parent that is before this item)
- mytree.post_item -> post item (the children in the parent that is after this item)
- mytree.level -> How deep the item is in the tree related to the root

- mytree.max_depth -> How deep the sub-items (nested) of the ´iTree´ go in maximum (deep levels)

In case the ´iTree´ object is not part of another ´iTree´ (is root) those attributes will deliver in most cases ´None´.

- mytree.is_root -> True in case the item is a root ´iTree´)no parent)
- mytree.is_tree_read_only -> True in case the subtree is protected and read-only
- mytree.is_value_read_only -> True in case the item value is protected and read-only
- mytree.is_linked -> True in case the item is a linked item (read_only)
- mytree.is_link_root-> True in case the item is a root for a link to another ´iTree´
- mytree.link_root-> Delivers the related link-root in case the item is linked
- mytree.value-> Delivers the value object stored in the ´iTree´ item

There are different ways to access the children and sub-children in the tree of a ´iTree´ object.

The standard access for single items is via ´itree_obj[target]´ (´__getitem__(target)´) call. As targets the user has different options:

- index - absolute target index integer (fastest operation)
- key - key tuple (family_tag, family_index)
- tag or tag sets- family_tag object targeting a whole family
- target-list - absolute indexes or keys to be replaced (indexes and keys can be mixed)
- index slice - slice of absolute indexes
- key slice - tuple of (family_tag, family_index_slice)
- filter-method - method to filtering specific children

Beside the first level functions the *iTree*-object contains the helper class *.deep* which contains the in-depth functionalities targeting all the nested sub-children of the object.

As the name itertree should suggest a wide range of iteration methods are available in the class. They can be combined with different kind of filters.

Note: As optional *filter_method*-parameter the user can give:

- *None*- filter inactive
- *Callable* delivering *True*/*False* related to a characteristic of the *iTree*-object (iterated items)

Beside this the internal filtering is normally a hierarchical filtering (If the parent does not match to the filter all children are excluded too, even that they match to the filter). Some methods contain a switch for non-hierarchical filtering too. But most often the non-hierarchical filtering can be realized via the build-in *filter()* method and in this case the switch is not available.

Here the power of the iterators is obvious because cascaded filter queries can be constructed and finally in only **one** full iteration over all the items is required to get the results back (sometimes the full iteration is not required).

It's recommended to have a look into itertools package for better usage of the delivered iteration-generators.

The design of the ´iTree´´ object is made for best possible performance even that it is pure Python. Some part of the code might look less good readable or in the iteration-generators you find the if else outside

the iteration functionality which is not realized via sub-functions we have here redundant codings. But its is made to avoid conditions or function calls inside the loops which would be bad for the performance.

Parameters

- **tag** (*Hashable*) – family tag of the iTree object (any hashable object)
- **value** (*object*) – value to be stored in the iTree object
- **subtree** (*Optional[Iterable]*) – Iterable or Iterator containing the subtree items or an argument list (internal functionality)
- **link** (*Optional[iTLink]*) – iTLink object targeting another iTree
- **flags** (*int*) – flags taken from iTFLAG class:
  - iTFLAG.READ_ONLY_TREE - mark the subtree of this iTree as read-only the subtree will be protected from changes in this case
  - iTFLAG.READ_ONLY_VALUE - mark the value of this iTree object as read-only
  - iTFLAG.LOAD_LINKS - load the links during instance automatically

  Multiple flags can be combined via |

__iter__

property parent

Property delivers current items parent-object.

Return type Union[*iTree*, None]

Returns iTree parent-object or None (in case no parent exists)

property is_root

Is this item a root-item (has no parent)?

Return type bool

Returns

- *True* - is root
- *False* - is not root

property root

property delivers the root-item of the tree

In case the item has no parent it will deliver itself

Return type *iTree*

Returns iTree root item

property tag

This is the access to the object-tag. The tag gives the relation to the tag-family in *iTree*-objects.

The tag is comparable with a key in dictionaries but in iTrees the tag is not unique! For unique iTree identification the *tag_idx* property must be used.

Any hashable object can be used as a tag, but in case "exotic" objects are used and serialization is required the user may have to extend the functionality of the serializer.

Return type Hashable

Returns tag - hashable object giving the family relation

property idx
Index of this object in the iTree (related to the absolute order)

Method is very important for internal functionalities

Note: In general the item index is cached but in case of deleted items or reorder operations the cache might be outdated. In this case the index update based on a search might take longer.

Return type Union[int, None]

Returns unsigned integer representing the index (related to absolute order of iTree)

property idx_path
delivers a list of absolute indexes from the root to this item

For items with no parent (root_item) an empty tuple will be delivered

---

Note: We deliver here a tuple because it might be helpful if the object is hashable (usage as a dict key)

Return type tuple

Returns tuple of index integers (here we do not deliver an iterator!)

property tag_idx
The tag_idx is a unique identification of the item. It is represented by a tuple containing the family-tag and the family related index of the item.

If the item is not part of a parent-tree (root-item) in this case the result will be *None*.

Return type Union[tuple, None]

Returns tuple (family-tag, family-index) or None (if item has no parent)

property tag_idx_path
The path is a tuple of tag_idx tuples from root to this item. Each tag_idx is a tuple containing the pair family-tag and family-index.

For items with no parent (rooot_item) an empty tuple will be delivered

---

Note: We deliver here a tuple because it might be helpful if the object is hashable (usage as a dict key)

Return type tuple

Returns tuple of key tuples containing family-tag and family-index

force_cache_update(*idx=True*, *fam_keys=True*, *all_keys=True*)
Forces the update of the index and keys in cache

Normally this is not required the methode is mainly used for testing proposes

Parameters

- **idx** – True - update absolute-indexes
- **fam_keys** – True - update this items family-indexes
- **all_keys** – True - update all families faimily-indexes

property pre_item
Delivers the pre-item (predecessor) of this object in the parent-tree. If self is first item or there is no parent *None* will be delivered.

Return type Union[*iTree*,None]

Returns iTree predecessor or None (no match)

property post_item
Delivers the post-item (successor) of this object in the parent-tree. If self is first item or there is no parent *None* will be delivered.

Return type Union[*iTree*,None]

Returns *iTree* successor or *None* (no match)

property level
Delivers the distance (number of levels) to the root-item of the tree. Or in other words how deep in tree the item is positioned. In case item has no parent (is a root-item) this method will deliver 0.

Return type int

Returns integer - number of levels (outer direction)

property max_depth
Relative from this item the method measures the maximum depth of the tree and delivers the maximum number of levels that are found in this object.

If the user wants to now the maximum depth of the whole tree ensure that the property of the root-item is read. The user might use *my_tree.root.max_depth* to ensure this.

Return type int

Returns integer maximal number of levels that exists in the tree (inner direction)

property tag_number
property contains the number of tags (families) the itree contains :return: integer

property deep
Subclass containing the deep access to the nested structures of iTree :return:

property flags
Give the flags value of the object. The integer value stored in this property contains the bit flags related to the constants iTFLAG or _iTFLAG.

To see the details the user might use *bin()* or the helper property *flags_repr* which delivers a string containing all set flags.

;rtype: int :return: The flags set for this item

flags_repr(*public_only=True*)
String representation of flags for this item

Parameters **public_only** (*bool*) –

- True - Consider only the public flags (given by the user) -> default
- False - Show all flags (also linked and placeholder flags)

;rtype: str :return: String repr of the flags set for this item

property is_tree_read_only
Is the tree protection flag set? In this case the tree structure cannot be changed

This property targets the tree structure not the value!

Return type bool

Returns

- False - subtree can be changed (writeable)
- True - subtree is protected (read-only)

set_tree_read_only()
Set the tree protection flag. If the flag is set the subtree structure can not be changed anymore.

> **Warning:** Setting the structural protection is always a deep operation. In all children and sub-children the protection flag will be activated too! But when unset the behavior it is not automatically made as a deep operation`. Here the differentiation in between the two methods *unset_tree_read_only()* and *unset_tree_read_only_deep()* exists.

unset_tree_read_only()
Unset the tree protection flag on the item. Only the children structure of this item is made writable by this operation.

Except If the parent contains the tree protection flag a PermissionError will be raised

property is_value_read_only
Is iTree value read_only? Is the value protection flag iTFLAG.READ_ONLY_VALUE is set?

Return type bool

Returns True - read-only protection of value active False - value is writeable

set_value_read_only()
Set the write protection of the value (set flag: iTFLAG.READ_ONLY_VALUE)

unset_value_read_only()
Unset the write protection flag of the value (set flag: iTFLAG.READ_ONLY_VALUE). Value will be writeable afterwards

property value
Delivers the full value object stored in the *iTree*-object

Return type object

Returns value-object of the item

set_value(*value*)
Set/replace the value content of the *iTree*-object.

The method returns the previous stored value object that was replaced by the operation.

Note: If an *iTValueModel* is stored as value in the *iTree* by default the set_value() method will target the value which is stored inside the model. If the model itself should be exchanged the user must give the new model as value parameter of this method. To replace the model with another Python object the user must first delete the model via *del_value()* command and afterwards set the new value.

Parameters **value** (*object*) – data-object that should be placed as value or in case we have a *iTValueModel* already as value it is placed inside the model.

Return type object

Returns old value object that was stored in iTree before

set_key_value(*key*, *value*)

Depending on the already stored object this operation is a sub-replacement of a part only.

The method returns the previous stored value object that was replaced by the operation.

The user can influence the behavior by giving the *key* parameter. And it depends on the already stored value object (e.g. a *list* or *dict*). Only the value of the related item will be replaced or in case the item did not exist yet the might object will be extended by the given value (*dict* only).

Depending on given key parameter and the already stored object we have the following possible behaviours:

- dict stored in value -> store the value in the dict with the key given in key_index
- dict stored in value and matching item-value is a *iTValueModel* -> replace value inside the model
- list stored in value -> key_index must be an index and replace the related item in the list with the value given
- list stored in value and matching (index) item-value is a *iTValueModel* -> replace value inside the model
- key == *INF* and list stored in value -> append given value in the list

---

Note: If an *iTValueModel* is stored as value in the *iTree* by default the *mytree.set_value()*-method will target the value which is stored inside the model. If the model itself should be exchanged the user must give a new model as value parameter of this method. To replace the model with another Python object the user must first delete the model via *del mytree.value[key]* command and afterwards set the new value or he sets the value directly via *mytree.value[key]==new_value* .

Parameters

- **key** (*Optional[Hashable, int]*) – key or index of the value object (depends on the object already stored in *iTree*). if *key==INF* the value will be appended in case a list-like object is already stored in the *iTree*-object.
- **value** (*object,*) – value object that should be placed as value or in case a key is given the sub-value in the *iTree* or in case we have a *iTValueModel* is used inside the model.

Return type object

Returns old value object that was stored in iTree before

get_value()

Delivers the value-object of the item or a sub-value in case key_index parameter is used and a matching object is stored in the *iTree* .

Note: If *iTValueModel* is stored in *iTree* the method will not target the model it will target the value inside. If the model itself is required the *value*-property of *iTree* must be used.

Except In case a key_index is given but the object is not a *dict* or a *list* like object an *AttributeError* will be raised (*__getitem__()`required). If no matching item is found an `IndexError* or *KeyError* will be raised.

Return type object

Returns value object the *iTree* or *iTValueModel* (in case a model is stored in the *iTree*)

get_key_value(*key*)

Delivers the value-object of the item or a sub-value in case key_index parameter is used and a matching object is stored in the *iTree* .

In case the stored value is a *dict*-like object the key will be used as the key of the dict. In case the stored value is a *list*-like object the keyx will be used as the index of the list.

In case the target value is a *iTValueModel* the value inside will be targeted and not the model itself.

Note: If *iTValueModel* is stored in *iTree* the method will not target the model it will target the value inside. If the model itself is required the *value*-property of *iTree* must be used.

Except In case a key_index is given but the object is not a *dict* or *list* like object an *AttributeError* will be raised (*__getitem__()*-method required). If no matching item is found an *IndexError* or *KeyError* will be raised.

Parameters **key** (*Optional[Hashable, int]*) – Optional key or index parameter

Return type object

Returns value object the *iTree* or *iTValueModel* (in case a model is stored in the *iTree*)

del_value()

Deletes the full value-object stored in ´iTree´ (´NoValue´ is stored in iTree).

This method will always delete the whole object stored in *iTree* even *iTValueModel*-objects are deleted. To delete the value content of a model *mytree.value.clear()* or 'set_value(NoValue)' might be used.

Returns deleted value

del_key_value(*key*)

If no parameter is given deletes the full value-object stored in ´iTree´ (store ´NoValue´).

In case a key or index is given and the value contains a matching object we will only pop out the related sub-item.

This method will always delete the whole targeted object even *iTValueModel*-objects are deleted. To delete the value content of a model *mytree.value.clear()* or 'set_value(NoValue)' might be used.

Except In case a key is given but the object is not *dict* or *list* like a TypeError or AttributeError will be raised (*__delitem__()*-method is targeted); If the given key does not exist or an invalid parameter is given a KeyError or IndexError will be raised.

Parameters **key** (*Optional[Hashable, int]*) – Optional key or index to exchange just sub-items in the value

Returns deleted value

property coupled_object

The *iTree*-object can be coupled with another Python-object. The pointer to the object is stored and can be reached via this property. (E.g. this can be helpful when connecting the *iTree* with a visual item (hypertree-list item) in a GUI)

Returns pointer to coupled-object or None if no object is stored

set_coupled_object(*coupled_object*)

Couple another Python-object with this *iTree*-object.

Compared with the *value* the coupled-object is not tracked by any internal functions. We do not consider it in any relation (e.g. *__contains__()* and do not dump it in files, etc. Even in linked items the coupled-object is not protected. And in copies it is ignored and not taken over.

Note: E.g. The coupled-object might be an object in a GUI that is related to this item.

Parameters **coupled_object** – object pointer to the object that should be coupled with this iTree item

append(*item=<class 'itertree.itree_helpers.NoValue'>*)

Append the given *iTree*-object to the *iTree* (new last child) The *append()* method is the fastest way to add a single item to the end of the tree.

Except In case *iTree*-object has already a parent a *RecursionError* will be raised Other exceptions might come up in case the *iTree* is protected (tree read-only mode).

Parameters **item** (*Union[iTree, object]*) – *iTree*-object to be appended

Warning: In case the given item-object is not a *iTree*-object the item is interpreted as a value and the *iTree* will be created implicit (with tag-family *NoTag*) in the way:

iTree(tag=NoTag, value=item) ~ ìTree(value=item) If no item is given an empty iTree is created tag=`NoTag`; value=`NoValue`.

```
>>> root=iTree('root')
>>> root.append('myvalue')
iTree(value='myvalue')
>>> root.append() # append an empty iTree-object
iTree()
```

Return type *iTree*

Returns Delivers the appended item itself (it might be useful for the user to get the updated information of the object).

appendleft(*item=<class 'itertree.itree_helpers.NoValue'>*)

Append the given *iTree*-object to the left of the parent-tree (new first child) The *appendleft()* method is the recommended method to add a new first item to iTree (quicker than *insert(0,item)*). Compared to *append()* the method is slower and the cache index information gets invalid after the operation (will be automatically updated later on if required).

Except In case *iTree*-object has already a parent a *RecursionError* will be raised. Other exceptions might come up in case the *iTree* is protected (tree read-only mode).

Parameters **item** (*Union[iTree, object]*) – *iTree*-object to be appended as first item.

Warning: As in *append()* in case the given item-object is not a *iTree*-object the item is interpreted as a value and the *iTree* will be created implicit.

Return type *iTree*

Returns Delivers the appended item itself (it might be useful for the user to get the updated information of the object).

insert(*target*, *item=<class 'itertree.itree_helpers.NoValue'>*)

Insert an item **before** a given target-position. The insertion works like in lists.

The insertion operation is slower as the append operations.

If *target=None* is given the operation inserts in the last position (== *append()*).

Except In case *iTree*-object has already a parent a *RecursionError* will be raised Other exceptions might come up in case the *iTree* is protected (tree read-only mode).

Parameters

- **target** (*Union[Integer,tuple,iTree,None]*) – target position definition; **target must target a single/unique item!** Possible targets:
  - index - absolute target index integer, negative values supported too (count from the end).
  - key - key-tuple (family_tag, family_index) pair
  - item - *iTree*-item that is already a children (future successor)
  - None - if *None* is given we will append the item in the last position of the ´iTree´-object
- **item** (*Union[iTree,object]*) – *iTree*-object to be inserted in the tree.

> **Warning:** As in *append()* in case the given item-object is not a *iTree*-object the item is interpreted as a value and the *iTree* will be created implicit.

Return type *iTree*

Returns Delivers the inserted item itself (it might be useful for the user to get the updated information of the object).

extend(*items*)

We extend the *iTree* with given items (multi append). The function is high performant and if you have to append a large number of items it is recommended to create an iterator of the items and feed them into this method. This is quicker compared to a loop doing multiple normal *append()* operations.

Note: In case the to be extended items have already a parent an implicit copy will be made. We do this because the internal copy can be created more effective. We accept also iTree-objects as extend_items parameter and the children which have a parent will be automatically copied to be integrated in this second tree. We have the same situation with a filtered iterator which might be used to extend this *iTree* too.

Parameters **items** (*Iterable*) – iterable-object that contains *iTree*-objects as items it can be:

- iterator or generator of *iTree*-objects (using next)
- *iTree*-object (children will be copied and extended in this tree)
- iterable of *iTree*-objects (list, tuple, . . .)
- argument list for *iTree*-instance (´__init__()´) (created by ´get_init_args()´ or ´get_init_args_deep()´) -> this is most often an internal functionality.
- iterator or generator of value-objects (using next) - implicit *iTree*-objects created
- iterable of value-objects (list, tuple, . . .)- implicit *iTree*-objects created

extendleft (*items*)

Multy item append on left hand-side (at the beginning) of the ´iTree´.

The operation is slower than ´extend()´ because it requires a reordering of all items in the *iTree*.

Note: The order of extended items is kept in the operation. It's comparable with: ´[1,2,3]+[4,5,6]=[1,2,3,4,5,6]´ but the result is not a new instance, self is kept.

Note: In case the to be extended items have already a parent an implicit copy will be made. We do this because the internal copy can be created more effective. We accept also iTree-objects as extend_items parameter and the children which have a parent will be automatically copied to be integrated in this second tree. We have the same situation with a filtered iterator which might be used to extend this *iTree* too.

Parameters **items** (*Iterable*) – iterable-object that contains *iTree*-objects as items it can be:

- iterator or generator of *iTree*-objects (using next)
- *iTree*-object (children will be copied and extended in this tree
- iterable of *iTree*-objects (list, tuple, . . .)
- argument list for *iTree*-instance (´__init__()´) (created by ´get_init_args()´ or ´get_init_args_deep()´)
- iterator or generator of value-objects (using next) - implicit *iTree*-objects created
- iterable of value-objects (list, tuple, . . .)- implicit *iTree*-objects created

__setitem__ (*target*, *value*)

Replace an item with the given new item given in the *value*-parameter. The method handles also multiple replaces (rearrangements) like:

```
>>> mytree[1],mytree[0]=mytree[0],mytree[1]
```

Warning: Because of the parent only principle in rearrangements operations an implicit copy might be created.

Note: Linked items cannot be changed. If changes are required The user must change the link source tree items and afterwards actively rerun *load_links()* to reload the linked tree.

Except In case the target is not found or the *iTree* is protected (read-only tree).

Parameters

- **target** – target object defining the replacement target; possible types are:
  - index - absolute target index integer (fastest operation)
  - key - key tuple (family_tag, family_index)
  - tag - Tag(family_tag) object targeting a whole family

- target-list - absolute indexes or keys to be replaced (indexes and keys can be mixed)
- index slice - slice of absolute indexes
- key slice - tuple: (family_tag, family_index_slice)

For multi targets the given value must have a matching structure (item list with same length).

We have two special targets which are used for placing/replacing single items in the iTree:

- Ellipsis ... - new_items tag-family will be deleted and the new-item is placed in families first item position
- items_tag - new_items tag-family will be delted and the new-item is placed in families last item position

If those two special targets are used and the new-items family does not exist yet, the method will just append the new item, no exception will be raised.

- **value** – iTree object that should replace the target or in case of multi targets a tuple of items that should be used for replacements

Returns value added items (only for internal usage)

move(*target=None*)

Move this item in given target position (item will be positioned **before** the given target). The given target must be a unique item! If None is given the item will be moved in the last position of the *iTree*. If an ìTree`-object is given as target it must be a children of the same parent (sibling).

Except LookupError in case the target is not found or not unique!

Parameters **target** (*Union[Integer,tuple,iTree,None]*) – target-object defining the replacement target; possible types are:

- index - absolute target index integer, negative values supported too (count from the end).
- key - key-tuple (family_tag, family_index) pair
- item - *iTree*-item that is already a children (future successor)
- None - if *None* is given we will move the item to the last position in the ´iTree´-object

Returns self (with updated indexes)

rename(*new_tag*)

give the item a new family tag

The renaming of the item implies a reordering of the items in the tree because the family order depends on the global/absolute order of items.

Parameters **new_tag** (*Hashable*) – new tag (any kind of hashable object)

Return type *iTree*

Returns Delivers the renamed item itself (it might be useful for the user to get the updated information of the object).

reverse()

Reverse the order of all children in the *iTree*.

If you do not want to change the object itself (in place operation) you might use the iterator *reversed()* instead.

rotate(*n=1*)

Rotate children of the *iTree*-object n times (n positions) (rotate 1 times means move last item to first position)

If no parameter is given we rotate by one position only.

The rotation can be made in negative direction too (give negative numbers).

In case zero is given the operation is neutral and nothing will be changed.

Note: There is no in-depth counterpart of this method available.

Parameters **n** (*integer*) – number of positions the items should be rotated

sort(*key=None*, *reverse=False*)

Sorting operation -> same behavior as sort of lists (parameter description is taken from list documentation).

Note: This is an "in place" operation which changes the content of the object the build-in *sorted()* might be use instead (if the original object should not be changed):

```
>>> a=iTree(subtree=[iTree(3),iTree(2),iTree(4),iTree(1)])
>>> a.render()
iTree()
> iTree(3)
> iTree(2)
> iTree(4)
> iTree(1)
>>> b=iTree(subtree=(a[i] for i in sorted(a.keys())))
iTree()
> iTree(1)
> iTree(2)
> iTree(3)
> iTree(4)
```

Internally in this operation a copied sorted list is created, and afterwards the whole structure is cleared and rebuild based on the sorted list.

The default-operation is to the sort based on the list of keys (tag-family, family_index) pair of the items. The base of the sorting can be modified by changing the *target_type* parameter.

Parameters

- **key** – specifies a function of one argument that is used to extract a comparison key from each list element (for example, key=str.lower). The key corresponding to each item in the list is calculated once and then used for the entire sorting process. The default value of None means that list items are sorted directly without calculating a separate key value.
- **reverse** – is a boolean value. If set to True, then the list elements are sorted as if each comparison were reversed.

__delitem__(*target*)

The function deletes the targeted item in the tree.

Except In case the target is not found or the *iTree* is protected (read-only tree).

Parameters **target** (*Union[int,tuple,Hashable,Iterable,slice]*) – target object defining the replacement target; possible types are:

- *index* - absolute target index integer (fastest operation)
- *key* - key tuple (family_tag, family_index)
- *tag* - Tag(family_tag) object targeting a whole family
- *target-list* - absolute indexes or keys to be replaced (indexes and keys can be mixed)
- *index-slice* - slice of absolute indexes
- *key-slice* - tuple of (family_tag, family_index_slice)
- *itree_filter* - method (callable) for filtering the children of the object

Returns deleted item

clear(*keep_value=False*, *local_only=False*)

deletes all children and the value!

All flags stay unchanged, except the load_links flag!

Parameters

- **keep_value** (*bool*) –
  - True - value is not deleted
  - False - value will be replaced with NoValue
- **local_only** (*bool*) –
  - True - clear only the local items
  - False - clear whole object (The object is reset to the no links loaded state and locals are deleted)

pop(*target=- 1*)

pop the item out of the tree, if no key is given the last item will be popped out

We do not have the method popleft because *pop(0)* does the same.

Parameters **target** (*Union[int,tuple,Hashable,Iterable,slice,iTree]*) – target of popped item(s):

- *index* - absolute target index integer (fastest operation)
- *key* - key tuple (family_tag, family_index)
- *tag* - Tag(family_tag) object targeting a whole family
- *target-list* - absolute indexes or keys to be replaced (indexes and keys can be mixed)
- *index-slice* - slice of absolute indexes
- *key-slice* - tuple of (family_tag, family_index_slice)
- *itree_filter* - method (callable) for filtering the children of the object

Returns popped out item(s) (parent will be set to None). In case multiple items are removed an iterator over the removed items is given.

remove(*item*)

With remove the given target is a *iTree* child that should be removed.

The method is only in because we like to be compatible with lists interface but the pop method target allows already to use a child as a target too.

Except If given item is not a child of the parent or the ìTree`-objects tree is protected

Parameters **item** (*Union[iTree, Iterable]*) – Child or iterable of children to be removed from the tree

Returns removed item(s) (parent will be set to None) - in case of multiple removes the method delivers a list no iterator because anyway a list is created

__getitem__(*target*)

Main common get method for children (first level items).

In case the given targets is a absolute index or a key (tag,family-index) pair the method will deliver a unique item back. This operation is prioritized over the other operations.

For all other targets the method will deliver a list with the targeted items as result.

In some cases an empty list might be delivered and no exception might be raised (e.g. filter query delivers no match).

In case user likes to have other return-types he might check the other available get methods (*get()*, *get.single()*, *get.iter()*) or he might also use the itertree helper method *getter_to_list()* to convert any of the possible results into a list.

Except In case of no match (even if a part is not matching (e.g. one index in an index-list) the method will raise a KeyError (no matching target given); IndexError (no matching index given) or ValueError (no valid type of target given).

Parameters **target** (*Union[int, tuple, list, slice]*) – target object targeting a child or multiple children in the ´iTree´. Possible types are:

- *index* - absolute target index integer (fastest operation)
- *key* - key tuple (family_tag, family_index)
- *index-slice* - slice of absolute indexes
- *key-index-slice* - tuple of (family_tag, family_index_slice)
- *target-list* - absolute indexes or keys to be replaced (indexes and keys can be mixed)
- *key-index-list* - tuple of (family_tag, family_index_list)
- *tag* - family_tag object targeting a whole family
- *tag-set* - a set of family-tags targeting the items of multiple families
- *itree_filter* - method (callable) for filtering the children of the object
- *all-children* - if build-in *iter* or . . . *`(Ellipsis) is given a list of all children will be given (same like list(itree.__iter__())*)

Return type Union[*iTree*,list]

Returns Target was *index* or *key* -> one *iTree* item will be given; for all other targets a list will be delivered.

copy_keep_value()

Create a copy of this item.

The difference in between normal *copy()* and this method is that the value objects are completely untouched in this operation (for immutable objects there is no difference in between the two copy operations).

Returns copied iTree object

copy (**args*, ***kwargs*)
create a copy of this item

The difference in between *copy()* and *deepcopy()* for *iTree* is just that we do in *deepcopy()* a deepcopy of all value items. In *copy()* we just copy the value object not the items inside, the pointers to the original objects are kept (for immutable objects there is no difference).

Returns copied iTree object

deepcopy (**args*, ***kwargs*)
create a deepcopy of this item

The difference in between *copy()* and *deepcopy()* for *iTree* is just that we do in *deepcopy()* a deepcopy of all value items. In *copy()* we just copy the value object not the items inside, the pointers to the original objects are kept (for immutable objects there is no difference).

Returns deep copied new iTree object

filtered_len (*filter_method*)
Calculates the number of filtered children.

Parameters **filter_method** (*Callable*) – filter method that checks for matching items and delivers *True*/*False*. The filter_method targets always the *iTree*-child-object and checks a characteristic of this object for matches (see *filter_method*)

Return type int

Returns Number of matching items found

is_tag_in (*tag*)
Checks if a iTree contains the given family-tag (first-level only) :param tag: family tag :return: True/False

is_in (*item*)
Checks if the given object is child of the iTree. Different to ´__contains__()´ we check here for the instance (specific) object (is) and not based on ´__eq__()´.

Parameters **item** – iTree object to be searched for

Returns

- True - matching child is found
- False - no matching item found

__eq__ (*other*)
compares if the tag, value and children content of another item matches with this item

---

Note: If you like to check if it is really the same object you should use ´is´ instead of ´==´ operator

---

Parameters **other** – other iTree

Returns boolean match result (True match/False no match)

equal (*other*, *check_coupled=False*, *check_flags=False*)
compares if the data content of another item matches with this item

Parameters

- **other** – other iTree
- **check_coupled** – check the couple object too? (Default False)
- **check_flags** – check the flags of the objects? (Default False)

Returns boolean match result (True match/False no match)

count (*item*)
Counts how many equal (==) children are in the *iTree*-object.

Parameters **item** (iTree) – The *iTree*-items will be compared with this item

Return type int

Returns Number of matching items found

index (*item*, *start=None*, *stop=None*)
The index method allows to search for the absolute index of a matching item in the *iTree*. The item must be a iTree object and the index will deliver the first match. The comparison is made via == operator.

If item is not found a IndexError will be raised

Note: To get the index of a specific item instance the *.idx*- property should be used.

Parameters

- **item** (iTree) – iTree object to be searched for
- **start** (*Union[iTree,target_path]*) – iTree item or start target_path where index search should be started (start item is included in search)
- **stop** (*Union[iTree,target_path]*) – iTree item or stop target_path where index search should be stopped (stop item is not included in search)

;rtype: int :return: absolute index of the found item

keys (*filter_method=None*)
Iterates over all children and deliver the children tag-idx tuple (family-tag,family_index)

Note: This is a dict like iterator that delivers the unique keys for all children.

Parameters **filter_method** (*Union[Callable,None]*) – filter method that checks the item and delivers *True*/*False*. The filter_method targets always the *iTree*-child-object and checks a characteristic of this object for matches

If *None* is given filtering is inactive.

Return type Iterator

Returns iterator over the tag-idx of the children

values (*filter_method=None*)
Iterates over all children and deliver the children values

Parameters **filter_method** (*Union[Callable,None]*) – filter method that checks for matching items and delivers *True*/*False*. The filter_method targets always the *iTree*-child-object and checks a characteristic of this object for matches (see *filter_method*)

If *None* is given filtering is inactive.

Return type Iterator

Returns iterator over the values stored in the children

items (*filter_method=None*, *values_only=False*)

Iterates over all children and deliver the children item-tuples (key,item) or (key,value). As key we use the unique tag-idx: (tag-family,family-index).

The function is comparable with dicts *items()* function.

Parameters

- **filter_method** (*Union[Callable,None]*) – filter method that checks for matching items and delivers *True*/*False*. The filter_method targets always the *iTree*-child-object and checks a characteristic of this object for matches (see *filter_method*)

  If *None* is given filtering is inactive.

- **values_only** (*bool*) –
  - *False* (default) - in the key,value tuple the iterator put the iTree object as value in
  - *True* - in the key,value tuple the iterator put "only" the value object of the *iTree*-object in

Return type Generator

Returns iterator over the target keys and item value of the children

iter_families (*filter_method=None*, *order_last=False*)

This is a special iterator that iterates over the families in *iTree*. It delivers per family the tag and a list of the containing items. The order is defined by the absolute index of the first item in each family

Method will be reached via *iTree.Families.iter()*

Parameters

- **filter_method** (*Union[Callable,None]*) – filter method that checks for matching items and delivers *True*/*False*. The filter_method targets always the *iTree*-child-object and checks a characteristic of this object for matches (see *filter_method*)

  If filter_method is None no filtering is performed

  Note: An internal filtering is available because this may change the order of the delivered items. An external filter with same method might deliver a different result!

- **order_last** (*bool*) –
  - False (default) - The tag-order is based on the order of the first items in the family
  - True - The tag-order is based on the order of the last items in the family

Return type Generator

Returns iterator over all families delivers tuples of (family-tag, family-item-list)

iter_family_items (*order_last=False*)

This is a special iterator that iterates over the families in *iTree*. It iters over the items of each family the ordered by the first or the last items of the families.

Parameters **order_last** (*bool*) –

- False (default) - The tag-order is based on the order of the first items in the family
- True - The tag-order is based on the order of the last items in the family

Return type Generator

Returns iterator over all families delivers tuples of (family-tag, family-item-list)

tags (*order_last=False*)
iters over all family-tags in level 1 (children). The order is based on first or last item in the family.

Parameters **order_last** (*bool*) –

- False (default) - The tag-order is based on the order of the first items in the family
- True - The tag-order is based on the order of the last items in the family

Return type Iterator

Returns tag iterator

renders (*filter_method=None*, *enumerate=None*, *renderer=<class 'itertree.itree_serializer.itree_renderer.iTreeRender'>*)
render the iTree into a string

Parameters

- **filter_method** (*Union[Callable, None]*) – filter method that checks for matching items and delivers *True*/*False*. The filter_method targets always the *iTree*-child-object and checks a characteristic of this object for matches (see *filter_method*)

  If *None* is given filtering is inactive.

  The method uses the given filter always as an hierachical filter.

- **enumerate** (*bool*) –
  - True - Add an enumeration before the items
  - False (default) - Output without enumeration
- **renderer** (*class*) – Give another renderer class for different formatting

Return type str

Returns Tree representation as string

render (*filter_method=None*, *enumerate=False*, *renderer=<class 'itertree.itree_serializer.itree_renderer.iTreeRender'>*)
Print the rendered string of the *iTree*-object to the console (stdout).

Parameters

- **filter_method** (*Union[Callable, None]*) – filter method that checks for matching items and delivers *True*/*False*. The filter_method targets always the *iTree*-child-object and checks a characteristic of this object for matches. If *None* is given filtering is inactive.
- **enumerate** – add an enumeration before the rendered items
- **renderer** – Render to be used (The given render is stored and will be used until another renderer is given).

Returns

get_init_args (*filter_method=None*, *_subtree_not_none=True*)
Method creates list of arguments that can be used as a pointer to create an equal instance of an iTree object. This is a method is used in most cases for internal functionalities (especially copy()).

Parameters

- **filter_method** (*Union[Callable, None]*) – filter method that checks for matching items and delivers *True*/*False*. The filter_method targets always the *iTree*-child-object and checks a characteristic of this object for matches (see *filter_method*)

  If *None* is given filtering is inactive.

- **_subtree_not_none** – internal parameter controlling if the subtree is added or not

Returns

loads(*data_str*, *check_hash=True*, *load_links=True*, *itree_serializer=<class 'itertree.itree_serializer.itree_json_serialize.iTStdJSONSerializer2'>*)
create an iTree object by loading from a string

If not overloaded or reinitialized the iTree Standard Serializer will be used. In this case we expect a matching JSON representation.

Parameters

- **data_str** – source string that contains the iTree information
- **check_hash** – True the hash of the file will be checked and the loading will be stopped if it doesn't match False - do not check the iTree hash
- **load_links** – True - linked iTree objects will be loaded
- **itree_serializer** – optional user defined serializer for iTree objects

Returns iTree object loaded from file

load(*file_path*, *check_hash=True*, *load_links=True*, *itree_serializer=<class 'itertree.itree_serializer.itree_json_serialize.iTStdJSONSerializer2'>*)
create an iTree object by loading from a file

If not overloaded or reinitialized the iTree Standard Serializer will be used. In this case we expect a matching JSON representation.

Parameters

- **file_path** – file path to the file that contains the iTree information
- **check_hash** – True the hash of the file will be checked and the loading will be stopped if it doesn't match False - do not check the iTree hash
- **load_links** – True - linked iTree objects will be loaded
- **itree_serializer** – optional user defined serializer for iTree objects

Returns iTree object loaded from file

dumps(*calc_hash=False*, *filter_method=None*, *itree_serializer=<class 'itertree.itree_serializer.itree_json_serialize.iTStdJSONSerializer2'>*)
serializes the iTree object to JSON (default serializer)

Parameters

- **calc_hash** – Tell if the hash should be calculated and stored in the header of string
- **itree_serializer** – optional user defined serializer for iTree objects

Returns serialized string (JSON in case of default serializer)

dump(*target_path*, *pack=True*, *calc_hash=True*, *overwrite=False*, *filter_method=None*, *itree_serializer=<class 'itertree.itree_serializer.itree_json_serialize.iTStdJSONSerializer2'>*)
serializes the iTree object to JSON (default serializer) and store it in a file

Parameters

- **target_path** – target path of the file where the iTree should be stored in
- **pack** – True - data will be packed via gzip before storage
- **calc_hash** – True - create the hash information of iTree and store it in the header

- **overwrite** – True - overwrite an existing file
- **itree_serializer** – optional user defined serializer for iTree obbjects

Returns True if file is stored successful

property is_placeholder
Property shows that item is a placeholder class

Normally there should be no placeholder class in the iTree but in case a loaded link does no more contain the expected items it might happen that such a class artifact is still in the tree. In placeholders the value contains the family index in the linked class.

Return type bool

Returns True/False

property is_link_cover
If the item is local and covers a linked item the property is True

Return type bool

Returns True/False

property is_linked
In contrast to iTreeLinked class this is False

Return type bool

Returns True/False

property is_link_loaded

property link_root
delivers the highest level item that is linked in case item is not linked it delivers itself

Return type *iTree*

Returns highest level linked item found in the parents

property is_link_root
property that marks the iTree item as an item that contains a link

Returns

- True - is a link root item
- False is no iTree link item

load_links(*force=False*, *delete_invalid_items=False*, *_items=None*, *_depth=0*)
Runs ove all children and sub children in case a ITreeLink object is found the linked items are load in

In case ´iTree´ is link root: load all linked items

Parameters

- **force** –
  - False (default) - load only if not already loaded
  - True - load even if already loaded (update)
- **delete_invalid_items** –
  - False (default) - in case of invalid items we will raise an exception!
  - True - invalid items will be removed from parent no exception raised
- **_items** – internal list parameter used for recursive calls of the function

- **_depth** – Internal parameter related to current item depth

Returns

- True - success
- False - load failed

make_local(*copy_subtree=True*)
make the current linked object a local object This is only possible if the parent is a iTree object is the link root-> only the first level children in a linked iTree can be made local The operation raises an SyntaxError in case it is used on a deeper level of the linked tree

Returns None

get

getitem_by_idx

3.4 itertree subclasses available in main

3.4.1 .get

This code is taken from the itertree package: https://pypi.org/project/itertree/ GIT Home: https://github.com/BR1py/itertree The documentation can be found here: https://itertree.readthedocs.io/en/latest/index.html

The code is published under MIT license incl. human protect patch:

The MIT License (MIT) incl. human protect patch Copyright © 2022 <copyright holders>

Permission is hereby granted, free of charge, to any person obtaining a copy of this software and associated documentation files (the "Software"), to deal in the Software without restriction, including without limitation the rights to use, copy, modify, merge, publish, distribute, sublicense, and/or sell copies of the Software, and to permit persons to whom the Software is furnished to do so, subject to the following conditions:

Human protect patch: The program and its derivative work will neither be modified or executed to harm any human being nor through inaction permit any human being to be harmed.

The above copyright notice and this permission notice shall be included in all copies or substantial portions of the Software.

THE SOFTWARE IS PROVIDED "AS IS", WITHOUT WARRANTY OF ANY KIND, EXPRESS OR IMPLIED, INCLUDING BUT NOT LIMITED TO THE WARRANTIES OF MERCHANTABILITY, FITNESS FOR A PARTICULAR PURPOSE AND NONINFRINGEMENT. IN NO EVENT SHALL THE AUTHORS OR COPYRIGHT HOLDERS BE LIABLE FOR ANY CLAIM, DAMAGES OR OTHER LIABILITY, WHETHER IN AN ACTION OF CONTRACT, TORT OR OTHERWISE, ARISING FROM, OUT OF OR IN CONNECTION WITH THE SOFTWARE OR THE USE OR OTHER DEALINGS IN THE SOFTWARE.

For more information see: https://en.wikipedia.org/wiki/MIT_License

This part of code contains the specific get methods for the iTree object

The specific getters are quicker compared with the common ones we have in iTree (__getitem__(); get(); get_single())

3.4.2 .deep

This code is taken from the itertree package: https://pypi.org/project/itertree/ GIT Home: https://github.com/BR1py/itertree The documentation can be found here: https://itertree.readthedocs.io/en/latest/index.html

The code is published under MIT license incl. human protect patch:

The MIT License (MIT) incl. human protect patch Copyright © 2022 <copyright holders>

Permission is hereby granted, free of charge, to any person obtaining a copy of this software and associated documentation files (the "Software"), to deal in the Software without restriction, including without limitation the rights to use, copy, modify, merge, publish, distribute, sublicense, and/or sell copies of the Software, and to permit persons to whom the Software is furnished to do so, subject to the following conditions:

Human protect patch: The program and its derivative work will neither be modified or executed to harm any human being nor through inaction permit any human being to be harmed.

The above copyright notice and this permission notice shall be included in all copies or substantial portions of the Software.

THE SOFTWARE IS PROVIDED "AS IS", WITHOUT WARRANTY OF ANY KIND, EXPRESS OR IMPLIED, INCLUDING BUT NOT LIMITED TO THE WARRANTIES OF MERCHANTABILITY, FITNESS FOR A PARTICULAR PURPOSE AND NONINFRINGEMENT. IN NO EVENT SHALL THE AUTHORS OR COPYRIGHT HOLDERS BE LIABLE FOR ANY CLAIM, DAMAGES OR OTHER LIABILITY, WHETHER IN AN ACTION OF CONTRACT, TORT OR OTHERWISE, ARISING FROM, OUT OF OR IN CONNECTION WITH THE SOFTWARE OR THE USE OR OTHER DEALINGS IN THE SOFTWARE.

For more information see: https://en.wikipedia.org/wiki/MIT_License

This part of code contains the main iTree object

3.5 itertree data classes

This code is taken from the itertree package: https://pypi.org/project/itertree/ GIT Home: https://github.com/BR1py/itertree The documentation can be found here: https://itertree.readthedocs.io/en/latest/index.html

The code is published under MIT license incl. human protect patch:

The MIT License (MIT) incl. human protect patch Copyright © 2022 <copyright holders>

Permission is hereby granted, free of charge, to any person obtaining a copy of this software and associated documentation files (the "Software"), to deal in the Software without restriction, including without limitation the rights to use, copy, modify, merge, publish, distribute, sublicense, and/or sell copies of the Software, and to permit persons to whom the Software is furnished to do so, subject to the following conditions:

Human protect patch: The program and its derivative work will neither be modified or executed to harm any human being nor through inaction permit any human being to be harmed.

The above copyright notice and this permission notice shall be included in all copies or substantial portions of the Software.

THE SOFTWARE IS PROVIDED "AS IS", WITHOUT WARRANTY OF ANY KIND, EXPRESS OR IMPLIED, INCLUDING BUT NOT LIMITED TO THE WARRANTIES OF MERCHANTABILITY, FITNESS FOR A PARTICULAR PURPOSE AND NONINFRINGEMENT. IN NO EVENT SHALL THE AUTHORS OR COPYRIGHT HOLDERS BE LIABLE FOR ANY CLAIM, DAMAGES OR OTHER LIABILITY, WHETHER IN AN ACTION OF CONTRACT, TORT OR OTHERWISE, ARISING FROM, OUT OF OR IN CONNECTION WITH THE SOFTWARE OR THE USE OR OTHER DEALINGS IN THE SOFTWARE.

For more information see: https://en.wikipedia.org/wiki/MIT_License

This part of code contains the helper functions related to the *iTree* data attribute

class itertree.itree_data.**iTValueModel**(*value=<class 'itertree.itree_helpers.NoValue'>*, *description=None*, *shape=<class 'itertree.itree_helpers.Any'>*, *contains=None*, *formatter=<class 'str'>*)

Bases: abc.ABC

This is the replacement for the old *iTDataModel()*-class.

This class can be used to define data models for the values that might be placed in the *iTree*.

The model should define min more detail which value objects are accepted or not and it defines also how not matching objects are handled:

- Deny the value and raise a ValueError exception
- Cast the value into a valid value object

But the definition of the data model allows limitations which goes far away from just data-type related topics. E.g. In the model the user can limit numerical values to a specific range (interval) or he might limit strings to specific characters.

All definitions related to checks and type casts must be defined by the user by overwriting the method *check_and_cast_single_item(value_item)* . The return of the method must e the checked and cast value. if the value does not match the method should raise a *ValueError* .

The method should always check and cast a single item, this is important. In case a list or an array like value should be stored in the model the base model will manage the required iteration over the sub-items and perform and utilize the single item check via the user defined method.

Note: In case a value like *[1,2,3,45]* is given each item in the list will be checked and/or casted.

This leads us to an important second definition functionality related to the model related to the size of dimensions (shape) of the value stored in the model. The definition of the shape is given as a parameter when the object is instanced. For the shape we expect a tuple with the dimension information. If a value object is given the maximum shape will be calculated and this will be compared with the expected one. The maximum is used because in nested lists the user can define sub-list with different length. Strings or bytes aare also seen as arrays in this case!

We have the following possibilities to define shapes:

- shape=Any -> accept any shape of the given value (no check performed)
- shape=tuple() -> empty tuple given no dimension expected model will accept single values only!
- shape=(Any,) -> tuple containing one element which is the Any helper class; We will accept single values or any 1 dimensional object here (e.g. values like: *1*; *'abc'*; *[1,2,3,4]*)
- shape=(10,) -> tuple containing one element. We expect one dimensional values with a length lower or equal to the given integer number
- shape=(INF,) -> tuple containing one element that is INF (infinite). We expect one dimensional values of any length
- shape=(3,4) -> Two dimensions expected with fixed size (e.g. [[1],[2,3]] would match)
- shape=(INF,4) -> Two dimensions expected with first length unlimited and second length limited to 3
- shape=(4,ANY) -> Minimum one dimensions expected with first length limited to 4; here the user can also put infinite dimensions in (e.g [1]; [[1],[2]] ; [[[[888],[202,500]]]] would fit)
- shape=(4,ANY,INF) -> 1 dimension or 3 dimensions accepted, 2 dimension will not be accepted

Note: The model base object *iTValueModel()* contains two checking levels. First the user defined check via method definition for checks and casts of single items given. In second step the model also checks the dimension (shape) of the given value.

In case a *str* or *bytes* objects are given the behavior related to the checks will be a bit different as for the other objects. The method *check_and_cast_single_item(value_item)* will target the whole string as a single item! But the shape check will be done also on the string as an object with a length.

This means a string is a 1 dimensional object and the user might limit the size of the string via a shape. (E.g.: The object "Hello" has the shape: (5,); the object ['one', 'two','three'] has the shape: (3,5)) The user might use the method 'get_max_shape()' to measure the shape of objects that is considered in the model base class.

During the instance of the object a formatter can be defined too. This might help the user e.g. do define if an integer value should be converted to a hex or binary representation during string conversion. The build-in command str() of this model class will deliver the formatted value only. The repr() will deliver the class definition.

To use the model the user should put the instanced model object as value in the *iTree*. The real value objects can be placed during object instance via the parameter value or later on via the *set()* method of the model (value exchange too). In case the value is not matching to the model definition an ValueError exception will be raised. If the user like to test first if the value is matching he can use the *in* keyword to check this. In case of no match the exception content might be picked via the last_exception property of the model in this case (might give a hint why the value is not accepted).

Standard Parameters:

Parameters

- **value** – value object to be stored in the model (must match to the model). In case no value is stored in the model (empty model) the value will be NoValue.
- **description** – Description string
- **shape** – Define the dimensions the object should have:
  - None - shape is ignored object might have dimensions or not
  - tuple() - empty tuple or iterable - value object will have no size/dimension
  - (InfShape) - one dimensional value object with infinite size

  (1~~0~~0) * one dimensional value object with max size of 100 items
  - (100,100) - two dimensional object with max size of 100 in each dimension
  - (InfShape,InfShape,InfShape) - three-dimensional object with infinite size in each dimension
  - ...

  Note: For multi-dimensional objects it's recommended to use numpy arrays or objects which have the attribute *shape* representing the size for each dimension available instead of tuples or lists. If not the object performance might be worse (internal iterations required to measure the shape).

- **formatter** – Formatter for the single item of the value object (see string formatting in python) In case no formatter is given *str()* will be used for creation of the item string representation.

check_and_cast_single_item(*value_item*)
method that should be overwritten in the user models

Depending on the requirements the input value might be casted in a target type and he can be checked before or afterwards against check criteria for matches. In case of no match a ValueError should be raised

Except Raise ValueError in case given value does not match

Parameters **value_item** – The value given to the model

Returns casted and checked value

set(*value*)
Set the value of the model in case the value does not match a ValueError exception will be raised

Parameters **value** – value to be placed inside the model

Returns old value stored in the model

get()
get the value that is placed inside the model

If no value is stored in the model the *NoValue*-object will be given back

Returns value stored in the model

property value
property delivering the value stored in the model

Returns value stored in the model

property description
optional description of the model

Returns description related to the model

set_description(*description*)
set/exchange the description of the model

Parameters **description** –

Returns old description

property formatter
get the formatter stored in the model

Returns formatter object

set_formatter(*formatter*)
set the formatter of the object

Parameters **formatter** – The formatter can be a callable method that delivers a str object or a string that contains teh formatting info

Returns old formatter

property contains
contains object stored in the model :return:

clear()
deletes teh value store din the model and place the *NoValue*-object in

property last_except
get the last exception

Returns last exception raised by the model related the storage or check of a value

property is_iTValueModel
used for model identification

Returns True

get_init_args(*full=False*, *clear=False*)
deliver all initial arguments used to instance this model object

Parameters

- **full** – True give always full list False (default) list is shortened in case of default parameter values
- **clear** – True - use NoValue object as value (ignore stored value) False - stored value is included in parameter tuple

Returns Tuple of initial parameters

class itertree.itree_data.**iTAnyValueModel**(*value=<class 'itertree.itree_helpers.NoValue'>*, *description=None*, *shape=<class 'itertree.itree_helpers.Any'>*, *contains=None*, *formatter=<class 'str'>*)
Bases: *itertree.itree_data.iTValueModel*

Model that will take any python object without any restrictions

check_and_cast_single_item(*value_item*)
required overload will allow any object to be stored in the model

Parameters **value_item** – potential value to be stored in the model

Returns confirmed value to be stored in the model

class itertree.itree_data.**iTRoundIntModel**(*value=<class 'itertree.itree_helpers.NoValue'>*, *description=None*, *shape=<class 'itertree.itree_helpers.Any'>*, *contains=None*, *formatter=<class 'str'>*)
Bases: *itertree.itree_data.iTValueModel*

Model that would store integer values The model accepts any object that can be casted into a float and rounded to an integer to be stored as a int in the model

check_and_cast_single_item(*value_item*)
method that should be overwritten in the user models

Depending on the requirements the input value might be casted in a target type and he can be checked before or afterwards against check criteria for matches. In case of no match a ValueError should be raised

Except Raise ValueError in case given value does not match

Parameters **value_item** – The value given to the model

Returns casted and checked value

class itertree.itree_data.**iTIntModel**(*value=<class 'itertree.itree_helpers.NoValue'>*, *description=None*, *shape=<class 'itertree.itree_helpers.Any'>*, *contains=None*, *formatter=<class 'str'>*)
Bases: *itertree.itree_data.iTValueModel*

This integer model allows only integers or strings containing a decimal integer to be stored in the model as int value

check_and_cast_single_item(*value_item*)
method that should be overwritten in the user models

Depending on the requirements the input value might be casted in a target type and he can be checked before or afterwards against check criteria for matches. In case of no match a ValueError should be raised

Except Raise ValueError in case given value does not match

Parameters **value_item** – The value given to the model

Returns casted and checked value

class itertree.itree_data.**iTInt8Model**(*value=<class 'itertree.itree_helpers.NoValue'>*, *description=None*, *shape=<class 'itertree.itree_helpers.Any'>*, *contains=None*, *formatter=<class 'str'>*)

Bases: *itertree.itree_data.iTValueModel*

Integer model that limits the given values to int8 values

interval = mSetInterval(mSetItem(-128), mSetItem(127))

check_and_cast_single_item(*value_item*)
method that should be overwritten in the user models

Depending on the requirements the input value might be casted in a target type and he can be checked before or afterwards against check criteria for matches. In case of no match a ValueError should be raised

Except Raise ValueError in case given value does not match

Parameters **value_item** – The value given to the model

Returns casted and checked value

get_init_args(*full=False*, *clear=False*)
deliver all initial arguments used to instance this model object

Parameters

- **full** – True give always full list False (default) list is shortened in case of default parameter values
- **clear** – True - use NoValue object as value (ignore stored value) False - stored value is included in parameter tuple

Returns Tuple of initial parameters

class itertree.itree_data.**iTUInt8Model**(*value=<class 'itertree.itree_helpers.NoValue'>*, *description=None*, *shape=<class 'itertree.itree_helpers.Any'>*, *contains=None*, *formatter=<class 'str'>*)

Bases: *itertree.itree_data.iTInt8Model*

Integer model that limits the given values to uint8 values

interval = mSetInterval(mSetItem(0), mSetItem(255))

class itertree.itree_data.**iTInt16Model**(*value=<class 'itertree.itree_helpers.NoValue'>*, *description=None*, *shape=<class 'itertree.itree_helpers.Any'>*, *contains=None*, *formatter=<class 'str'>*)

Bases: *itertree.itree_data.iTInt8Model*

Integer model that limits the given values to int16 values

interval = mSetInterval(mSetItem(-32768), mSetItem(32767))

class itertree.itree_data.**iTUInt16Model**(*value=<class 'itertree.itree_helpers.NoValue'>*, *description=None*, *shape=<class 'itertree.itree_helpers.Any'>*, *contains=None*, *formatter=<class 'str'>*)

Bases: *itertree.itree_data.iTInt8Model*

Integer model that limits the given values to uint16 values

interval = mSetInterval(mSetItem(0), mSetItem(65535))

class itertree.itree_data.**iTInt32Model**(*value=<class 'itertree.itree_helpers.NoValue'>*, *description=None*, *shape=<class 'itertree.itree_helpers.Any'>*, *contains=None*, *formatter=<class 'str'>*)

Bases: *itertree.itree_data.iTInt8Model*

Integer model that limits the given values to int32 values

interval = mSetInterval(mSetItem(-2147483648), mSetItem(2147483647))

class itertree.itree_data.**iTUInt32Model**(*value=<class 'itertree.itree_helpers.NoValue'>*, *description=None*, *shape=<class 'itertree.itree_helpers.Any'>*, *contains=None*, *formatter=<class 'str'>*)

Bases: *itertree.itree_data.iTInt8Model*

Integer model that limits the given values to uint32 values

interval = mSetInterval(mSetItem(0), mSetItem(4294967295))

class itertree.itree_data.**iTInt64Model**(*value=<class 'itertree.itree_helpers.NoValue'>*, *description=None*, *shape=<class 'itertree.itree_helpers.Any'>*, *contains=None*, *formatter=<class 'str'>*)

Bases: *itertree.itree_data.iTInt8Model*

Integer model that limits the given values to int64 values

interval = mSetInterval(mSetItem(-9223372036854775808), mSetItem(9223372036854775807))

class itertree.itree_data.**iTUInt64Model**(*value=<class 'itertree.itree_helpers.NoValue'>*, *description=None*, *shape=<class 'itertree.itree_helpers.Any'>*, *contains=None*, *formatter=<class 'str'>*)

Bases: *itertree.itree_data.iTInt8Model*

Integer model that limits the given values to uint64 values

interval = mSetInterval(mSetItem(0), mSetItem(18446744073709551615))

class itertree.itree_data.**iTFloatModel**(*value=<class 'itertree.itree_helpers.NoValue'>*, *description=None*, *shape=<class 'itertree.itree_helpers.Any'>*, *contains=None*, *formatter=<class 'str'>*)

Bases: *itertree.itree_data.iTValueModel*

Float model that allows any float or string that can be casted to float to be stored in the model as float value

check_and_cast_single_item(*value_item*)

method that should be overwritten in the user models

Depending on the requirements the input value might be casted in a target type and he can be checked before or afterwards against check criteria for matches. In case of no match a ValueError should be raised

Except Raise ValueError in case given value does not match

Parameters **value_item** – The value given to the model

Returns casted and checked value

class itertree.itree_data.**iTStrModel**(*value=<class 'itertree.itree_helpers.NoValue'>*, *description=None*, *shape=<class 'itertree.itree_helpers.Any'>*, *contains=None*, *formatter=<class 'str'>*)

Bases: *itertree.itree_data.iTValueModel*

A model to store a string

check_and_cast_single_item(*value_item*)

method that should be overwritten in the user models

Depending on the requirements the input value might be casted in a target type and he can be checked before or afterwards against check criteria for matches. In case of no match a ValueError should be raised

Except Raise ValueError in case given value does not match

Parameters **value_item** – The value given to the model

Returns casted and checked value

class itertree.itree_data.**iTStrFnPatternModel**(*value=<class 'itertree.itree_helpers.NoValue'>*, *description=None*, *shape=<class 'itertree.itree_helpers.Any'>*, *contains=None*, *pattern=None*, *formatter=None*)

Bases: *itertree.itree_data.iTStrModel*

A model to store a string that matches to the fnmatch pattern

property pattern

get_init_args(*full=False*, *clear=False*)

deliver all initial arguments used to instance this model object

Parameters

- **full** – True give always full list False (default) list is shortened in case of default parameter values
- **clear** – True - use NoValue object as value (ignore stored value) False - stored value is included in parameter tuple

Returns Tuple of initial parameters

class itertree.itree_data.**iTStrRegexPatternModel**(*value=<class 'itertree.itree_helpers.NoValue'>*, *description=None*, *shape=<class 'itertree.itree_helpers.Any'>*, *contains=None*, *pattern=None*, *formatter=None*)

Bases: *itertree.itree_data.iTStrModel*

A string model that matches to the regex pattern

property pattern

get_init_args(*full=False*, *clear=False*)

deliver all initial arguments used to instance this model object

Parameters

- **full** – True give always full list False (default) list is shortened in case of default parameter values
- **clear** – True - use NoValue object as value (ignore stored value) False - stored value is included in parameter tuple

Returns Tuple of initial parameters

class itertree.itree_data.**iTASCIIStrModel**(*value=<class 'itertree.itree_helpers.NoValue'>*, *description=None*, *shape=<class 'itertree.itree_helpers.Any'>*, *contains=None*, *formatter=<class 'str'>*)

Bases: *itertree.itree_data.iTValueModel*

A string model that accepts only ASCII characters

check_and_cast_single_item(*value_item*)

method that should be overwritten in the user models

Depending on the requirements the input value might be casted in a target type and he can be checked before or afterwards against check criteria for matches. In case of no match a ValueError should be raised

Except Raise ValueError in case given value does not match

Parameters **value_item** – The value given to the model

Returns casted and checked value

class itertree.itree_data.**iTUTF8StrModel**(*value=<class 'itertree.itree_helpers.NoValue'>*, *description=None*, *shape=<class 'itertree.itree_helpers.Any'>*, *contains=None*, *formatter=<class 'str'>*)

Bases: *itertree.itree_data.iTValueModel*

A string model that accepts only UTF-8 characters

check_and_cast_single_item(*value_item*)

method that should be overwritten in the user models

Depending on the requirements the input value might be casted in a target type and he can be checked before or afterwards against check criteria for matches. In case of no match a ValueError should be raised

Except Raise ValueError in case given value does not match

Parameters **value_item** – The value given to the model

Returns casted and checked value

class itertree.itree_data.**iTUTF16StrModel**(*value=<class 'itertree.itree_helpers.NoValue'>*, *description=None*, *shape=<class 'itertree.itree_helpers.Any'>*, *contains=None*, *formatter=<class 'str'>*)

Bases: *itertree.itree_data.iTStrModel*

A string model that accepts only UTF16 characters

check_and_cast_single_item(*value_item*)

method that should be overwritten in the user models

Depending on the requirements the input value might be casted in a target type and he can be checked before or afterwards against check criteria for matches. In case of no match a ValueError should be raised

Except Raise ValueError in case given value does not match

Parameters **value_item** – The value given to the model

Returns casted and checked value

class itertree.itree_data.**iTEnumerateModel**(*value=<class 'itertree.itree_helpers.NoValue'>*, *enumerate_dict={}*)

Bases: *itertree.itree_data.iTValueModel*

check_and_cast_single_item(*value_item*)

method that should be overwritten in the user models

Depending on the requirements the input value might be casted in a target type and he can be checked before or afterwards against check criteria for matches. In case of no match a ValueError should be raised

Except Raise ValueError in case given value does not match

Parameters **value_item** – The value given to the model

Returns casted and checked value

exception itertree.itree_data.**iTDataValueError**

Bases: ValueError

Exception to be raised in case a validator finds a non matching value related to the iDataModel

exception itertree.itree_data.**iTDataTypeError**

Bases: ValueError

Exception to be raised in case a validator finds a non matching value type related to the iDataModel

class itertree.itree_data.**iTDataModel**(*value=<class 'itertree.itree_helpers.NoValue'>*)

Bases: abc.ABC

The default iTree data model class This the interface definition for specific data model classes that might be created using this superclass

The data model checks the given value for a specific data item. So that we can ensure that the given value matches to the expectations. We can check for types, shapes (length), limits, or matching patterns.

Besides the check we can also define a default formatter for the value that is used when it is translated into a string.

(see examples/itree_data_examples.py)

property is_empty

tells if the iTreeDataModel is empty or contains a value :return:

property is_iTDataModel

get()

the stored value :return: object stored in value

set(*value*)

put a specific value into the data model

Except raises an iTreeValidationError in case a not matching object is given

Parameters **value** – value object to be placed in the data model

property value

the stored value :return: object stored in value

check(*value*)

put a specific value into the data model

Except raises an iTreeValidationError in case a not matching object is given

Parameters **value** – value object to be placed in the data model

clear()
clears (deletes) the current value content and sets the state to "empty"

Returns returns the value object that was stored in the iTreeDataModel

abstract validator(*value*)
This method should check the given value.

It should raise an iDataValueError Exception with a failure explanation in case the value is not matching to the iDataModel.

..warning:: The validator in an explicit iDataModel class must always return the value itself and it must raise the iDataValueError in case of a no matching value. It should also call the super().validator() method or at least consider that *NoValue* is a no matching value.

Except iDataValueError in case value is not matching

Parameters **value** – to be checked against the model

Returns value (which might be casted)

abstract formatter(*value=None*)
The formatter function allows us to create a specific string representation

Especially in case of numerical values this is interesting. You can define here that an integer should be represented always as hex, bin, ... or for floats you can give digits.

The formatter can be created by using the classical format options of string but for enumerations we can put here also a table, etc.

Returns string representing the value

abstract get_init_args()

class itertree.itree_data.**iTDataModelAny**(*value=<class 'itertree.itree_helpers.NoValue'>*)
Bases: *itertree.itree_data.iTDataModel*

Example iDataModel class that accepts any kind of value

validator(*value*)
This method should check the given value.

It should raise an iDataValueError Exception with a failure explanation in case the value is not matching to the iDataModel.

..warning:: The validator in an explicit iDataModel class must always return the value itself and it must raise the iDataValueError in case of a no matching value. It should also call the super().validator() method or at least consider that *NoValue* is a no matching value.

Except iDataValueError in case value is not matching

Parameters **value** – to be checked against the model

Returns value (which might be casted)

formatter(*value=None*)
The formatter function allows us to create a specific string representation

Especially in case of numerical values this is interesting. You can define here that an integer should be represented always as hex, bin, ... or for floats you can give digits.

The formatter can be created by using the classical format options of string but for enumerations we can put here also a table, etc.

Returns string representing the value

get_init_args()

class itertree.itree_data.**iTData**(*seq=None*, ***kwargs*)
Bases: dict

Standard itertree Data management object might be overloaded or changed by the user

GET_LOOK_UP_METHOD = {0: <function iTData.<lambda>>, 1: <function iTData.<lambda>>,

update(*E=None*, ***F*)
function update of multiple items if one item is invalid the whole update will be skipped and an iDataValueError exception will thrown!

In case the replace_model flag is set the model will be exchanged.

Parameters taken from builtin dict:

Update D from dict/iterable E and F. If E is present and has a .keys() method, then does: If E is present and lacks a .keys() method, then does: In either case, this is followed by:

Except raises iDataValueError exception if a value in the given object is not matching to the data-model. The iData object will not be updated in this case.

Parameters

- **E** –
  - with .keys() method: for k in E: D[k] = E[k]
  - without .keys() method: for k, v in E: D[k] = v
- ****F** – we run: for k in F: D[k] = F[k]
- **replace_models** –
  - True - Will replace the whole key related value (also iTDataModels are replaced)
  - **False (default) - All values are replaced in case of iTDataModel object the internal value will** be replaced

copy()
create a new object with same items

Returns new object copied from self

clear() → None. Remove all items from D.

pop(*key=<class 'itertree.itree_helpers.NoKey'>*, *default=<class 'itertree.itree_helpers.NoKey'>*, *value_only=True*)
delete a stored value

Except will case KeyError if key is not found and default is not set

Parameters

- **key** – key where the item should be popped out
- **value_only** – True - only value will be deleted model will be kept in iTreeData False - whole model will be popped out

Default define the value given back in case key is not found else KeyError will be raised

Returns deleted item or default

get(*key=<class 'itertree.itree_helpers.NoKey'>*, *default=None*, *return_type=0*)
get a specific data item by key

Parameters

- **key** – key of the data item (if not given __NOKEY__ is used)
- **default** – default value that will be delivered in case of no match
- **_return_type** – We can deliver different returns * VALUE - value object * FULL - iTreeDataModel (only if used else same as VALUE) * STR - formatted string representation of the data value

Returns requested value

fromkeys(**args*, ***kwargs*)
create a new iData object based on given keys and optional value

- real signature unknown

delete_item(*key*, *value_only=True*)
delete a item by key

Except KeyError is raised in case item key is unknown

Parameters

- **key** – key of the data item (if not given __NOKEY__ is used!
- **value_only** –
  - **True - (default) in case of iDataModel items we delete only the internal value** not the model itself
  - False - we delete the value independent from the type (also iDataModel objects)

Returns deleted value

model_values()
iterator that takes in case of iDataModel values the value out of the model, in case of non iDataModel values the value is given directly as it is

Returns iterator

model_items()
iterator that takes in case of iDataModel values the value out of the model, in case of non iDataModel values the value is given directly as it is

Returns iterator

property is_empty
used for identification of this class :return: True

property is_no_key_only
used for identification of this class :return: True

property is_iTData

is_key_empty(*key=<class 'itertree.itree_helpers.NoKey'>*)
Function delivers a key empty state (it delivers True in case key is absent or value is __NOVALUE__ :param key: key to be check (delault is __NOKEY__ :return: True/False

deepcopy()
create a deep copy of this object

also all internal items will be copied!

Returns new object deep copied from self

get_init_args()

class itertree.itree_data.**iTDataReadOnly**(*seq=None*, ***kwargs*)

Bases: *itertree.itree_data.iTData*

Standard itertree Data management object might be overloaded or changed by the user

pop(**arg*, ***kwargs*)

delete a stored value

Except will case KeyError if key is not found and default is not set

Parameters

- **key** – key where the item should be popped out
- **value_only** – True - only value will be deleted model will be kept in iTreeData
  False - whole model will be popped out

Default define the value given back in case key is not found else KeyError will be raised

Returns deleted item or default

update(**arg*, ***kwargs*)

function update of multiple items if one item is invalid the whole update will be skipped and an iDataValueError exception will thrown!

In case the replace_model flag is set the model will be exchanged.

Parameters taken from builtin dict:

Update D from dict/iterable E and F. If E is present and has a .keys() method, then does: If E is present and lacks a .keys() method, then does: In either case, this is followed by:

Except raises iDataValueError exception if a value in the given object is not matching to the data-model. The iData object will not be updated in this case.

Parameters

- **E** –
  - with .keys() method: for k in E: D[k] = E[k]
  - without .keys() method: for k, v in E: D[k] = v
- ****F** – we run: for k in F: D[k] = F[k]
- **replace_models** –
  - True - Will replace the whole key related value (also iTDataModels are replaced)
  - **False (default) - All values are replaced in case of iTDataModel object the internal value will** be replaced

clear() → None. Remove all items from D.

delete_item(*key*, *value_only=True*)

delete a item by key

Except KeyError is raised in case item key is unknown

Parameters

- **key** – key of the data item (if not given __NOKEY__ is used!
- **value_only** –
  - **True - (default) in case of iDataModel items we delete only the internal value** not the model itself

– False - we delete the value independent from the type (also iDataModel objects)

Returns deleted value

get_init_args()

3.6 itertree filter classes

This code is taken from the itertree package: https://pypi.org/project/itertree/ GIT Home: https://github.com/BR1py/itertree The documentation can be found here: https://itertree.readthedocs.io/en/latest/index.html

The code is published under MIT license incl. human protect patch:

The MIT License (MIT) incl. human protect patch Copyright © 2022 <copyright holders>

Permission is hereby granted, free of charge, to any person obtaining a copy of this software and associated documentation files (the "Software"), to deal in the Software without restriction, including without limitation the rights to use, copy, modify, merge, publish, distribute, sublicense, and/or sell copies of the Software, and to permit persons to whom the Software is furnished to do so, subject to the following conditions:

Human protect patch: The program and its derivative work will neither be modified or executed to harm any human being nor through inaction permit any human being to be harmed.

The above copyright notice and this permission notice shall be included in all copies or substantial portions of the Software.

THE SOFTWARE IS PROVIDED "AS IS", WITHOUT WARRANTY OF ANY KIND, EXPRESS OR IMPLIED, INCLUDING BUT NOT LIMITED TO THE WARRANTIES OF MERCHANTABILITY, FITNESS FOR A PARTICULAR PURPOSE AND NONINFRINGEMENT. IN NO EVENT SHALL THE AUTHORS OR COPYRIGHT HOLDERS BE LIABLE FOR ANY CLAIM, DAMAGES OR OTHER LIABILITY, WHETHER IN AN ACTION OF CONTRACT, TORT OR OTHERWISE, ARISING FROM, OUT OF OR IN CONNECTION WITH THE SOFTWARE OR THE USE OR OTHER DEALINGS IN THE SOFTWARE.

For more information see: https://en.wikipedia.org/wiki/MIT_License

This part of code contains the iTree filter classes

we use here lambda to create a method which is feed with an item and delivers then True/False depending on the given condition so that it can be used in filter iterators

itertree.itree_filters.**iter_items_over_filter_method**(*filter_method*, *item_iter*)

helper function that delivers an iterator of True/False based on given filter_method and the item iterator given

Parameters

- **filter_method** – Item filter method
- **item_iter** – iterable where each item should be checked against the filter

Returns iterator of True/False objects matches to filter or not

class itertree.itree_filters.**has_item_flags**(*flag_mask*, *invert=False*)

Bases: object

Check the iTree flags for match to the given flag mask

Parameters

- **item** – *iTree*-item to be checked against the criteria of the method (for filtering out or not)
- **flag_mask** – flag mask E.g. can be build like: iTFLAG.READ_ONLY_TREE|iTFLAG.READ_ONLY_VALUE

Return type bool

Returns

- True -> match
- False -> no match

class itertree.itree_filters.**is_item_tag**(*target_tag*, *invert=False*)
Bases: object

Check the iTree tag is equal to the given target_tag

Parameters

- **target_tag** – tag string do not give Tag() objects here! Use Tag().tag if really required
- **invert** –
  - False (default) -> unchanged result
  - True -> invert the result True->False; False->True

class itertree.itree_filters.**has_item_tag_fnmatch**(*tag_match_pattern*, *invert=False*)
Bases: object

Check the iTree tag is matching to given fnmatch match_pattern

Parameters **match_pattern** – str or bytes related to fnmatch pattern definitions

class itertree.itree_filters.**has_item_value**(*target_value*, *invert=False*)
Bases: object

Check the iTree value is equal to given value

Parameters

- **target_value** – value object that should be equal with iTree.value
- **invert** –
  - False (default) -> unchanged result
  - True -> invert the result True->False; False->True

class itertree.itree_filters.**has_item_value_dict_value**(*target_value*, *invert=False*)
Bases: object

Check if in case the iTree value is a dict a value in the dict is equal to given value

Parameters

- **target_value** – value object that should be equal with iTree.value
- **invert** –
  - False (default) -> unchanged result
  - True -> invert the result True->False; False->True

class itertree.itree_filters.**has_item_value_list_value**(*target_value*, *invert=False*)
Bases: object

Check if in case the iTree value is a list a value in the list is equal to given value

Parameters

- **target_value** – value object that should be equal with iTree.value
- **invert** –

- False (default) -> unchanged result
- True -> invert the result True->False; False->True

class itertree.itree_filters.**has_item_value_fnmatch**(*target_value_pattern*, *invert=False*)

Bases: object

Check if value matches to the given fnmatch pattern

Parameters

- **target_value_pattern** – str or bytes related to fnmatch pattern definitions
- **invert** –
  - False (default) -> unchanged result
  - True -> invert the result True->False; False->True

class itertree.itree_filters.**has_item_value_dict_value_fnmatch**(*target_value_pattern*, *invert=False*)

Bases: object

Check if in case the iTree value is a dict a value in the dict matches to the given pattern

Parameters

- **target_value_pattern** – str or bytes related to fnmatch pattern definitions
- **invert** –
  - False (default) -> unchanged result
  - True -> invert the result True->False; False->True

class itertree.itree_filters.**has_item_value_list_item_fnmatch**(*target_value_pattern*, *invert=False*)

Bases: object

Check if in case the iTree value is a list a value in the list matches to the given pattern

Parameters

- **target_value_pattern** – str or bytes related to fnmatch pattern definitions
- **invert** –
  - False (default) -> unchanged result
  - True -> invert the result True->False; False->True

class itertree.itree_filters.**is_item_value_in**(*target_value_interval*, *invert=False*)

Bases: object

Check if iTree value is in the given iTInterval object, no numeric values will be ignored

Parameters

- **target_key_interval** – msetInterval object defining the range (any object that supports "in" can be used)
- **invert** –
  - False (default) -> unchanged result
  - True -> invert the result True->False; False->True

class itertree.itree_filters.**has_item_value_dict_value_in**(*target_value_interval*, *invert=False*)

Bases: object

Check if in case the iTree value is a dict a value in the dict is in the given iTInterval object, no numeric values will be ignored

Parameters

- **target_key_interval** – msetInterval object defining the range (any object that supports "in" can be used)
- **invert** –
  - False (default) -> unchanged result
  - True -> invert the result True->False; False->True

class itertree.itree_filters.**has_item_value_list_item_in**(*target_value_interval*, *invert=False*)

Bases: object

Check if in case the iTree value is a list a value in the list is in the given iTInterval object, non numeric values will be ignored

Parameters

- **target_key_interval** – msetInterval object defining the range (any object that supports "in" can be used)
- **invert** –
  - False (default) -> unchanged result
  - True -> invert the result True->False; False->True

class itertree.itree_filters.**has_item_value_dict_key**(*target_key*, *invert=False*)

Bases: object

Check if in case the iTree value is a dict a key in the dict is equal with the given target_key no numeric values will be ignored

Parameters **target_key** – dict key

class itertree.itree_filters.**has_item_value_list_idx**(*target_idx*, *invert=False*)

Bases: object

Check if in case the iTree value is a list the given target_key is lower than list length (inside) no numeric values will be ignored

Parameters

- **target_idx** – target-index
- **invert** –
  - False (default) -> unchanged result
  - True -> invert the result True->False; False->True

class itertree.itree_filters.**has_item_value_dict_key_fnmatch**(*target_key_pattern*, *invert=False*)

Bases: object

Check if in case the iTree value is a dict a key in the dict matches to the given key pattern (fnmatch) no numeric values will be ignored

Parameters

- **target_key_pattern** – str or bytes related to fnmatch pattern definitions
- **invert** –
  - False (default) -> unchanged result
  - True -> invert the result True->False; False->True

class itertree.itree_filters.**has_item_value_dict_key_in**(*target_key_interval*, *invert=False*)

Bases: object

Check if in case the iTree value is a dict a key in the dict is in the given iTInterval object range no numeric values will be ignored

Parameters

- **target_key_interval** – msetInterval object defining the range (any object that supports "in" can be used)
- **invert** –
  - False (default) -> unchanged result
  - True -> invert the result True->False; False->True

This code is taken from the itertree package: https://pypi.org/project/itertree/ GIT Home: https://github.com/BR1py/itertree The documentation can be found here: https://itertree.readthedocs.io/en/latest/index.html

The code is published under MIT license incl. human protect patch:

The MIT License (MIT) incl. human protect patch Copyright © 2022 <copyright holders>

Permission is hereby granted, free of charge, to any person obtaining a copy of this software and associated documentation files (the "Software"), to deal in the Software without restriction, including without limitation the rights to use, copy, modify, merge, publish, distribute, sublicense, and/or sell copies of the Software, and to permit persons to whom the Software is furnished to do so, subject to the following conditions:

Human protect patch: The program and its derivative work will neither be modified or executed to harm any human being nor through inaction permit any human being to be harmed.

The above copyright notice and this permission notice shall be included in all copies or substantial portions of the Software.

THE SOFTWARE IS PROVIDED "AS IS", WITHOUT WARRANTY OF ANY KIND, EXPRESS OR IMPLIED, INCLUDING BUT NOT LIMITED TO THE WARRANTIES OF MERCHANTABILITY, FITNESS FOR A PARTICULAR PURPOSE AND NONINFRINGEMENT. IN NO EVENT SHALL THE AUTHORS OR COPYRIGHT HOLDERS BE LIABLE FOR ANY CLAIM, DAMAGES OR OTHER LIABILITY, WHETHER IN AN ACTION OF CONTRACT, TORT OR OTHERWISE, ARISING FROM, OUT OF OR IN CONNECTION WITH THE SOFTWARE OR THE USE OR OTHER DEALINGS IN THE SOFTWARE. For more information see: https://en.wikipedia.org/wiki/MIT_License

This part of code contains helper classes used in DataTree object

class itertree.itree_mathsets.**mSetItem**(*value*, *complement=False*, *formatter=<class 'str'>*)

Bases: object

item object to used in the different mSet objects Depending on the object it is a "normal" item (*mSetRoster*) or it is the lower or upper limit of the *mSetInterval* object. In first case complemented items will be ignored.

The object contains a formatting option to define how the string representation of the item should look like. Especially integer formatting is used to to create representations of the other base like hex or octal.

Parameters

- **value** (*object*) – numerical value to be stored in object or definition str the user can give also a variable-name here
- **complement** (*bool*) – complement type item (only required for interval limits
- **formatter** – explicit formatter user can give as formatter 1. formatter method (callable) 2. escape string for *format()* method 3 escape string for classical "%" replacement

property is_mSetItem
Property used for identification of the object

Return type bool

Returns True

property is_complement
property tells if the object is complement

Return type bool

Returns True/False

property value
property contains the value of the item :return: value of the object

property formatter
property delivers the formatter of the object

Return type Union[Callable,str]

Returns formatter object (Callable or string)

property formatter_type
property delivers the formatter type integer constant of the object formatter

Return type int

Returns formatter type integer (`_CALL`~0;,`_FORMAT`~1; ,`_OLD`~2)

property is_var
property returns if the value is a variable name or a normal numerical value

Return type bool

Returns True - is variable; False - normal numerical value

get_init_args (*full=False*)
get the initial arguments for instance the object

Parameters **full** (*bool*) – do not shorten even that we have default values

Return type tuple

Returns tuple of init arguments

math_repr (*formatter=None*)
delivers the formatted value :rtype formatter: Union[Callable,str,None][:param formatter: optionally an explicit formatter can be given

Return type str

Returns Formatted value stored in the *mSetItem*-object

class itertree.itree_mathsets.**mSetInterval**(**definition*, *lower=None*, *upper=None*, *int_only=False*, *complement=False*)
Bases: itertree.itree_mathsets._mSetBase

Mathematical interval set object. Here the user can define a mathematical interval with closed or open boarders.

For more details related to mathematical intervals you may have a look here: https://en.wikipedia.org/wiki/Interval_(mathematics)

property is_lower_closed
do we have a closed lower limit "(" :return: True is closed, False is open

property is_lower_open
do we have a open lower limit "(" :return: True is open, False is closed

property lower_value
Property delivers the lower limit value :return: value of the lower limit

property is_upper_closed
do we have a closed upper limit "(" :return: True is closed, False is open

property is_upper_open
do we have a open upper limit "(" :return: True is open, False is closed

property upper_value
Property delivers the upper limit value :return: value of the upper limit

property is_int_only
Is this an integer number only interval? :return:

property cardinality
The cardinality is somehow the size of the set it delivers how many items the set contains. The result is not in all cases correct furthermore it is just an estimation!

In many cases in float intervals the user will find infinite as the result of the operation. :return: number of items integer or float('inf') for infinite results

property is_empty_set
Is the interval same as an empty set (no item inside) :return: True is empty; False is not empty

property is_empty_set_complement
Is the complement interval same as an empty set (no item inside). If this is the case the set is the universal set (any item inside). But this is a relative definition to the "universe". E.g. strings will never be found inside a numerical set they are not in the "universe".

Returns True complement is empty; False complement is not empty

iter_in(*value*, *vars_dict=None*)
For each item in the given iterable value we check if the item is in this mSet object the result is a iterable over the single results :param value: to be checked iterable value (single item check) :param vars_dict: variable replacement dict :return: iterable True/False

filter(*value*, *vars_dict=None*)
For each item in the given iterable value we check if the item is in this mSet object or not in case it is in the item will be delivered back if not it is skipped

Parameters

- **value** – iterable value which items will be checked
- **vars_dict** – variable replacement dict

Returns iterable of matching items

get_init_args(*full=False*)
delivers tuple of all initial arguments given to instance the mSet object :param full: True all arguments given also defaults :return: tuple of initial arguments

math_repr(*formatters=None*)
mathematical representation of the object (we try to match as good as possible to the mathematical standards here but we avoid exotic characters! :return: mathematical representation string

class itertree.itree_mathsets.**mSetRoster**(**definition*, *items=None*, *complement=False*)
Bases: itertree.itree_mathsets._mSetBase

property cardinality
The cardinality is somehow the size of the set it delivers how many items the set contains. The result is not in all cases correct furthermore it is just an estimation!

In many cases in float intervals the user will find infinite as the result of the operation. :return: number of items integer or float('inf') for infinite results

property is_empty_set
For some set definition no matching item can be found! Then the set is equal to the empty set and this property will deliver True

Return type bool

Returns True is empty set; False set contains items

property is_empty_set_complement
Is the complement interval same as an empty set (no item inside). If this is the case the set is the universal set (any item inside). But this is a relative definition to the "universe". E.g. strings will never be found inside a numerical set they are not in the "universe".

Returns True complement is empty; False complement is not empty

items()

iter_in(*value*, *vars_dict=None*)
For each item in the given iterable value we check if the item is in this mSet object the result is a iterable over the single results :param value: to be checked iterable value (single item check) :param vars_dict: variable replacement dict :return: iterable True/False

filter(*value*, *vars_dict=None*)
For each item in the given iterable value we check if the item is in this mSet object or not in case it is in the item will be delivered back if not it is skipped

Parameters

- **value** – iterable value which items will be checked
- **vars_dict** – variable replacement dict

Returns iterable of matching items

get_init_args(*full=False*)
delivers tuple of all initial arguments given to instance the mSet object :param full: True all arguments given also defaults :return: tuple of initial arguments

math_repr(*formatters=None*)
mathematical representation of the object (we try to match as good as possible to the mathematical standards here but we avoid exotic characters! :return: mathematical representation string

class itertree.itree_mathsets.**mSetCombine**(**definition*, *is_union=True*, *complement=False*)
Bases: itertree.itree_mathsets._mSetBase

class where the user can combine different sets to unions

In this class the user can combine different types of sets (all objects with *__contains__()* and a length are allowed to be added.

If the object is used to check if a value is in it is sufficient if the value is in one of the subsets to create a positive response for a match

property is_union

property is_intersection

items()

property cardinality
The cardinality is somehow the size of the set it delivers how many items the set contains. The result is not in all cases correct furthermore it is just an estimation!

Especially in this case the cardinally is really an estimation only. It's not teh case that we check here for overlapping intervals which might reduce the cardinality. We create just an estimation based the cardinalities of the subitems

In many cases in float intervals the user will find infinite as the result of the operation. :return: number of items integer or float('inf') for infinite results

property is_empty_set
For some set definition no matching item can be found! Then the set is equal to the empty set and this property will deliver True

Return type bool

Returns True is empty set; False set contains items

property is_empty_set_complement
Is the complement interval same as an empty set (no item inside). If this is the case the set is the universal set (any item inside). But this is a relative definition to the "universe". E.g. strings will never be found inside a numerical set they are not in the "universe".

Returns True complement is empty; False complement is not empty

math_repr()
mathematical representation of the object (we try to match as good as possible to the mathematical standards here but we avoid exotic characters! :return: mathematical representation string

iter_in(*value*, *vars_dict=None*)
For each item in the given iterable value we check if the item is in this mSet object the result is a iterable over the single results :param value: to be checked iterable value (single item check) :param vars_dict: variable replacement dict :return: iterable True/False

filter(*value*, *vars_dict=None*)
For each item in the given iterable value we check if the item is in this mSet object or not in case it is in the item will be delivered back if not it is skipped

Parameters

- **value** – iterable value which items will be checked
- **vars_dict** – variable replacement dict

Returns iterable of matching items

get_init_args(*full=False*)
delivers tuple of all initial arguments given to instance the mSet object :param full: True all arguments given also defaults :return: tuple of initial arguments

3.7 itertree serializing

This code is taken from the itertree package: https://pypi.org/project/itertree/ GIT Home: https://github.com/BR1py/itertree The documentation can be found here: https://itertree.readthedocs.io/en/latest/index.html

The code is published under MIT license incl. human protect patch:

The MIT License (MIT) incl. human protect patch Copyright © 2022 <copyright holders>

Permission is hereby granted, free of charge, to any person obtaining a copy of this software and associated documentation files (the "Software"), to deal in the Software without restriction, including without limitation the rights to use, copy, modify, merge, publish, distribute, sublicense, and/or sell copies of the Software, and to permit persons to whom the Software is furnished to do so, subject to the following conditions:

Human protect patch: The program and its derivative work will neither be modified or executed to harm any human being nor through inaction permit any human being to be harmed.

The above copyright notice and this permission notice shall be included in all copies or substantial portions of the Software.

THE SOFTWARE IS PROVIDED "AS IS", WITHOUT WARRANTY OF ANY KIND, EXPRESS OR IMPLIED, INCLUDING BUT NOT LIMITED TO THE WARRANTIES OF MERCHANTABILITY, FITNESS FOR A PARTICULAR PURPOSE AND NONINFRINGEMENT. IN NO EVENT SHALL THE AUTHORS OR COPYRIGHT HOLDERS BE LIABLE FOR ANY CLAIM, DAMAGES OR OTHER LIABILITY, WHETHER IN AN ACTION OF CONTRACT, TORT OR OTHERWISE, ARISING FROM, OUT OF OR IN CONNECTION WITH THE SOFTWARE OR THE USE OR OTHER DEALINGS IN THE SOFTWARE.

For more information see: https://en.wikipedia.org/wiki/MIT_License

This part of code contains the standard iTree serializers (JSON and rendering)

class itertree.itree_serializer.itree_json_serialize.**iTStdJSONSerializer2**(*itree_class*)

Bases: object

This is the standard serializer for DataTree which translates the structure into the JSON format. Users might implement their own serializers using the interface methods defined in this serializer

not_linked_filter()

ALL_TYPE = 0

BYTE_TYPE = 1

STR_TYPE = 2

DICT_TYPE = 3

ITER_TYPE = 4

NP_TYPE = 5

IT_TYPE = 6

IT_LINK_TYPE = 7

IT_CONST_TYPE = 8

NO_VALUE = 0

NO_TAG = 1

ANY = 2

NO_KEY = 3

TRANSLATE_OBJ2KEY = {<class 'itertree.itree_helpers.Any'>: 2, <class 'itertree.itree_

TRANSLATE_KEY2OBJ = {0: <class 'itertree.itree_helpers.NoValue'>, 1: <class 'itertre

CONVERT_MAP = {<class 'collections.deque'>: <function iTStdJSONSerializer2.<lambda>>,

CONVERT_FROM_JSON_MAP = {0: <function iTStdJSONSerializer2.<lambda>>, 1: <function i'

convert_to_json_item(*o*)

convert_numpy(*data*, *dtype*, *shape*)

convert_it_type(*o*)

convert_from_json_obj(*json_obj*)

convert_single_itree_to_json_obj(*depth*, *itree*, *fidx*)

convert_single_itree_to_json_obj2(*depth*, *itree*, *fidx*)

dumps(*o*, *add_header=False*, *calc_hash=False*, *filter_method=None*)

In JSON the iTree object is represented in the following form Item-> dict with all properties (Special keys used) Tree structure is stored in list

Parameters

- **o** – iTree object to be serialized
- **add_header** – True - the header information will be added (containing Version info and hash) False - no header pure data
- **calc_hash** – True - A sha1 hash is calculated over the data section of iTree and added in the header False - no hash will be calculated

Return type tuple

Returns hash,string containing the serialized data -> if no hash calculation requested hash will be None

dump(*o*, *file_path*, *pack=True*, *calc_hash=True*, *overwrite=False*, *filter_method=None*)

Serialize iTree object into a file

Parameters

- **o** – iTree object to be serialized
- **file_path** – target file path where to store the data in
- **pack** – True - gzip the data, False - do not zip
- **overwrite** – True - an existing fie will be overwritten False (default) - in case the file exists an FileExistsError Exception will be raised
- **calc_hash** – True - A sha1 hash is calculated over the data section of iTree and added in the header False - no hash will be calculated

Returns None

create_itree_from_raw(*raw_o*)

create_itree_from_raw2(*raw_o*)

loads(*source_str*, *check_hash=True*, *load_links=True*, *_source=None*)

create an iTree object by loading from a string.

Parameters

- **source_str** – source string that contains the iTree information

- **check_hash** – True the hash of the file will be checked and the loading will be stopped if it doesn't match False - do not check the iTree hash
- **load_links** – True - linked iTree objects will be loaded
- **_source** – Path of a loaded source file (for internal use)

Returns iTree object loaded from file

load(*file_path*, *check_hash=True*, *load_links=True*)
create an iTree object by loading from a file

Parameters

- **file_path** – file path to the file that contains the iTree information
- **check_hash** – True the hash of the file will be checked and the loading will be stopped if it doesn't match False - do not check the iTree hash
- **load_links** – True - linked iTree objects will be loaded

Returns iTree object loaded from file

This code is taken from the itertree package: https://pypi.org/project/itertree/ GIT Home: https://github.com/BR1py/itertree The documentation can be found here: https://itertree.readthedocs.io/en/latest/index.html

The code is published under MIT license incl. human protect patch:

The MIT License (MIT) incl. human protect patch Copyright © 2022 <copyright holders>

Permission is hereby granted, free of charge, to any person obtaining a copy of this software and associated documentation files (the "Software"), to deal in the Software without restriction, including without limitation the rights to use, copy, modify, merge, publish, distribute, sublicense, and/or sell copies of the Software, and to permit persons to whom the Software is furnished to do so, subject to the following conditions:

Human protect patch: The program and its derivative work will neither be modified or executed to harm any human being nor through inaction permit any human being to be harmed.

The above copyright notice and this permission notice shall be included in all copies or substantial portions of the Software.

THE SOFTWARE IS PROVIDED "AS IS", WITHOUT WARRANTY OF ANY KIND, EXPRESS OR IMPLIED, INCLUDING BUT NOT LIMITED TO THE WARRANTIES OF MERCHANTABILITY, FITNESS FOR A PARTICULAR PURPOSE AND NONINFRINGEMENT. IN NO EVENT SHALL THE AUTHORS OR COPYRIGHT HOLDERS BE LIABLE FOR ANY CLAIM, DAMAGES OR OTHER LIABILITY, WHETHER IN AN ACTION OF CONTRACT, TORT OR OTHERWISE, ARISING FROM, OUT OF OR IN CONNECTION WITH THE SOFTWARE OR THE USE OR OTHER DEALINGS IN THE SOFTWARE.

For more information see: https://en.wikipedia.org/wiki/MIT_License

This part of code contains the standard iTree serializers (JSON and rendering)

class itertree.itree_serializer.itree_renderer.**iTreeRender**
Bases: object

Standard renderer fr the iTree object for creating a very simple pretty print output

renders(*itree_object*, *filter_method=None*, *enumerate=False*)

creates a pretty print string from iTree object and returns it in a string

The rendered outputs can be filtered but only in hierarchical way.

param itree_object iTree object to be converted

param filter_method item filter method or filter-constant to filter specific items out
Note:: The root of the object is not filtered and always in the outputs first line

Parameters **enumerate** – add enumeration before the items

return string containing the pretty print output

This code is taken from the itertree package: https://pypi.org/project/itertree/ GIT Home: https://github.com/BR1py/itertree The documentation can be found here: https://itertree.readthedocs.io/en/latest/index.html

The code is published under MIT license incl. human protect patch:

The MIT License (MIT) incl. human protect patch Copyright © 2022 <copyright holders>

Permission is hereby granted, free of charge, to any person obtaining a copy of this software and associated documentation files (the "Software"), to deal in the Software without restriction, including without limitation the rights to use, copy, modify, merge, publish, distribute, sublicense, and/or sell copies of the Software, and to permit persons to whom the Software is furnished to do so, subject to the following conditions:

Human protect patch: The program and its derivative work will neither be modified or executed to harm any human being nor through inaction permit any human being to be harmed.

The above copyright notice and this permission notice shall be included in all copies or substantial portions of the Software.

THE SOFTWARE IS PROVIDED "AS IS", WITHOUT WARRANTY OF ANY KIND, EXPRESS OR IMPLIED, INCLUDING BUT NOT LIMITED TO THE WARRANTIES OF MERCHANTABILITY, FITNESS FOR A PARTICULAR PURPOSE AND NONINFRINGEMENT. IN NO EVENT SHALL THE AUTHORS OR COPYRIGHT HOLDERS BE LIABLE FOR ANY CLAIM, DAMAGES OR OTHER LIABILITY, WHETHER IN AN ACTION OF CONTRACT, TORT OR OTHERWISE, ARISING FROM, OUT OF OR IN CONNECTION WITH THE SOFTWARE OR THE USE OR OTHER DEALINGS IN THE SOFTWARE.

For more information see: https://en.wikipedia.org/wiki/MIT_License

This is a class wihch translates an iTree in a dot graph

This part of code contains the standard iTree serializers (JSON and rendering)

This code is taken from the itertree package: https://pypi.org/project/itertree/ GIT Home: https://github.com/BR1py/itertree The documentation can be found here: https://itertree.readthedocs.io/en/latest/index.html

The code is published under MIT license incl. human protect patch:

The MIT License (MIT) incl. human protect patch Copyright © 2022 <copyright holders>

Permission is hereby granted, free of charge, to any person obtaining a copy of this software and associated documentation files (the "Software"), to deal in the Software without restriction, including without limitation the rights to use, copy, modify, merge, publish, distribute, sublicense, and/or sell copies of the Software, and to permit persons to whom the Software is furnished to do so, subject to the following conditions:

Human protect patch: The program and its derivative work will neither be modified or executed to harm any human being nor through inaction permit any human being to be harmed.

The above copyright notice and this permission notice shall be included in all copies or substantial portions of the Software.

THE SOFTWARE IS PROVIDED "AS IS", WITHOUT WARRANTY OF ANY KIND, EXPRESS OR IMPLIED, INCLUDING BUT NOT LIMITED TO THE WARRANTIES OF MERCHANTABILITY, FITNESS FOR A PARTICULAR PURPOSE AND NONINFRINGEMENT. IN NO EVENT SHALL THE AUTHORS OR COPYRIGHT HOLDERS BE LIABLE FOR ANY CLAIM, DAMAGES OR OTHER LIABILITY, WHETHER IN AN ACTION OF CONTRACT, TORT OR OTHERWISE, ARISING FROM, OUT OF OR IN CONNECTION WITH THE SOFTWARE OR THE USE OR OTHER DEALINGS IN THE SOFTWARE.

For more information see: https://en.wikipedia.org/wiki/MIT_License

This part of code contains the standard iTree serializers (JSON and rendering)

itertree.itree_serializer.itree_json_converter.**Converter_1_1_1_to_2_0_0**(*src_path*, *check_hash=True*)

class itertree.itree_serializer.itree_json_converter.**Converter_1_1_1_to_2_0_0_Cls**

Bases: object

This is the standard serializer for DataTree which translates the structure into the JSON format. Users might implement their own serializers using the interface methods defined in this serializer

ITREE_SERIALIZE_VERSION = '1.1.1'

TREE = 'iT'

DATA = 'DT'

LINK = 'LK'

TAG = 'TG'

IDX = 'IDX'

DATA_MODELL = 'DM'

DTYPE = 'TP'

DATA_CONTAINER = 'DC'

ITREE_ITEMS_DECODE = {'iT', 'iTI', 'iTPH', 'iTRO', 'iTl'}

convert(*src_path*, *check_hash=True*)

create_itree_from_raw(*raw_o*)

3.8 itertree helper classes

This code is taken from the itertree package: https://pypi.org/project/itertree/ GIT Home: https://github.com/BR1py/itertree The documentation can be found here: https://itertree.readthedocs.io/en/latest/index.html

The code is published under MIT license incl. human protect patch:

The MIT License (MIT) incl. human protect patch Copyright © 2022 <copyright holders>

Permission is hereby granted, free of charge, to any person obtaining a copy of this software and associated documentation files (the "Software"), to deal in the Software without restriction, including without limitation the rights to use, copy, modify, merge, publish, distribute, sublicense, and/or sell copies of the Software, and to permit persons to whom the Software is furnished to do so, subject to the following conditions:

Human protect patch: The program and its derivative work will neither be modified or executed to harm any human being nor through inaction permit any human being to be harmed.

The above copyright notice and this permission notice shall be included in all copies or substantial portions of the Software.

THE SOFTWARE IS PROVIDED "AS IS", WITHOUT WARRANTY OF ANY KIND, EXPRESS OR IMPLIED, INCLUDING BUT NOT LIMITED TO THE WARRANTIES OF MERCHANTABILITY, FITNESS FOR A PARTICULAR PURPOSE AND NONINFRINGEMENT. IN NO EVENT SHALL THE AUTHORS OR COPYRIGHT HOLDERS BE LIABLE FOR ANY CLAIM, DAMAGES OR OTHER LIABILITY, WHETHER IN AN ACTION OF CONTRACT, TORT OR OTHERWISE, ARISING FROM, OUT OF OR IN CONNECTION WITH THE SOFTWARE OR THE USE OR OTHER DEALINGS IN THE SOFTWARE.

For more information see: https://en.wikipedia.org/wiki/MIT_License

This part of code contains helper classes used in DataTree object

itertree.itree_helpers.**accu_iterator**(*iterable*, *accu_method*, *initial_value=(None)*)
A method that enables itertools accumulation over a method .. note:: This method is just needed because in python <3.8 itertools accumulation has no initial parameter! :param iterable: iterable :param accu_method: accumulation method (will be fet by two parameters cumulated and new item) :return: accumulated iterator

itertree.itree_helpers.**is_iterator_empty**(*iterator*)
checks if the given iterator is empty

Parameters **iterator** – iterator to be checked

Return type tuple

Returns

- (True, iterator) - empty
- (False, iterator) - item inside

itertree.itree_helpers.**rindex**(*lst*, *value*)
find last occurance of a itme in the list :param lst: list :param value: search value :return:

class itertree.itree_helpers.**iTLink**(*file_path=None*, *target_path=None*, *link_item=None*)
Bases: object

Definition of a link to an element in another DataTree

get_init_args()

get_target_tree(*self_itree*, *source_dir=None*)

is_file_updated(*source_dir=None*)

property loaded

property is_loaded

property link_item

property file_path

property target_path

property link_tag

property link_data

property source_path

set_source_path(*path*)

set_loaded(*tag=None*, *data=None*)

property tags

set_tags_and_keys(*tags*, *keys*)

property file_crc

get_args()

class itertree.itree_helpers.**iTFLAG**
Bases: object

public flags for setting the *iTree behavior during `__init__()*

READ_ONLY_TREE = 1

READ_ONLY_VALUE = 2

LOAD_LINKS = 4

class itertree.itree_helpers.**Any**
Bases: object

Helper class used for marking that the ìTree()-object is "empty" no value is stored inside.

If required use the class as it is, do not instance an object.

class itertree.itree_helpers.**NoValue**
Bases: object

Helper class used for marking that the ìTree()-object is "empty" no value is stored inside.

If required use the class as it is, do not instance an object.

class itertree.itree_helpers.**NoTag**
Bases: object

Helper class used for the NoTag-family tag which is automatically used in case no explicit tag is given during creation of the ìTree()-object.

If required use the class as it is, do not instance an object.

class itertree.itree_helpers.**NoKey**
Bases: object

Helper class used for the NoKey entries in dicts stored as value object in the ìTree()-object.

If required use the class as it is, do not instance an object.

class itertree.itree_helpers.**NoTarget**
Bases: object

Helper class used for the NoKey entries in dicts stored as value object in the ìTree()-object.

If required use the class as it is, do not instance an object.

class itertree.itree_helpers.**ArgTuple**
Bases: tuple

class itertree.itree_helpers.**Tag**(*tag=<class 'itertree.itree_helpers.NoTag'>*)
Bases: object

Helper class used in get-methods for marking that the given value is a family-tag and not an index or key, etc.

tag

class itertree.itree_helpers.**TagIdx**(*tag*, *idx*)
Bases: tuple

property idx
Alias for field number 1

property tag
Alias for field number 0

itertree.itree_helpers.**getter_to_list**(*get_result*)
Helper function that always creates a list from a `iTree`get-method result.

1. In case we have a iterator the list with the iterator items is created.
2. In case we have a single item a list [single_item] is created

3. In case we have no item or empty iterator an empty list is created.

Parameters **get_result** – result coming from a getter method

Return type list

Returns result list

CHAPTER

FOUR

ITERTREE EXAMPLES

4.1 Usage examples

In the example section you can find two example files itree_usage_example.py and itree_data_examples.py which explain how itertree package might be used.

4.1.1 itree_usage_example1.py

In this example we build a tree that contains the continents and the related countries insite. Additionally we add some country related data. Finally we do some analysis and filter the tree for specific content.

After executing the script the output might look like this:

```
iTree('root')
 > iTree('Africa', value={'surface': 30200000, 'inhabitants': 1257000000})
 .  > iTree('Ghana', value={'surface': 238537, 'inhabitants': 30950000})
 .  > iTree('Nigeria', value={'surface': 1267000, 'inhabitants': 23300000})
 > iTree('Asia', value={'surface': 44600000, 'inhabitants': 4000000000})
 .  > iTree('China', value={'surface': 9596961, 'inhabitants': 1411780000})
 .  > iTree('India', value={'surface': 3287263, 'inhabitants': 1380004000})
 > iTree('America', value={'surface': 42549000, 'inhabitants': 1009000000})
 .  > iTree('Canada', value={'surface': 9984670, 'inhabitants': 38008005})
 .  > iTree('Mexico', value={'surface': 1972550, 'inhabitants': 127600000})
 > iTree('Australia&Oceania', value={'surface': 8600000, 'inhabitants': 36000000})
 .  > iTree('Australia', value={'surface': 7688287, 'inhabitants': 25700000})
 .  > iTree('New Zealand', value={'surface': 269652, 'inhabitants': 4900000})
 > iTree('Europe', value={'surface': 10523000, 'inhabitants': 746000000})
 .  > iTree('France', value={'surface': 632733, 'inhabitants': 67400000})
 .  > iTree('Finland', value={'surface': 338465, 'inhabitants': 5536146})
 > iTree('Antarctica', value={'surface': 14000000, 'inhabitants': 1100})
Filtered items; inhabitants in range: [0,20000000]
iTree('New Zealand', value={'surface': 269652, 'inhabitants': 4900000})
iTree('Finland', value={'surface': 338465, 'inhabitants': 5536146})
iTree('Antarctica', value={'surface': 14000000, 'inhabitants': 1100})
Filter2 items (we expect Antarctica does not match anymore!):
iTree('New Zealand', value={'surface': 269652, 'inhabitants': 4900000})
iTree('Finland', value={'surface': 338465, 'inhabitants': 5536146})
```

4.1.2 itree_usage_example2.py

In this code we build a larger *iTree* by reading a part of the filesystem into a tree. Again we do some analysis related to the read in files and folder.

After executing the script the output might look like this:

```
We read a part of the filesystem ('C:\\Tools\\Python\\Python39') into an itertree
Number of items read in 19878
The load in tree has a depth of 11
How many files are bigger then 1000000 Bytes?
Number of Matches: 2
How many files are in size 9000 ~ 10000 Bytes?
Number of Matches: 6
How many files are touched (modified) during the last day?
Number of Matches: 0
How many files are touched (modified) during the last minute?
Number of Matches: 0
```

4.1.3 itree_data_models.py

This examples focuses on the value stored in the *iTree*-object. We play with data-moddels and show how the user can use the models to determine the values stored in the related *iTree*-object.

We do not use any external packages in the examples but the user may also use the more advanced Pydantic package as a good option to define very powerful data models.

About the data models one can say that the data model can be used with the focus of checking and formatting of the stored data:

- check data type
- check value range (give intervals, limits)
- do we have an array of the data type and what is max length
- for strings we can use matches or regex checks of values
- for formatting think about numerical values (integer dec/hex/bin representation) or float number of digits to round to
- We can also define more abstract datatypes like keylists or enumerated keys.

In the file you can see some examples of how this data models can be defined and used.

After executing the script the output might look like this:

```
Run itertree data_model.py example
Each iTree item will contain different types of data models for the values
Build the tree
Append model items and enter values

Enter a string in the string model, iTree nows that a model is in value and takes␣
↪over the given value implicit into the model
Appended item: iTree('str_len20_item', value=iTStrModel(<class 'itertree.itree_
↪helpers.NoValue'>, None, (20,)))
Content stored in item value: iTStrModel('Hello world', None, (20,))
Content delivered via get_value(): Hello world
Enter a string in the string model which is to long
Exception raised (and handled):
```

(continues on next page)

(continued from previous page)

```
Given value shape=(55,) (position=0) too large for model (shape=(20))

Enter a string in the string model, iTree nows that a model is in value and takes␣
↪over the given value implicit into the model
Appended item: iTree('ascii_str_len40_item', value=iTASCIIStrModel(<class 'itertree.
↪itree_helpers.NoValue'>, None, (40,)))
Content stored in item value: iTASCIIStrModel('Hello world', None, (40,))
Content delivered via get_value(): Hello world
Enter a string in the ASCII string model which is to long
Exception raised (and handled):
Given value shape=(440,) (position=0) too large for model (shape=(40))
Enter a string in the ASCII string model which contains non ascii characters
Exception raised (and handled):
Non ASCII character '°' found in value (position=0) -> not accepted by model

Enter len()=2 floats list
Appended item: iTree('float_array2_item', value=iTFloatModel(<class 'itertree.itree_
↪helpers.NoValue'>, None, (2,), None, '{:.2f}'))
Content stored in item value: iTFloatModel([1.3, 6.4], None, (2,), None, '{:.2f}')
Content delivered via get_value(): [1.3, 6.4]
Enter a numeric string in the float model
Content delivered via get_value(): [1.0, 3.0]
Content delivered via get_value(): [1.3, 3.1]
Enter a single item list in the float array model
Content delivered via get_value(): [1.1]
Enter a string in the float model
Exception raised (and handled):
could not convert string to float: 'ABC'
Enter a single float in the float array model
Exception raised (and handled):
Given value shape=() too small for model (shape=(2)) ->expecting more dimensions
Enter a triple item list in the float array model
Exception raised (and handled):
Given value shape=(3,) (position=0) too large for model (shape=(2))

Enter single float with range
Appended item: iTree('float_single_item', value=iTFloatModel(<class 'itertree.itree_
↪helpers.NoValue'>, None, (), mSetInterval(mSetItem(-10, True), mSetItem(10, True)),
↪'{:.4e}'))
Content stored in item value: iTFloatModel(5.5, None, (), mSetInterval(mSetItem(-10,␣
↪True), mSetItem(10, True)), '{:.4e}')
Content delivered via get_value(): 5.5
Enter a numeric string in the float model
Content delivered via get_value(): -4.4
Enter a string in the float model
Exception raised (and handled):
could not convert string to float: 'ABC'
Enter a float list in the float model
Exception raised (and handled):
Given value shape=(2,) has more dimensions as model accepts (model-shape=()
Enter a float out of range upper limit in the float model
Exception raised (and handled):
Given value does not match to given filter_method (out of range)
Enter a float out of range lower limit in the float model
Exception raised (and handled):
Given value does not match to given filter_method (out of range)
```

(continues on next page)

(continued from previous page)

```
Enter timestamp
Appended item: iTree('time_stamp_item', value=TimeModel(datetime.datetime(1970, 1, 1,␣
↪1, 0)))
Content stored in item value: TimeModel(datetime.datetime(2023, 6, 15, 22, 33, 48,␣
↪451045))
Content delivered via get_value(): 2023-06-15 22:33:48.451045
Content stored in item value: TimeModel(datetime.datetime(2023, 6, 15, 22, 33, 48,␣
↪451045))
Content delivered via get_value(): 2023-06-15 22:33:48.451045
Enter a string in the time model
Exception raised (and handled):
Given value 'ABC' could not be converted in internal datetime object
Enter a negative float in the time model
Exception raised (and handled):
Given value -100 could not be converted in internal datetime object
```

itree_link_example1.py

This example file should show the user how links can be used and how the links are stored. THe user can also see how specific items are converted to local ones. Especially take a look on when the *load_links()* is called and which effect it has if the method is not called or called.

Please compare the output with the code executed:

```
iTree with linked element but no links loaded:
iTree('root')
 > iTree('A')
 > iTree('B')
 > iTree('B')
 .  > iTree('Ba')
 .  > iTree('Bb')
 .  > iTree('Bb')
 .  > iTree('Bc')
 > iTree('internal_link', link=iTLink(None,[('B', 1)]), flags=0b100000)
None
iTree with linked element with links loaded
iTree loaded links:

iTree('root')
 > iTree('A')
 > iTree('B')
 > iTree('B')
 .  > iTree('Ba')
 .  > iTree('Bb')
 .  > iTree('Bb')
 .  > iTree('Bc')
 > iTree('internal_link', link=iTLink(None,[('B', 1)]), flags=0b100100)
 .  >>iTree('Ba')
 .  >>iTree('Bb')
 .  >>iTree('Bb')
 .  >>iTree('Bc')
iTree with updated linked element but no reload of the links:

iTree('root')
 > iTree('A')
```

(continues on next page)

(continued from previous page)

```
 > iTree('B')
 > iTree('B')
 .  > iTree('Ba')
 .  > iTree('Bb')
 .  > iTree('Bb')
 .  > iTree('Bc')
 .  > iTree('B_post_append')
 > iTree('internal_link', link=iTLink(None,[('B', 1)]), flags=0b100100)
 .  >>iTree('Ba')
 .  >>iTree('Bb')
 .  >>iTree('Bb')
 .  >>iTree('Bc')
iTree with linked element with links loaded
iTree with updated linked element and with links reloaded:
iTree('root')
 > iTree('A')
 > iTree('B')
 > iTree('B')
 .  > iTree('Ba')
 .  > iTree('Bb')
 .  > iTree('Bb')
 .  > iTree('Bc')
 .  > iTree('B_post_append')
 > iTree('internal_link', link=iTLink(None,[('B', 1)]), flags=0b100100)
 .  >>iTree('Ba')
 .  >>iTree('Bb')
 .  >>iTree('Bb')
 .  >>iTree('Bc')
 .  >>iTree('B_post_append')
iTree with linked element and additional local items:
iTree('root')
 > iTree('A')
 > iTree('B')
 > iTree('B')
 .  > iTree('Ba')
 .  > iTree('Bb')
 .  > iTree('Bb')
 .  > iTree('Bc')
 .  > iTree('B_post_append')
 > iTree('internal_link', link=iTLink(None,[('B', 1)]), flags=0b100100)
 .  >>iTree('Ba')
 .  >>iTree('Bb')
 .  > iTree('Bb', value='myvalue')
 .  .  > iTree('sublocal')
 .  >>iTree('Bc')
 .  >>iTree('B_post_append')
 .  > iTree('new')
iTree with linked element and the overloading local item deleted:
iTree('root')
 > iTree('A')
 > iTree('B')
 > iTree('B')
 .  > iTree('Ba')
 .  > iTree('Bb')
 .  > iTree('Bb')
 .  > iTree('Bc')
 .  > iTree('B_post_append')
```

(continues on next page)

(continued from previous page)

```
 > iTree('internal_link', link=iTLink(None,[('B', 1)]), flags=0b100100)
 .  >>iTree('Ba')
 .  >>iTree('Bb')
 .  >>iTree('Bb')
 .  >>iTree('Bc')
 .  >>iTree('B_post_append')
 .  > iTree('new')
iTree load from file with load_links parameter disabled (to make internal structure␣
↪visible):
-> See the placeholder element that was added to keep the key of the local item Bb[1]␣
↪(flags==0b10000)
iTree('root')
 > iTree('A')
 > iTree('B')
 > iTree('B')
 .  > iTree('Ba')
 .  > iTree('Bb')
 .  > iTree('Bb')
 .  > iTree('Bc')
 .  > iTree('B_post_append')
 > iTree('internal_link', link=iTLink(None,[('B', 1)]), flags=0b100000)
 .  >>iTree('Ba')
 .  > iTree('Bb', value=0, flags=0b10000)
 .  > iTree('Bb', value='myvalue')
 .  .  > iTree('sublocal')
 .  > iTree('Bc', value=0, flags=0b10000)
 .  >>iTree('B_post_append')
 .  > iTree('new')
iTree load from file with load_links() executed:
iTree('root')
 > iTree('A')
 > iTree('B')
 > iTree('B')
 .  > iTree('Ba')
 .  > iTree('Bb')
 .  > iTree('Bb')
 .  > iTree('Bc')
 .  > iTree('B_post_append')
 > iTree('internal_link', link=iTLink(None,[('B', 1)]), flags=0b100100)
 .  >>iTree('Ba')
 .  >>iTree('Bb')
 .  > iTree('Bb', value='myvalue')
 .  .  > iTree('sublocal')
 .  >>iTree('Bc')
 .  >>iTree('B_post_append')
 .  > iTree('new')
```

performance_analysis.exec_performance

In this example we run performance tests related different functionalities with diffrent types of packages targeting tree functionalities.

For the results and output please have a look in the chapter **`Comparison of the iTree object with lists and dicts`_** .

If the user likes to run the performance test on his own maschine he must ensure that the targeted packeges installed. The user may also change the setup related to the number of items or the depth of the tree which is used for comparison. The parameters can be found in the *__init__()*-method of test classes defined.

CHAPTER

FIVE

COMPARISON

In this chapter we compare the itertree package with other packages which are targeting nested tree structures too or that might be used for such an approach. We like to show that itertree package is on a comparable performance level (or better) even that in most cases we have from the functional point much more features implemented. In the final comparison with the other packages we will mark those functional differences too.

Each package is developed with a specific focus and therefore comparisons are always a bit misleading. Most often there are good reasons for the different behavior (e.g. ElementTrees are build for xml representations and therefore tags are limited to the "rules" of xml). We tried to use the packages as correct as possible but we may missed a function and the performance shown might be worse. We apologize in tis case and ask the autor to contact us via GIT Issue.

We compare *iTree* also with the standard types like *dict*, *list* and *deque*, *OrderedDict* from the collections package. This is done to see how far away we are with itertree from the build-in classes which are somehow the benchmark. In the comparison we must consider that some package are C-compiled (like the build-in types or ElementTree). This leads automatically into a speed advantage against the packages which are python based (like itertree)..

The code for this analysis is placed in itertree/examples/performance folder and the user can execute the analysis on his own environment too. The user can easy adapt the parameters related to size and depth of the created trees. The analysis can only be performed if the targeted packages are available in the local installation. Not found modules are skipped automatically. The user can find some experimental not published packages imported in the code, this should be ignored. In case no blist package is installed you may skip the *insert()* operation of *iTree* for large trees, it s slowed down a lot (standard *list* used). **-> Actions which take 30 seconds or more are skipped and just marked as "-> skipped too slow".**

Finally we tried to make the testing as comparable as possible:

- Because some tree-like-classes are limited to string-type keys/tags we always used string type keys (The string creation costs time so we create the strings for all objects).
- For flat list-like objects (level one only objects we must create a nested structure to make it comparable and to test the in-depth access each item is in this case a tuple of (tag/key, value, subtree).
- For flat dict-like objects we also had to extend a nested structure again a tuple of (value, subtree) is used.
- We do not use the quickest possible functions to get a specific object we try to use the best comparable function (e.g. we do not compare a tree build via append() with a build via comprehension).
- We used helper functions to realize comparable functionalities in case the object does not provide it out of the box. Sometimes this might be really meaning less (test a dict for index access or a list for key access). But this is done to really compare to the feature-set of itertree but we do not stress this comparison to much. But it shows very often that this object cannot be used out of the box as a nested tree representation.

  Sometimes we use helper functions also to overcome RecursionErrors in deep trees. In case of recursive definitions in the object we needed a iterative counter part. But it's not done in all cases (e.g. *deepcopy()* does not work on all other objects).

Note: IMPORTANT: Please consider that for the smaller trees the shown relative differences are less important because the absolute time for the operation is anyway very short. Normally nobody will have an issue with this the times are anyway extremely short and if the operation is not repeated the difference is neglect able even that the factor might be 10. If we compare the performance in between the objects we mainly look on the large scale analysis because some objects getting much slower if the size or the depth grows.

We used for the here discussed analysis a setup with Python 3.9 incl. blist package installed on a Windows OS.

..note :: If you need even better performance we can recommend Python 3.11 or cython. If you have cython installed you can use: *cythonize.exe -i itree_...* on the modules in itertree. At least we can recommend to do this on the modules:

- itree_main.py
- itree_private.py
- itree_getitem.py
- itree_indepth.py

Let's see the difference for the *append()* by appending 500000 items in the tree:

Python 3.9 incl. blist (as used in this comparison:

```
itertree.iTree:
tree=iTree(); tree.append(iTree(tag,value))                    0.663867 s
build-in list:
tree=list(); tree.append((key,value,list()))                   0.091123 s  ->
↪ 7.285x faster as iTree
build-in dict:
tree=dict(); tree[key]=(value,dict())                          0.115628 s  ->
↪ 5.741x faster as iTree
```

Python 3.9 cythonized incl. blist:

```
itertree.iTree:
tree=iTree(); tree.append(iTree(tag,value))                    0.058339 s
build-in list:
tree=list(); tree.append((key,value,list())                    0.009436 s  ->
↪ 6.183x faster as iTree
build-in dict:
tree=dict(); tree[key]=(value,dict())                          0.011373 s  ->
↪ 5.129x faster as iTree
```

We see that we can win "only" 10% of speed.

If we move to Python 3.11 we see also a better performance:

Python 3.11 incl. blist; append 500000 items:

```
itertree.iTree:
tree=iTree(); tree.append(iTree(tag,value))                    0.606875 s
build-in list:
tree=list(); tree.append((key,value,list())                    0.089229 s  ->
↪ 6.801x faster as iTree
build-in dict:
tree=dict(); tree[key]=(value,dict())                          0.146663 s  ->
↪ 4.138x faster as iTree
```

We see that we can win here 8% of speed.

The effect is a bit dependent to the used functionality. But we can see that the implementation is on a very high level and even cythonize the modules does not bring a big performance boost.

The output is reduced by some spaces so it fits better on the html page.

5.1 Analysis Results

5.1.1 Building trees via item append

Performance analysis related to level 1 only trees with a size of 5000; build via *append()* function:

```
itertree.iTree:
tree=iTree(); tree.append(iTree(tag,value))                              0.007014 s
build-in list:
tree=list(); tree.append((key,value,list())                              0.000890 s  -> 7.
↪883x faster as iTree
build-in dict:
tree=dict(); tree[key]=(value,dict())                                    0.000870 s  -> 8.
↪065x faster as iTree
collections.deque:
tree=deque(); tree.append((key,value,deque())                            0.001786 s  -> 3.
↪927x faster as iTree
collections.OrderedDict:
tree=odict(); tree[key]=(value,odict())                                  0.001079 s  -> 6.
↪500x faster as iTree
blist.blist:
tree=blist(); tree.append((key,value,blist())                            0.002133 s  -> 3.
↪288x faster as iTree
indexed.IndexedOrderedDict:
tree=IndexedOrderedDict(); tree[key]=(value,IndexedOrderedDict())        0.004104 s  -> 1.
↪709x faster as iTree
indexed.Dict:
tree=Dict(); tree[key]=(value,Dict())                                    0.003955 s  -> 1.
↪773x faster as iTree
xml.etree.ElementTree.Element:
tree=Element(); tree.append(Element(key,{"value":key}))                  0.004201 s  -> 1.
↪670x faster as iTree
lxml.etree.Element:
tree=Element(); tree.append(Element(key,{"value":key}))                  0.020792 s  -> 0.
↪337x faster as iTree
pyTooling.Tree.Node:
tree=Node(); tree.AddChild(Node(key,value))                              0.007428 s  -> 0.
↪944x faster as iTree
treelib.Node:
tree=Tree(); Tree.create_node(key,key, parent="root",value=value)        0.023033 s  -> 0.
↪305x faster as iTree
anytree.Node:
tree=Node(); Node(key, parent=tree,value=value)                          0.401775 s  -> 0.
↪017x faster as iTree
```

Performance analysis related to level 1 only trees with a size of 500000; build via *append()* function:

```
itertree.iTree:
tree=iTree(); tree.append(iTree(tag,value))                            0.663867 s
```

(continues on next page)

(continued from previous page)

```
build-in list:
tree=list(); tree.append((key,value,list())                              0.091123 s  -> 7.
↪285x faster as iTree
build-in dict:
tree=dict(); tree[key]=(value,dict())                                    0.115628 s  -> 5.
↪741x faster as iTree
collections.deque:
tree=deque(); tree.append((key,value,deque())                            0.166790 s  -> 3.
↪980x faster as iTree
collections.OrderedDict:
tree=odict(); tree[key]=(value,odict())                                  0.148325 s  -> 4.
↪476x faster as iTree
blist.blist:
tree=blist(); tree.append((key,value,blist())                            0.218836 s  -> 3.
↪034x faster as iTree
indexed.IndexedOrderedDict:
tree=IndexedOrderedDict(); tree[key]=(value,IndexedOrderedDict())        0.232782 s  -> 2.
↪852x faster as iTree
indexed.Dict:
tree=Dict(); tree[key]=(value,Dict())                                    0.232265 s  -> 2.
↪858x faster as iTree
xml.etree.ElementTree.Element:
tree=Element(); tree.append(Element(key,{"value":key}))                  0.235984 s  -> 2.
↪813x faster as iTree
lxml.etree.Element:
tree=Element(); tree.append(Element(key,{"value":key}))                  1.775910 s  -> 0.
↪374x faster as iTree
pyTooling.Tree.Node:
tree=Node(); tree.AddChild(Node(key,value))                              0.701030 s  -> 0.
↪947x faster as iTree
treelib.Node:
tree=Tree(); Tree.create_node(key,key, parent="root",value=value)        1.945515 s  -> 0.
↪341x faster as iTree
anytree.Node
tree=Tree(); %s(key, parent=tree,value=value)                              -> skipped too␣
↪slow
```

Performance analysis related related to trees with depth 100 and a size of 1000; build via *append()* function:

```
itertree.iTree:
tree=iTree(); tree.append(iTree(tag,value))                              0.001502 s
build-in list:
tree=list(); tree.append((key,value,list())                              0.000279 s  -> 5.
↪390x faster as iTree
build-in dict:
tree=dict(); tree[key]=(value,dict())                                    0.000368 s  -> 4.
↪081x faster as iTree
collections.deque:
tree=deque(); tree.append((key,value,deque())                            0.000480 s  -> 3.
↪132x faster as iTree
collections.OrderedDict:
tree=odict(); tree[key]=(value,odict())                                  0.000377 s  -> 3.
↪986x faster as iTree
blist.blist:
tree=blist(); tree.append((key,value,blist())                            0.000581 s  -> 2.
↪587x faster as iTree
indexed.IndexedOrderedDict:
```

(continues on next page)

(continued from previous page)

```
tree=IndexedOrderedDict(); tree[key]=(value,IndexedOrderedDict())  0.001621 s  -> 0.
↪927x faster as iTree
indexed.Dict:
tree=Dict(); tree[key]=(value,Dict())                              0.001446 s  -> 1.
↪039x faster as iTree
xml.etree.ElementTree.Element:
tree=Element(); tree.append(Element(key,{"value":key}))            0.001595 s  -> 0.
↪942x faster as iTree
lxml.etree.Element:
tree=Element(); tree.append(Element(key,{"value":key}))            0.004319 s  -> 0.
↪348x faster as iTree
pyTooling.Tree.Node:
tree=Node(); tree.AddChild(Node(key,value))                        0.001427 s  -> 1.
↪053x faster as iTree
treelib.Node:
tree=Tree(); Tree.create_node(key,key, parent="root",value=value)  0.005254 s  -> 0.
↪286x faster as iTree
anytree.Node:
tree=Node(); Node(key, parent=tree,value=value)                    0.009790 s  -> 0.
↪153x faster as iTree
```

Performance analysis related related to trees with depth 1000 and a size of 10000; build via *append()* function:

```
itertree.iTree:
tree=iTree(); tree.append(iTree(tag,value))                        0.013546 s
build-in list:
tree=list(); tree.append((key,value,list())                        0.003512 s  -> 3.
↪857x faster as iTree
build-in dict:
tree=dict(); tree[key]=(value,dict())                              0.004493 s  -> 3.
↪015x faster as iTree
collections.deque:
tree=deque(); tree.append((key,value,deque())                      0.005670 s  -> 2.
↪389x faster as iTree
collections.OrderedDict:
tree=odict(); tree[key]=(value,odict())                            0.005158 s  -> 2.
↪626x faster as iTree
blist.blist:
tree=blist(); tree.append((key,value,blist())                      0.007198 s  -> 1.
↪882x faster as iTree
indexed.IndexedOrderedDict:
tree=IndexedOrderedDict(); tree[key]=(value,IndexedOrderedDict())  0.013275 s  -> 1.
↪020x faster as iTree
indexed.Dict:
tree=Dict(); tree[key]=(value,Dict())                              0.013865 s  -> 0.
↪977x faster as iTree
xml.etree.ElementTree.Element:
tree=Element(); tree.append(Element(key,{"value":key}))            0.006693 s  -> 2.
↪024x faster as iTree
lxml.etree.Element:
tree=Element(); tree.append(Element(key,{"value":key}))            0.045103 s  -> 0.
↪300x faster as iTree
pyTooling.Tree.Node:
tree=Node(); tree.AddChild(Node(key,value))                        0.013264 s  -> 1.
↪021x faster as iTree
treelib.Node:
tree=Tree(); Tree.create_node(key,key, parent="root",value=value)  0.092746 s  -> 0.
↪146x faster as iTree
```

(continues on next page)

(continued from previous page)

```
anytree.Node:
tree=Node(); Node(key, parent=tree,value=value)                    0.587480 s  -> 0.
↪023x faster as iTree
```

The *iTree*-object and the most other objects show here comparable performance.

- *list*, *dict* : Both build-in object are the benchmark in this analysis. *list* is the clear winner of this comparison. The *dict*- object shows like all dict-like objects a relative drop in performance if the tree size grows. If we compare *iTree* with those objects we see that we are for level 1 trees round about 7-5 times slower and the really deep trees 3-4 times slower. This is not surprising considering the c-code base and the deep integration into the Python-Interpreter.
- Other dicts and lists: We see that those objects are slower as the build-in counterparts We can say in mean *iTree* is round about two times slower. As standard dict the dict-like objects getting relative-slower for larger sized trees.
- The two ElementTrees shows an ambivalent picture but all in all we would say they on large trees they are on same level like *iTree*. As we will see from design the ElementTree from xml is optimized for access where lxml seems to be optimized for build (instance). We see that lxml ElementTree is here a head the of xml counter-part and *iTree* too.
- Indexed dicts and the PyTooling are on really comparable level as *iTree* in all *append()* cases executed.
- The tree related objects treelib and anytree are clearly slower. As we will see for all other functions too anytree is a lot slower especially if the tree size crows. At one point the objects seems do block even after many minutes of execution we do not get a result.

5.1.2 Build tree via extend or comprehension

The iTree object supports the build of an object via a comprehension like functionality which is the fastest way to build the object. The operation is for nested structures not so much quicker compared with *append()* (only 10-20% times quicker). We present here just the max-size results.

Performance analysis related to level 1 only trees with a size of 500000; build via comprehension or *extend()* function:

```
itertree.iTree:
tree=iTree(key,subtree=(iTree(key,value) for ....))                0.610306 s
build-in list:
tree=list((key,value,list()) for ....))                            0.125169 s  ->␣
↪4.876x faster as iTree
build-in dict:
tree=dict((key,(value,dict())) for ....))                          0.215009 s  ->␣
↪2.839x faster as iTree
collections.deque:
tree=deque((key,value,deque()) for ....))                          0.207484 s  ->␣
↪2.941x faster as iTree
collections.OrderedDict:
tree=odict((key,(value,odict())) for ....))                        0.299324 s  ->␣
↪2.039x faster as iTree
blist.blist:
tree=blist((key,value,blist()) for ....))                          0.303959 s  ->␣
↪2.008x faster as iTree
indexed.IndexedOrderedDict:
tree=IndexedOrderedDict((key,(value,IndexedOrderedDict())) for ....))  0.782604 s  ->␣
↪0.780x faster as iTree
indexed.Dict:
```

(continues on next page)

(continued from previous page)

```
tree=Dict((key,(value,Dict())) for ....))                                    0.778467 s  ->␣
↪0.784x faster as iTree
xml.etree.ElementTree.Element:
tree.extend(Element(key,{"value":key}))                                      0.301490 s  ->␣
↪2.024x faster as iTree
lxml.etree.Element:
tree.extend(Element(key,{"value":key}))                                      1.804367 s  ->␣
↪0.338x faster as iTree
pyTooling.Tree.Node:
tree=Node(children=[Node(key,value) for ...])                                0.734321 s  ->␣
↪0.831x faster as iTree
anytree.Node:
tree=%s(children=[%s(key,value) for ...])                                    -> skipped too␣
↪slow
```

Performance analysis related related to trees with depth 1000 and a size of 10000; build via comprehension or *extend()* function:

```
itertree.iTree:
tree=iTree(key,subtree=(iTree(key,value) for ....))                          0.598814 s
build-in list:
tree=list((key,value,list()) for ....))                                      0.112530 s  ->␣
↪5.321x faster as iTree
build-in dict:
tree=dict((key,(value,dict())) for ....))                                    0.197339 s  ->␣
↪3.034x faster as iTree
collections.deque:
tree=deque((key,value,deque()) for ....))                                    0.198221 s  ->␣
↪3.021x faster as iTree
collections.OrderedDict:
tree=odict((key,(value,odict())) for ....))                                  0.275480 s  ->␣
↪2.174x faster as iTree
blist.blist:
tree=blist((key,value,blist()) for ....))                                    0.271218 s  ->␣
↪2.208x faster as iTree
indexed.IndexedOrderedDict:
tree=IndexedOrderedDict((key,(value,IndexedOrderedDict())) for ....))        0.712246 s  ->␣
↪0.841x faster as iTree
indexed.Dict:
tree=Dict((key,(value,Dict())) for ....))                                    0.710830 s  ->␣
↪0.842x faster as iTree
xml.etree.ElementTree.Element:
tree.extend(Element(key,{"value":key}))                                      0.299102 s  ->␣
↪2.002x faster as iTree
lxml.etree.Element:
tree.extend(Element(key,{"value":key}))                                      1.978916 s  ->␣
↪0.303x faster as iTree
pyTooling.Tree.Node:
tree=Node(children=[Node(key,value) for ...])                                0.691485 s  ->␣
↪0.866x faster as iTree
anytree.Node:
tree=%s(children=[%s(key,value) for ...])                                    -> skipped too␣
↪slow
```

We see that in this case the differences in between the objects are less compared to *append()*. The build-in *list* is again the fastest object it is 5 times quicker than *iTree*.

The results we have seen in *append()* are somehow reproduced. The indexed dicts and the pyToolingTree are here a bit behind *ìTree*.

5.1.3 Index based item access

Beside the build of the nested structure the access of items in the diffrent levels is the second important core-function we see for trees. We can here differentiate in between the index and the key/tag based access.

In *iTree* the user has the choice in between the "lazy" get item access with flexible targets or a specific access. The flexible (common) access is slower because the given target must be identified. Because this feature does not exist in the other objects we mainly compare with the specific access (even that common access comparison is given in brackets too).

We know that list-like object are designed for index-access only and dict-like objects (except indexed dict) are designed for key-based-access. We had to use helper functions for the missing function and we will see that they are comparable slow.

Let's first have a look on index based access. Dict-like objects access via *next(itertools.islice(tree,idx))* which is much slower for the last items in the stored order but we show here the mean access time.

Performance analysis related to level 1 only trees with a size of 5000; access via *__getitem__(index)* function:

```
itertree.iTree (common target access):
tree[idx]                                                0.001344 s
itertree.iTree (index-specific access):
tree.get.by_idx(idx)                                     0.000857 s ->  1.568x faster␣
↪as common access
build-in list:
tree[idx]                                                0.000274 s  -> 3.124x (4.
↪898x) faster as iTree
build-in dict:
next(islice(tree.values(),idx))                          0.046756 s  -> 0.018x (0.
↪029x) faster as iTree
collections.deque:
tree[idx]                                                0.000458 s  -> 1.872x (2.
↪935x) faster as iTree
collections.OrderedDict:
next(islice(tree.values(),idx))                          0.274780 s  -> 0.003x (0.
↪005x) faster as iTree
blist.blist:
tree[idx]                                                0.000271 s  -> 3.169x (4.
↪969x) faster as iTree
indexed.IndexedOrderedDict:
tree.values()[idx]                                       0.001780 s  -> 0.482x (0.
↪755x) faster as iTree
indexed.Dict:
tree.values()[idx]                                       0.001685 s  -> 0.509x (0.
↪798x) faster as iTree
xml.etree.ElementTree.Element:
tree[idx]                                                0.000245 s  -> 3.494x (5.
↪479x) faster as iTree
lxml.etree.Element:
tree[idx]                                                0.044648 s  -> 0.019x (0.
↪030x) faster as iTree
pyTooling.Tree.Node:
next(islice(tree.GetChildren(),idx))                     0.263845 s  -> 0.003x (0.
↪005x) faster as iTree
```

(continues on next page)

(continued from previous page)

```
treelib.Node:
tree.children[idx]                                    2.032886 s  -> 0.000x (0.
↪001x) faster as iTree
anytree.Node:
tree.children[idx]                                    0.038357 s  -> 0.022x (0.
↪035x) faster as iTree
```

Performance analysis related to level 1 only trees with a size of 500000; access via *__getitem__(index)* function:

```
itertree.iTree (common target access):
tree[idx]                                             0.142918 s
itertree.iTree (index-specific access):
tree.get.by_idx(idx)                                  0.097269 s ->  1.469x faster␣
↪as common access
build-in list:
tree[idx]                                             0.028292 s  -> 3.438x (5.
↪052x) faster as iTree
build-in dict:
next(islice(tree.values(),idx))                         -> skipped too slow
collections.deque:
tree[idx]                                             6.575242 s  -> 0.015x (0.
↪022x) faster as iTree
collections.OrderedDict:
next(islice(tree.values(),idx))                         -> skipped too slow
blist.blist:
tree[idx]                                             0.029997 s  -> 3.243x (4.
↪764x) faster as iTree
indexed.IndexedOrderedDict:
tree.values()[idx]                                    0.187038 s  -> 0.520x (0.
↪764x) faster as iTree
indexed.Dict:
tree.values()[idx]                                    0.190313 s  -> 0.511x (0.
↪751x) faster as iTree
xml.etree.ElementTree.Element:
tree[idx]                                             0.029029 s  -> 3.351x (4.
↪923x) faster as iTree
lxml.etree.Element:
tree[idx]                                               -> skipped too slow
pyTooling.Tree.Node:
next(islice(tree.GetChildren(),idx))                    -> skipped too slow
treelib.Node:
tree.children[idx]                                      -> skipped too slow
anytree.Node no test source was build (append())        -> operation skipped
```

The *iTree*-class supports the in-depth access of items out of the box (via *itree.deep.*). For most other objects an in-depth helper access function was created. For treelib we couldn't create a comparable function so that the object is not considered in the followoing analysis.

Performance analysis related related to trees with depth 100 and a size of 1000; access via *__getitem__(index)* function:

```
itertree.iTree (common target access):
tree.get(*idxs)                                        0.010597 s
itertree.iTree (index-specific access):
tree.get.by_idx(*idxs)                                 0.002180 s ->  4.862x␣
↪faster as common access
build-in list:
tree[idx]                                              0.001252 s  -> 1.741x (8.
↪463x) faster as iTree
```

(continues on next page)

(continued from previous page)

```
build-in dict:
next(islice(tree.values(),idx))                                  0.006946 s  -> 0.314x (1.
↪526x) faster as iTree
collections.deque:
tree[idx]                                                        0.001490 s  -> 1.463x (7.
↪114x) faster as iTree
collections.OrderedDict:
next(islice(tree.values(),idx))                                  0.009487 s  -> 0.230x (1.
↪117x) faster as iTree
blist.blist:
tree[idx]                                                        0.001353 s  -> 1.611x (7.
↪832x) faster as iTree
indexed.IndexedOrderedDict:
tree.values()[idx]                                               0.016197 s  -> 0.135x (0.
↪654x) faster as iTree
indexed.Dict:
tree.values()[idx]                                               0.016333 s  -> 0.133x (0.
↪649x) faster as iTree
xml.etree.ElementTree.Element:
tree[idx]                                                        0.001144 s  -> 1.905x (9.
↪262x) faster as iTree
lxml.etree.Element:
tree[idx]                                                        0.006120 s  -> 0.356x (1.
↪732x) faster as iTree
pyTooling.Tree.Node:
next(islice(tree.GetChildren(),idx))                             0.014973 s  -> 0.146x (0.
↪708x) faster as iTree
anytree.Node:
tree.children[idx]                                               0.007702 s  -> 0.283x (1.
↪376x) faster as iTree
```

Performance analysis related related to trees with depth 1000 and a size of 10000; access via *__getitem__(index)* function:

```
itertree.iTree (common target access):
tree.get(*idxs)                                                  1.049203 s
itertree.iTree (index-specific access):
tree.get.by_idx(*idxs)                                           0.197017 s ->  5.325x␣
↪faster as common access
build-in list:
tree[idx]                                                        0.117011 s  -> 1.684x (8.
↪967x) faster as iTree
build-in dict:
next(islice(tree.values(),idx))                                  0.679821 s  -> 0.290x (1.
↪543x) faster as iTree
collections.deque:
tree[idx]                                                        0.149676 s  -> 1.316x (7.
↪010x) faster as iTree
collections.OrderedDict:
next(islice(tree.values(),idx))                                  0.938039 s  -> 0.210x (1.
↪119x) faster as iTree
blist.blist:
tree[idx]                                                        0.130424 s  -> 1.511x (8.
↪045x) faster as iTree
indexed.IndexedOrderedDict:
tree.values()[idx]                                               1.543223 s  -> 0.128x (0.
↪680x) faster as iTree
```

(continues on next page)

(continued from previous page)

```
indexed.Dict:
tree.values()[idx]                                    1.548948 s  -> 0.127x (0.
↪677x) faster as iTree
xml.etree.ElementTree.Element:
tree[idx]                                             0.098422 s  -> 2.002x (10.
↪660x) faster as iTree
lxml.etree.Element:
tree[idx]                                             6.198828 s  -> 0.032x (0.
↪169x) faster as iTree
pyTooling.Tree.Node:
next(islice(tree.GetChildren(),idx))                  1.437700 s  -> 0.137x (0.
↪730x) faster as iTree
anytree.Node:
tree.children[idx]                                    0.747130 s  -> 0.264x (1.
↪404x) faster as iTree
```

First we like to remark that for small trees the common access function in *iTree* is only 1-1.5 times slower as the specific one. Only for larger trees the difference get obvoius up two 5 times slower in our examples. What we can also see that *iTree* supports it's nested structure quite well and it has even more advantages for in-depth access.

- dict-like objects: We do not want to stress this point here they are obviously not made for this kind of access and therefore slower.
- *list* - is again the fastest object. Of course it is designed for index access. But the difference to *iTree* is not much in deeper trees *list* is less then two times quicker (only).
- Indexed dicts - do not perform as good as the name and functions let us expect. The index access is better then for normal dicts for sure but it is clearly behind *iTree*.
- ElementTrees - Getting slower for lager number of children. For the deep structures *iTree* outperforms those objects. For this function lxml ElementTree is clearly slower as the xml ElementTree.
- All other tree objects - People may say index access is less important in trees this might be the reason why index access is for all of them slower as in *iTree*.

5.1.4 Key based item access

As mentioned in the sentence before for some users this access type might be for trees more important then the index access. This means at the end trees are more seen as nested dicts.

The list-like are not designed for this kind of access and for those objects we end up in a search functionality which is based on an interation and comparison (we used *tree[tree.index((key,value,subtree))]*). The operation is not 100% accurate normally we should just search for the key with something like *next(dropwhile(lambda item: item[0] != key,tree))* but this would be even slower but it is used where *index()*-method was not avaiable.

Performance analysis related to level 1 only trees with a size of 5000; access via *__getitem__(key)* function:

```
itertree.iTree (common target access):
tree[key]                                             0.002031 s
itertree.iTree (tag_idx-specific access):
tree.get.by_tag_idx(key)                              0.001589 s ->  1.278x␣
↪faster as common access
build-in list:
tree[tree.index(key)]                                 0.172946 s  -> 0.009x (0.
↪012x) faster as iTree
build-in dict:
tree[key]                                             0.000844 s  -> 1.882x (2.
↪406x) faster as iTree
```

(continues on next page)

(continued from previous page)

```
collections.deque:
tree[tree.index(key)]                                   0.180119 s  -> 0.009x (0.
↪011x) faster as iTree
collections.OrderedDict:
tree[key]                                               0.000852 s  -> 1.865x (2.
↪384x) faster as iTree
blist.blist:,
tree[tree.index(key)]                                   0.205660 s  -> 0.008x (0.
↪010x) faster as iTree
indexed.IndexedOrderedDict:
tree[key]                                               0.000984 s  -> 1.616x (2.
↪065x) faster as iTree
indexed.Dict:
tree[key]                                               0.000958 s  -> 1.659x (2.
↪120x) faster as iTree
xml.etree.ElementTree.Element:
tree.find(key)                                          0.122876 s  -> 0.013x (0.
↪017x) faster as iTree
lxml.etree.Element:
tree.find(key)                                          0.114247 s  -> 0.014x (0.
↪018x) faster as iTree
pyTooling.Tree.Node:
tree.GetNodeByID(key))                                  0.001260 s  -> 1.261x (1.
↪612x) faster as iTree
treelib.Node:
tree.get_node(key)                                      0.001685 s  -> 0.943x (1.
↪206x) faster as iTree
anytree.Node:
search.find(tree, lambda node: node.name == key)                          14.093013␣
↪s  -> 0.000x (0.000x) faster as iTree
next(dropwhile(lambda item: item.name != key, tree.children)              1.835935 s␣
↪ -> 0.001x (0.001x) faster as iTree
```

Performance analysis related to level 1 only trees with a size of 500000; access via *__getitem__(key)* function:

```
itertree.iTree (common target access):
tree[key]                                               0.266813 s
itertree.iTree (tag_idx-specific access):
tree.get.by_tag_idx(key)                                0.215222 s ->  1.240x␣
↪faster as common access
build-in list:
tree[tree.index(key)]                                     -> skipped too slow
build-in dict:
tree[key]                                               0.103994 s  -> 2.070x (2.
↪566x) faster as iTree
collections.deque:
tree[tree.index(key)]                                     -> skipped too slow
collections.OrderedDict:
tree[key]                                               0.103348 s  -> 2.082x (2.
↪582x) faster as iTree
blist.blist:
tree[tree.index(key)]                                     -> skipped too slow
indexed.IndexedOrderedDict:
tree[key]                                               0.119337 s  -> 1.803x (2.
↪236x) faster as iTree
indexed.Dict:
tree[key]                                               0.117257 s  -> 1.835x (2.
↪275x) faster as iTree
```

(continues on next page)

(continued from previous page)

```
xml.etree.ElementTree.Element:
tree.find(key)                                          -> skipped too slow
lxml.etree.Element:
tree.find(key)                                          -> skipped too slow
pyTooling.Tree.Node:
tree.GetNodeByID(key))                                0.158424 s  -> 1.359x (1.
↪684x) faster as iTree
treelib.Node:
tree.get_node(key)                                    0.196832 s  -> 1.093x (1.
↪356x) faster as iTree
anytree.Node no test source was build (append())                          ->␣
↪operation skipped
```

Performance analysis related related to trees with depth 100 and a size of 1000; access via *__getitem__(key)* function:

```
itertree.iTree (common target access):
tree[key]                                             0.012834 s
itertree.iTree (tag_idx-specific access):
tree.get.by_tag_idx(key)                              0.003589 s ->  3.575x␣
↪faster as common access
build-in list:
tree[tree.index(key)]                                 0.014637 s  -> 0.245x (0.
↪877x) faster as iTree
build-in dict:
tree[key]                                             0.001911 s  -> 1.878x (6.
↪715x) faster as iTree
collections.deque:
tree[tree.index(key)]                                 0.014943 s  -> 0.240x (0.
↪859x) faster as iTree
collections.OrderedDict:
tree[key]                                             0.001984 s  -> 1.809x (6.
↪468x) faster as iTree
blist.blist:
tree[tree.index(key)]                                 0.014691 s  -> 0.244x (0.
↪874x) faster as iTree
indexed.IndexedOrderedDict:
tree[key]                                             0.002619 s  -> 1.371x (4.
↪900x) faster as iTree
indexed.Dict:
tree[key]                                             0.002598 s  -> 1.382x (4.
↪940x) faster as iTree
xml.etree.ElementTree.Element:
tree.find(key)                                        0.004508 s  -> 0.796x (2.
↪847x) faster as iTree
lxml.etree.Element:
tree.find(key)                                        0.152578 s  -> 0.024x (0.
↪084x) faster as iTree
pyTooling.Tree.Node:
tree.GetNodeByID(key))                                0.004712 s  -> 0.762x (2.
↪724x) faster as iTree
treelib.Node:
tree.get_node(key)                                    0.001022 s  -> 3.513x (12.
↪559x) faster as iTree
anytree.Node:
search.find(tree, lambda node: node.name == key)                        17.558134␣
↪s  -> 0.000x (0.001x) faster as iTree
next(dropwhile(lambda item: item.name != key, tree.children))           0.021549 s␣
↪ -> 0.167x (0.596x) faster as iTree
```

(continues on next page)

(continued from previous page)

Performance analysis related related to trees with depth 1000 and a size of 10000; access via *__getitem__(key)* function:

```
itertree.iTree (common target access):
tree[key]                                                    1.230146 s
itertree.iTree (tag_idx-specific access):
tree.get.by_tag_idx(key)                                     0.327140 s ->  3.760x␣
↪faster as common access
build-in list:
tree[tree.index(key)]                                        1.392063 s  -> 0.235x (0.
↪884x) faster as iTree
build-in dict:
tree[key]                                                    0.169229 s  -> 1.933x (7.
↪269x) faster as iTree
collections.deque:
tree[tree.index(key)]                                        1.410674 s  -> 0.232x (0.
↪872x) faster as iTree
collections.OrderedDict:
tree[key]                                                    0.165853 s  -> 1.972x (7.
↪417x) faster as iTree
blist.blist:
tree[tree.index(key)]                                        1.353723 s  -> 0.242x (0.
↪909x) faster as iTree
indexed.IndexedOrderedDict:
tree[key]                                                    0.223637 s  -> 1.463x (5.
↪501x) faster as iTree
indexed.Dict:
tree[key]                                                    0.222354 s  -> 1.471x (5.
↪532x) faster as iTree
xml.etree.ElementTree.Element:
tree.find(key)                                               0.419544 s  -> 0.780x (2.
↪932x) faster as iTree
lxml.etree.Element:
tree.find(key)                                              33.371158 s  -> 0.010x (0.
↪037x) faster as iTree
pyTooling.Tree.Node:
tree.GetNodeByID(key))                                       0.497697 s  -> 0.657x (2.
↪472x) faster as iTree
treelib.Node:
tree.get_node(key)                                           0.054741 s  -> 5.976x (22.
↪472x) faster as iTree
anytree.Node:
search.find(tree, lambda node: node.name == key)                                 ->␣
↪skipped too slow
next(dropwhile(lambda item: item.name != key, tree.children))                2.157126 s␣
↪ -> 0.152x (0.570x) faster as iTree
```

Even that *iTree* is in base more related to a list we can see that the key access is on a very high level.

- *dict* and all dict-like objects (inkl. indexed) - This build-in object is for sure the benchmark for al key related access objects. Suprsingly *iTree* is not far away it is less as two times slower.
- treelib - the flatten storage structure of treelib allows very quick key access over the different levels of the tree. This structure is for in-depth access the clear winner.
- other tree objects except treelib - all other tree objects are slower as *iTree* especially anytree is again incredible

slow.

- ElementTree - those objects are list-like and the search for tags is clearly slower then in *iTree*. For in-depth access the difference get less and the performance is comparable. The bottleneck is here clearly a level with a lot of items.

5.1.5 copy the tree

The copy function is the most difficult function related to the *iTree* architecture. The challenge is that in *iTree*-objects the *one parent only principle is mandatory*. And therefore we cannot just copy the toplevel item we must copy all the items inside the tree too. The *itree.copy()* operation copies in fact all containing items and it copies in the item the values too. But the values are copied just first level. Which makes the main difference to the deepcopy() operation were we do a deepcopy() of the whole value objects too.

To make the comparison comparable we ensured in the first analysis (against *itree.copy()*) a comparable operation in the objects. We copied the main object and additional we copied all children via:

new_tree=tree.copy() new_tree.clear() new_tree.extend(((i[0],copy(i[1]),copy(i[2]) for i in tree))

We think this kind of copy of all items is the expected behavior in a nested tree.

Second we run the command *copy.copy()* here we do not consider if in this case children are copied or not. For most of the other objects this in fact a top level copy only, we can see this in the huge speed difference. In *iTree* we use for comparison the command *itree.copy_keep_value()* which does not copy the values and is a bit faster as *copy.copy()* ~ *itree.copy()*.

Performance analysis related to level 1 only trees with a size of 5000; for *copy()* functions:

```
itertree.iTree:
tree.copy()                                                                    0.006894 s
tree.copy_keep_value()                                                         0.006740 s
copy.deepcopy(tree)                                                            0.008287 s
build-in list:
n=tree.copy();n.clear();n.extend(((i[0],copy(i[1]),copy(i[2])) for ....))  0.001564 s␣
↪ -> 4.407x faster as iTree
copy.copy(tree)                                                                0.000011 s␣
↪ -> 618.330x faster as iTree
copy.deepcopy(tree)                                                            0.010031 s␣
↪ -> 0.826x faster as iTree
build-in dict:
n=tree.copy();n.update(k:(copy(i[0]),copy(i[1])) for k,i in tree.items())) 0.009887 s␣
↪ -> 0.697x faster as iTree
copy.copy(tree)                                                                0.000024 s␣
↪ -> 282.000x faster as iTree
copy.deepcopy(tree)                                                            0.010486 s␣
↪ -> 0.790x faster as iTree
collections.deque:
n=tree.copy();n.clear();n.extend(((i[0],copy(i[1]),copy(i[2])) for ....))  0.002446 s␣
↪ -> 2.818x faster as iTree
copy.copy(tree)                                                                0.000025 s␣
↪ -> 268.518x faster as iTree
copy.deepcopy(tree)                                                            0.014278 s␣
↪ -> 0.580x faster as iTree
collections.OrderedDict:
n=tree.copy();n.update(k:(copy(i[0]),copy(i[1])) for k,i in tree.items())) 0.011029 s␣
↪ -> 0.625x faster as iTree
copy.copy(tree)                                                                0.000483 s␣
↪ -> 13.960x faster as iTree
```

(continues on next page)

(continued from previous page)

```
copy.deepcopy(tree)                                                    0.011297 s␣
↪ -> 0.734x faster as iTree
blist.blist:
n=tree.copy();n.clear();n.extend(((i[0],copy(i[1]),copy(i[2])) for ....))  0.005918 s␣
↪ -> 1.165x faster as iTree
copy.copy(tree)                                                        0.000003 s␣
↪ -> 2106.188x faster as iTree
copy.deepcopy(tree)                                                    0.018452 s␣
↪ -> 0.449x faster as iTree
indexed.IndexedOrderedDict:
n=tree.copy();n.update(k:(copy(i[0]),copy(i[1])) for k,i in tree.items())) 0.013091 s␣
↪ -> 0.527x faster as iTree
copy.copy(tree)                                                        0.001601 s␣
↪ -> 4.209x faster as iTree
copy.deepcopy(tree)                                                    0.012605 s␣
↪ -> 0.657x faster as iTree
indexed.Dict:
n=tree.copy();n.update((k,(copy(i[0]),copy(i[1])) for k,i in tree.items()))0.013222 s␣
↪ -> 0.521x faster as iTree
copy.copy(tree)                                                        0.001580 s␣
↪ -> 4.265x faster as iTree
copy.deepcopy(tree)                                                    0.012263 s␣
↪ -> 0.676x faster as iTree
xml.etree.ElementTree.Element:
n=tree.copy();n.clear();n.extend((copy(i) for i in tree))              0.001259 s␣
↪ -> 5.476x faster as iTree
copy.copy(tree)                                                        0.000013 s␣
↪ -> 518.446x faster as iTree
copy.deepcopy(tree)                                                    0.000878 s␣
↪ -> 9.438x faster as iTree
lxml.etree.Element:
n=tree.copy();n.clear();n.extend((copy(i) for i in tree))              0.006082 s␣
↪ -> 1.133x faster as iTree
copy.copy(tree)                                                        0.001318 s␣
↪ -> 5.114x faster as iTree
copy.deepcopy(tree)                                                    0.000973 s␣
↪ -> 8.515x faster as iTree
pyTooling.Tree.Node:
copy.copy(tree)                                                        0.000004 s␣
↪ -> 1604.714x faster as iTree
copy.deepcopy(tree)                                                    0.055884 s␣
↪ -> 0.148x faster as iTree
anytree.Node:
copy.copy(tree)                                                        0.000020 s␣
↪ -> 333.653x faster as iTree
copy.deepcopy(tree)                                                    0.021417 s␣
↪ -> 0.387x faster as iTree
```

Performance analysis related to level 1 only trees with a size of 500000; for *copy()* functions:

```
itertree.iTree:
tree.copy()                                                            0.835458 s
tree.copy_keep_value()                                                 0.796100 s
copy.deepcopy(tree)                                                    0.952877 s
build-in list:
n=tree.copy();n.clear();n.extend(((i[0],copy(i[1]),copy(i[2])) for ....))  0.211637 s␣
↪ -> 3.948x faster as iTree
```

(continues on next page)

(continued from previous page)

```
copy.copy(tree)                                                          0.012144 s␣
↪ -> 65.553x faster as iTree
copy.deepcopy(tree)                                                      1.215360 s␣
↪ -> 0.784x faster as iTree
build-in dict:
n=tree.copy();n.update(k:(copy(i[0]),copy(i[1])) for k,i in tree.items()) 1.120784 s␣
↪ -> 0.745x faster as iTree
copy.copy(tree)                                                          0.020014 s␣
↪ -> 39.776x faster as iTree
copy.deepcopy(tree)                                                      1.290939 s␣
↪ -> 0.738x faster as iTree
collections.deque:
n=tree.copy();n.clear();n.extend(((i[0],copy(i[1]),copy(i[2])) for ....)) 0.306400 s␣
↪ -> 2.727x faster as iTree
copy.copy(tree)                                                          0.012119 s␣
↪ -> 65.691x faster as iTree
copy.deepcopy(tree)                                                      1.625661 s␣
↪ -> 0.586x faster as iTree
collections.OrderedDict:
n=tree.copy();n.update(k:(copy(i[0]),copy(i[1])) for k,i in tree.items()) 1.347654 s␣
↪ -> 0.620x faster as iTree
copy.copy(tree)                                                          0.172031 s␣
↪ -> 4.628x faster as iTree
copy.deepcopy(tree)                                                      1.505746 s␣
↪ -> 0.633x faster as iTree
blist.blist:
n=tree.copy();n.clear();n.extend(((i[0],copy(i[1]),copy(i[2])) for ....)) 0.699678 s␣
↪ -> 1.194x faster as iTree
copy.copy(tree)                                                          0.000093 s␣
↪ -> 8532.692x faster as iTree
copy.deepcopy(tree)                                                      2.336833 s␣
↪ -> 0.408x faster as iTree
indexed.IndexedOrderedDict:
n=tree.copy();n.update(k:(copy(i[0]),copy(i[1])) for k,i in tree.items()) 1.617339 s␣
↪ -> 0.517x faster as iTree
copy.copy(tree)                                                          0.294597 s␣
↪ -> 2.702x faster as iTree
copy.deepcopy(tree)                                                      1.535705 s␣
↪ -> 0.620x faster as iTree
indexed.Dict:
n=tree.copy();n.update((k,(copy(i[0]),copy(i[1])) for k,i in tree.items()))1.549998 s␣
↪ -> 0.539x faster as iTree
copy.copy(tree)                                                          0.265445 s␣
↪ -> 2.999x faster as iTree
copy.deepcopy(tree)                                                      1.579321 s␣
↪ -> 0.603x faster as iTree
xml.etree.ElementTree.Element:
n=tree.copy();n.clear();n.extend((copy(i) for i in tree))                0.169371 s␣
↪ -> 4.933x faster as iTree
copy.copy(tree)                                                          0.008154 s␣
↪ -> 97.634x faster as iTree
copy.deepcopy(tree)                                                      0.132343 s␣
↪ -> 7.200x faster as iTree
lxml.etree.Element:
n=tree.copy();n.clear();n.extend((copy(i) for i in tree))                2.620617 s␣
↪ -> 0.319x faster as iTree
copy.copy(tree)                                                          1.134328 s␣
↪ -> 0.702x faster as iTree
```

(continues on next page)

(continued from previous page)

```
copy.deepcopy(tree)                                                    1.031247 s␣
↪ -> 0.924x faster as iTree
pyTooling.Tree.Node:
copy.copy(tree)                                                        0.000004 s␣
↪ -> 204128.257x faster as iTree
copy.deepcopy(tree)                                                    6.419806 s␣
↪ -> 0.148x faster as iTree
anytree.Node no test source was build (append())                         ->␣
↪operation skipped
```

For in-depth copies over multiple levels we use just *deepcopy()*. But fo tre depth above 500 all othetr objects except *iTree* raise RecursionError.

Performance analysis related to trees with depth 100 and a size of 1000; for *deepcopy()* function:

```
itertree.iTree:
tree.copy()                                                  0.001347 s
copy.deepcopy(tree)                                          0.002609 s
build-in list:
copy.deepcopy(tree)                                          0.003200 s  -> 0.815x (0.
↪421x) faster as iTree
build-in dict:
copy.deepcopy(tree)                                          0.003270 s  -> 0.798x (0.
↪412x) faster as iTree
collections.deque:
copy.deepcopy(tree)                                          0.003990 s  -> 0.654x (0.
↪338x) faster as iTree
collections.OrderedDict:
copy.deepcopy(tree)                                          0.003983 s  -> 0.655x (0.
↪338x) faster as iTree
blist.blist:
copy.deepcopy(tree)                                          0.005017 s  -> 0.520x (0.
↪269x) faster as iTree
indexed.IndexedOrderedDict:
copy.deepcopy(tree)                                          0.006796 s  -> 0.384x (0.
↪198x) faster as iTree
indexed.Dict:
copy.deepcopy(tree)                                          0.006855 s  -> 0.381x (0.
↪197x) faster as iTree
xml.etree.ElementTree.Element:
copy.deepcopy(tree)                                          0.000184 s  -> 14.173x (7.
↪318x) faster as iTree
lxml.etree.Element:
copy.deepcopy(tree)                                          0.008311 s  -> 0.314x (0.
↪162x) faster as iTree
pyTooling.Tree.Node:
copy.deepcopy(tree)                                          0.012250 s  -> 0.213x (0.
↪110x) faster as iTree
treelib.Node:
copy.deepcopy(tree)                                          0.011016 s  -> 0.237x (0.
↪122x) faster as iTree
anytree.Node:
copy.deepcopy(tree)                                                    0.005676 s␣
↪ -> 0.460x (0.237x) faster as iTree
anytree.Node no test source was build (append())                         ->␣
↪operation skipped
```

Performance analysis related to trees with depth 100 and a size of 1000; for *deepcopy()* function:

```
itertree.iTree:
tree.copy()                                                    0.014717 s
copy.deepcopy(tree)                                            0.027662 s
build-in list:
copy.deepcopy(tree)                                              skipped -
↪> RecursionError
build-in dict:
copy.deepcopy(tree)                                              skipped -
↪> RecursionError
...
```

We see that copying is a bit tricky for trees and when ever we really copy the tree in depth the performance of *iTree* is quite good. But of course for top level copies *iTree* has disadvantages.

But even for *deepcopy()* operation outperforms *iTree* all the other objects (except xml.ElementTree which is quicker) Because of the iterative copy implementation in *iTree* this even works for very deep trees where all other objects fails (if the user does not increase the recursion limit).

5.1.6 Delete items

Performance analysis related to level 1 only trees with a size of 50000; delete items:

```
itertree.iTree (del by idx):
del tree[0] for  ...                        0.039939 s
itertree.iTree (del by idx):
del tree[-1] for ...                        0.035988 s
itertree.iTree (self by key):
del tree[tag_idx] for ...                   0.345494 s ->  0.116x faster as idx access
build-in list:
del tree[0]                                 2.156652 s  -> 0.160x faster as iTree
del tree[-1]                                0.003988 s  -> 9.024x faster as iTree
build-in dict:
del tree[key]                               0.008149 s  -> 42.397x faster as iTree
collections.deque:
del tree[0]                                 0.008281 s  -> 41.722x faster as iTree
del tree[-1]                                0.009760 s  -> 3.687x faster as iTree
collections.OrderedDict:
del tree[key]                               0.010482 s  -> 32.959x faster as iTree
blist.blist:
del tree[0]                                 0.019203 s  -> 17.992x faster as iTree
del tree[-1]                                0.016717 s  -> 2.153x faster as iTree
indexed.IndexedOrderedDict:
del tree[key]                               2.171378 s  -> 0.159x faster as iTree
indexed.Dict:
del tree[key]                               2.175590 s  -> 0.159x faster as iTree
xml.etree.ElementTree.Element:
del tree[0]                                 0.742840 s  -> 0.465x faster as iTree
del tree[-1]                                0.006250 s  -> 5.758x faster as iTree
lxml.etree.Element:
del tree[0]                                 0.011022 s  -> 31.346x faster as iTree
del tree[-1]                                0.011105 s  -> 3.241x faster as iTree
treelib.Node:
tree.remove_node(key)                       2.356795 s  -> 0.147x faster as iTree
```

The comparison related to item delete operation is really difficult. And we see very different behavior for the executed cases. The results are very wide variance (e.g. dict is 40 times quicker as ìTree`and indexed dicts are 6 times slower).

We must also say that for a size of 50000 items for some classes the time gets already critical (more then 2 seconds) and surprisingly *list* is also in this category for first item delete. We do not show the results for 500000 items here, because many classes would have bin skipped because of the time limit. The situation is here that the operation for those classes gets a lot more difficult if the size grows.

We ran the following cases:

- list-like delete first element (index 0) -> compared with same operation in *iTree*
- list-like delete last element (index -1) -> compared with same operation in *iTree*
- dict-like delete per key -> compared with same operation in *iTree*

For PyToolingTree and anytree we did not found a delete function for items.

For the *iTree*-class the *__delitem__()* method is very difficult. We must delete the item in the main list and in the family. We must consider different cases and in case of local items which overload linked items we must replace in stead of delete. But even though the speed of the operation (in the level 1 example is good. But we must say that the class take big advantages of the good delete performance of the blist class (if package is not installed this operation will be much worse).

- *list* - "our all time winner" performance for this operation not very well. We see that especially the delete of the first items is very costy (all items must be reindexed). For the last items the list is quicker then *iTree*.
- *dict* and *OrderedDict* - are clearly much quicker then *iTree* (more then 40-30 times) and round about 5 times quicker then delete per index in *iTree*.
- *deque* - performance very well and much quicker then *iTree*. Suprisingly the delete from the end is slower then the delete from the beginning.
- Indexed Dicts - The indexed dicts are much slower then *iTree*
- xml-ElementTree - behaves like list
- lxml ElementTree - is clearly quicker then *iTree*
- treelib - is much slower then *iTree* but we must say we didn't find here a way for indexed based deletes, we used a deleted targeted key

5.1.7 Tree` operations

Finally we just ran an analysis of the *iTree* object itself so that we have an overview of the main functionalities.

We target a lot of functions which are only available in *iTree* and where we found no counterpart in the other objects.

Performance analysis related to level 1 only trees with a size of 500000:

```
tree=iTree("root",subtree=[...])                                    0.574042 s
tree=iTree(); tree.append()...                                      0.683904 s␣
↪->  0.839x faster as extend()
tree=iTree(); tree.insert()...                                      0.831075 s␣
↪->  0.823x faster as append()
tree.load_links()            # 500000 linked-items loaded           0.919389 s
tree.get.by_idx(idx)         # specific absolute index access       0.097410 s
tree[idx]                    # common absolute index access         0.145026 s␣
↪->  0.672x faster as specific
tree.get.by_idx_slice(slice) # specific absolute index slice access 0.012604 s
tree[slice]                  # common absolute index slice access   0.012821 s␣
↪->  0.983x faster as specific
tree.get.by_tag_idx(tag_idx) # specific tag-idx access             0.206083 s␣
↪->  0.473x faster as get_by_idx()
```

(continues on next page)

(continued from previous page)

```
tree[tag_idx]                # common tag-idx access                    0.256707 s␣
↪->  0.803x faster as specific
tree.getitem_tag_idx_slice((tag,fam_idx_slice)) # specific tag_idx slice   0.001470 s
tree[(tag,fam_idx_slice]     # common tag_idx slice                     0.001866 s␣
↪->  0.788x faster as specific
tree.get.by_tag(tag)         # specific family-tag access               0.263708 s
tree[tag]                    # common family-tag access                 0.350820 s␣
↪->  0.752x faster as specific
tree.dumps()                 # serialize into string (json)             0.996655 s
pickle.dumps(tree)           # serialize via pickle                     0.647319 s
```

Performance analysis related to trees with depth 100 and a size of 1000:

```
tree=iTree(); tree.append()...                                          0.013060 s
tree.load_links()            # 10 linked-items loaded                   0.033486 s
tree.get.by_idx(idx)         # specific absolute index access           0.193028 s
tree[idx]                    # common absolute index access             1.039325 s␣
↪->  0.186x faster as specific
tree.get.by_tag_idx(tag_idx) # specific tag-idx access                  0.344499 s␣
↪->  0.560x faster as get_by_idx()
tree[tag_idx]                # common tag-idx access                    1.247241 s␣
↪->  0.276x faster as specific
tree.get.by_tag(tag)         # specific family-tag access               0.307431 s
tree[tag]                    # common family-tag access                 2.242912 s␣
↪->  0.137x faster as specific
tree.dumps()                 # serialize into string (json)             0.039315 s
```

The *insert()* operation based on the internal usage of the blist-package is impressive only 20% slower compared to *append()*.

This analysis shows on first level the common access can be up to two times slower as the specific item access. For in-depth access the difference grows.

Serialization via pickle is quicker compared to the json-serialization used by *iTree().dumps()* but for deep trees RecursionErrors will appear.

5.2 Final summary

From the functional point *iTree* ,has the following functions that are not found in most of the other objects:

- linking of branches and overwrite local items
- store the structure in a file by serializing all value objects too (we do not consider here something like pickle)
- in-depth access and iterators

If the objects have such solutions too it will be mentioned.

5.2.1 iTree vs. list like objects

Related to performance we can see that the internal structure of *iTree* is also list like. But we have some overhead to handle (tag family related management) so we are for most operations a bit slower than the list like objects. And we must consider here that most of this objects are implemented on c-level which gives an additional boost.

For the un-typic operations like key-access (were lists must do at least a search by iterating over all elements) we see that *iTree* behaves much quicker. We think that such operations are mandatory for trees. As we see in the other tree like objects the targeting related to keys is much more important than index access. For us this is the main reason why list-like objects are not fitting to the requirements of tree structures.

We must also remark here that in the comparison we had to find a way to use the list-like objects as nested objects. We stored in each item a tuple of (key,value,subtree). A pure flat list of values and not containing such tuples would be much quicker. But this is not the use-case of a tree were you need the possibility of subtrees.

Focusing on functional limitations we must first see that list are not made for nested, in-depth structures. In our comparison we had to use a helper by putting tuples in the values in which as last item again a list for the deeper sub-structure was placed. So with out such a help object and with addtional methods for in-depth functions lists can not be used for trees out of the box.

And as already said the in our opinion mandatory key/item access is very slow in lists.

In lists any object-type can be used as key if stored in the helper structure (tuple).

5.2.2 iTree vs. dict like objects

Talking about performance the speed of the standard dict is not so far ahead from *iTree* as we can see it in lists. Especially for structures with a large number of items dicts getting relatively slower.

The other non-standard dicts are only in some cases a bit quicker as *iTree*.

The non typic access via index is slow except for the indexed dicts. But even those are slower in index access as *iTree*. And we must they that index acces in dicts is quicker as key-access in lists (for larger number of items).

Dict objects contains normally only level 1 children and as in lists an additional helper object is required to store sub-dicts. And we can see that for in-depth access most of the dicts are slower then *iTree*.

The indexed dicts of the indexed module are an interesting alternative to *iTree*. We can imagine that those those objects would be a good base for tree structures. But in practice we can see that those objects behave slower then *iTree* in most cases and therefore there we see no reason to use those objects for trees.

When we talk about functional limits of dicts compared to *iTree* we see as explained that they are not out of the box nested.

Second they are not capable to store an item with same key multiple times as you can do it in *iTree* but also in most of the other tree structures (like xml ElementTree).

The order of the items is not always kept (depends also on the python version) but even if the order is kept the change of the order is not possible or difficult.

In dicts any hashable type can be used as key (as it is for tags in *iTree*.

5.2.3 iTree vs. ElementTree

The ElementTrees gave a very ambivalent picture in general we sse that the object from the xml package is designed for quicker instancing and longer access times compared to the one from the package lxml.

If we look just on the performance we can say that index related functions are very quick better or on same level as *iTree* depending which variant you are looking on. In mean we must say we are on same level. The key related access (tag search) is slower as we have it in *iTree*.

It's not shown here but the storage into files (save/load to/from xml) is quicker then the related functions we have in *iTree* (json files). But as we will see we have normally just strings stored in the object (tag,value).

In general we must say that in those objects we have a real tree functionality realized we have also a larger range of functionalities available then we have it in *iTree*. Especially we have in-depth operations like iterators or access. We have also the very powerful xpath search function. And as in *iTree`the user can store the tag mutliple times in `ElementTree*.

But those trees are made for xml storage and this means they normally handle just strings. If other objects stored in the values a special serializing must be adapted (which will decrease the performance). Especially in the tags the limits are even higher, no special characters can be used there (e.g. spaces are not allowed in xml-tags). The possibility of *iTrees* related to the usage any hashable object as a tag can not be realized in those objects (out of the box).

In the value (in case of ElementTree attrib) we have a dict like structure and the user must use it he cannot exchange the dict-like behavior of the value object.

Finally we can say those alternatives are only good as long as the user just tags/stores string like objects.

5.2.4 iTree vs. PyToolingTree

Related to performance we can say that the two objects are on same level. (On PytToolingTree docu they mention 2 times quicker performance but this was related to older version of *iTree*).

In our opinion the focus related access in PyToolingTree is more in the direction of key-access as index access (in last topic the object is slower).

The overall functionality of this object (Version 4.0.1) is very limited compared to *iTree*. We did not checked all details here but we see the following differences. The item used IDs and those IDs must be unique this means you cannot store same key multiple times (like in *iTree*). We do not see any special in-depth functions all this access must be programmed outside of the object.

The storage into files (serializing) does not exists.

Summary for specific implementations we see this as an alternative. But we see a much bigger functionality in *iTree* with same or even better performance.

5.2.5 iTree vs. treelib

Treelib was integrated relative late in the comparison and some analysis are missing. The structural setup is completely different (nested items are stored in a flat list) and some functions cannot be realized (in our opinion e.g. nested index access).

On performance side we can see that the object is slower for nearly any access type and most of the other functions. Because of the structure we see that the whole tree iteration is very quick but we do not see that the order is really kept here.

In general we found that the object is very difficult to be used. And because of the architecture we see functional limitations (e.g. in-depth index access). also we do not see real in-depth functionalities.

From our point of few there is no reason to take this alternative. The object has functional limits, it's slower and from our experience difficult to use. The documentation is even incomplete from our point of view.

5.2.6 iTree vs. anytree

The recommended object for trees for many users is anytree. And before we started with itertree implementation we thought this object might match to our requirements. But as you can see in the performance analysis the behavior is really disappointing.

The object behaves in all directions very slow. And even in flat trees with more then 5000 elements the objects gets unusable slow. The bad performance was shortly discussed with the author: https://github.com/c0fec0de/anytree/issues/169.

Some case could not work at all the objects seems to block (even for very simple operations e.g. index access on flat trees with 50000 items (I had to wait some minutes to create such trees)).

Additionally we see limitations in anytree:

- You can only use string based tags (not hashable objects like in itertree).
- functional properties of a specific item do not exists (iTree.idx, *iTree*.idx_path, ...)
- But the main issue from our point of view is the really bad performance in case of huge trees (Especially search for item.name is very slow)
- filtering is very slow and not as powerful as in itertree

In general the functionality in anytree is much less and not comparable with *iTree*.

Finally we must say this is the only package which we found not usable at all. It is very slow and blocks in some operations. We cannot recommend to use this package.

5.2.7 Other arguments for *iTree*

One main functionality in *iTree* that is not found in any of the other objects is the possibility to link from one tree to the other tree. This "inheritance" of subtrees seams to be a unique feature.

Also the possibility of marking elements as read-only for specific functions (value read-only, subtree read-only) is unique.

Another thing we do not find one to one in the other objects ís the possibility to store out of the box the trees in files. Especially if we consider that in *iTree* the value objects are serialized too even if they are complex types like (lists, dicts, data-models or even numpy arrays).

The original requirement to develop itertree was the target to store configurations in a more efficient way compared with ini-files, xml-files, json-files, yaml-files. We wanted to extend a tree like data structure with the possibility of linking sub-trees in a main-tree by linking from different sources. Additionally we like to overload in the linked tree some items if required. We can say itertree contains those functionalities and we do not know any other object supporting this.

But beside the orignal starting point we extended the object to a generic python object for trees. It contains a very pythonic standard interface (lists/dict). And can be used for many other proposes too.

As you can see from the naming iterations are supported in a wide range. Especially filtering is important. Here we can find another unique feature for nested trees we did not found in the other objects this is the possibility of hierarchical filtering. The filter will not consider the subtree if the parent does not match. In general for most objects such a filter can be programmed from the outside too. But this has disadvantages if this is not done inside the iterator and it makes additional effort that is not needed in ìtertree.

If the user knows to use the iterators (see e.g. itertools) very efficient code can be created especially if you want to dive inside the tree. The iterators can be cascaded and instanced extremely quick. The iteration runs only finally in the moment you consume the iterator. This is much quicker then instances lists in many steps in between which you iterate multiple times. Those iterators are widely used in itertree and can be used from the outside too. In many of the other objects the delivered objects are lists or tuples and not iterators which is a big disadvantage from our point of view.

CHAPTER

SIX

BACKGROUND INFORMATION ABOUT ITERTREE

The itertree package is originally developed to be used in an internal test-system configuration and measurement environment. In this environment we must handle a huge number of parameters and attributes which are configured via a Graphical User Interface (GUI). The connection of the data and the GUI (editor) is realized via the coupled_object function we have in *iTree*. The so created configuration can be interpreted by test-systems and can be stored in version control systems.

But the idea of tree based configuration is nothing exceptionally new and of course trees can be used for many other proposes. The itertree package for Python is a new approach to get a very performant solution for these proposes even when the trees are very huge (many attributes in deep hierarchies).

In our case the package is also used in embedded environments and for this a pure Python implementation helps to prevent us from different type of cross compilations for our targets. The package should run on any Python >=3.4 interpreter.

6.1 Architecture

To find the best solution we made a lot of testing (check of the already available packages) and we checked other implementation alternatives (like sorted or ordered dicts) but we came to the conclusion that it makes sense to develop an own, new package to match all our requirements.

Based on the pre tests we created an architecture based on a list (blist) and a parallel managed dict that contains the tag families again as lists (blist).

The *iTree* objects is build on these three base elements:

- _items (list/blist) -> main list of items
- _families (dict) -> dict containing the family list (key is tag)
- _value -> place to store the data content of the item

Beside this structure the parent *iTree*-object is stored in the *iTree*-object by this we create the hierarchy. An *iTree*-object can only have one parent! When you feed an *iTree* object during instantiation as subtree parameter then the *iTree* objects children will be copied and taken over in the new *iTree*. The extend function has the same behavior.

A free to use couple_object can be used to combine an *iTree* object with any other python object (e.g. an object in a related tree GUI element) Or for other temporary data. It is not permanent in meaning that it will not be stored in a file (if tree is saved) and it will not be considered in any comparisons (except *equal()* where tit can be included).

The profiling of the package done by running over 100000 base operations gives the following result based on blist:

```
Running on itertree version: 1.0.1
Profiling is done based on 100000 single operations (some clas might be even used␣
↪more often)
```

(continues on next page)

(continued from previous page)

```
         5000038 function calls (4900039 primitive calls) in 3.334 seconds

   Ordered by: standard name

   ncalls  tottime  percall  cumtime  percall filename:lineno(function)
        1    0.000    0.000    3.334    3.334 <string>:1(<module>)
        1    0.000    0.000    0.000    0.000 copy.py:107(_copy_immutable)
        1    0.000    0.000    0.000    0.000 copy.py:66(copy)
   500005    0.033    0.000    0.033    0.000 itree_getitem.py:48(__init__)
   100002    0.078    0.000    0.130    0.000 itree_main.py:1041(append)
500005/400006    1.440    0.000    1.999    0.000 itree_main.py:108(__init__)
   100000    0.128    0.000    0.159    0.000 itree_main.py:1187(insert)
        1    0.000    0.000    0.588    0.588 itree_main.py:1279(extend)
   100000    0.074    0.000    0.118    0.000 itree_main.py:1840(__delitem__)
   200000    0.067    0.000    0.094    0.000 itree_main.py:2018(__getitem__)
        1    0.000    0.000    0.803    0.803 itree_main.py:2160(__mul__)
   199999    0.050    0.000    1.051    0.000 itree_main.py:2261(__copy__)
        1    0.000    0.000    0.000    0.000 itree_main.py:2290(copy)
        2    0.000    0.000    0.000    0.000 itree_main.py:316(parent)
   200000    0.032    0.000    0.032    0.000 itree_main.py:3194(is_link_root)
        2    0.000    0.000    0.000    0.000 itree_main.py:340(root)
   300003    0.532    0.000    1.646    0.000 itree_private.py:223(_iter_extend)
   200000    0.068    0.000    0.113    0.000 itree_private.py:452(_get_copy_args)
   200000    0.193    0.000    1.001    0.000 itree_private.py:555(_iter_copy)
        1    0.356    0.356    3.334    3.334 itree_profile.py:54(performance_dt)
        1    0.038    0.038    0.271    0.271 itree_profile.py:67(<listcomp>)
   300000    0.017    0.000    0.017    0.000 {built-in method builtins.callable}
        1    0.000    0.000    3.334    3.334 {built-in method builtins.exec}
   400002    0.030    0.000    0.030    0.000 {built-in method builtins.hasattr}
   200003    0.015    0.000    0.015    0.000 {built-in method builtins.len}
   100000    0.006    0.000    0.006    0.000 {method '__getitem__' of 'blist.blist'␣
↪objects}
   100000    0.008    0.000    0.008    0.000 {method '__getitem__' of 'dict' objects}
   299800    0.023    0.000    0.023    0.000 {method 'append' of 'blist.blist'␣
↪objects}
   100200    0.007    0.000    0.007    0.000 {method 'append' of 'list' objects}
        2    0.000    0.000    0.000    0.000 {method 'copy' of 'blist.blist' objects}
   100000    0.005    0.000    0.005    0.000 {method 'copy' of 'list' objects}
        1    0.000    0.000    0.000    0.000 {method 'disable' of '_lsprof.Profiler'␣
↪objects}
        1    0.010    0.010    0.588    0.588 {method 'extend' of 'blist.blist'␣
↪objects}
   600002    0.087    0.000    0.087    0.000 {method 'get' of 'dict' objects}
   100000    0.015    0.000    0.015    0.000 {method 'insert' of 'blist.blist'␣
↪objects}
   100000    0.021    0.000    0.021    0.000 {method 'pop' of 'blist.blist' objects}
```

Form our point of view we see a well balanced behavior. Copy is relative costly because it is always an in-depth copy. Deletion is slower then append but still relative quick.

Running the same profiling actions without blist package (using normal list) we get:

```
Running on itertree version: 1.0.1
Profiling is done based on 100000 single operations (some clas might be even used␣
↪more often)
         5000038 function calls (4900039 primitive calls) in 12.823 seconds
```

(continues on next page)

(continued from previous page)

```
   Ordered by: standard name

   ncalls  tottime  percall  cumtime  percall filename:lineno(function)
        1    0.000    0.000   12.823   12.823 <string>:1(<module>)
        1    0.000    0.000    0.000    0.000 copy.py:107(_copy_immutable)
        1    0.000    0.000    0.000    0.000 copy.py:66(copy)
   500005    0.034    0.000    0.034    0.000 itree_getitem.py:48(__init__)
   100002    0.315    0.000    0.361    0.000 itree_main.py:1041(append)
500005/400006    0.615    0.000    1.463    0.000 itree_main.py:108(__init__)
   100000    0.171    0.000    1.515    0.000 itree_main.py:1187(insert)
        1    0.000    0.000    0.533    0.533 itree_main.py:1279(extend)
   100000    0.081    0.000    8.687    0.000 itree_main.py:1840(__delitem__)
   200000    0.066    0.000    0.091    0.000 itree_main.py:2018(__getitem__)
        1    0.000    0.000    0.685    0.685 itree_main.py:2160(__mul__)
   199999    0.049    0.000    0.386    0.000 itree_main.py:2261(__copy__)
        1    0.000    0.000    0.000    0.000 itree_main.py:2290(copy)
        2    0.000    0.000    0.000    0.000 itree_main.py:316(parent)
   200000    0.030    0.000    0.030    0.000 itree_main.py:3194(is_link_root)
        2    0.000    0.000    0.000    0.000 itree_main.py:340(root)
   300003    0.959    0.000    1.399    0.000 itree_private.py:223(_iter_extend)
   200000    0.067    0.000    0.108    0.000 itree_private.py:452(_get_copy_args)
   200000    0.103    0.000    0.337    0.000 itree_private.py:555(_iter_copy)
        1    0.194    0.194   12.823   12.823 itree_profile.py:54(performance_dt)
        1    0.038    0.038    0.238    0.238 itree_profile.py:67(<listcomp>)
   300000    0.016    0.000    0.016    0.000 {built-in method builtins.callable}
        1    0.000    0.000   12.823   12.823 {built-in method builtins.exec}
   400002    0.030    0.000    0.030    0.000 {built-in method builtins.hasattr}
   200003    0.014    0.000    0.014    0.000 {built-in method builtins.len}
   100000    0.033    0.000    0.033    0.000 {method '__getitem__' of 'dict' objects}
   100000    0.005    0.000    0.005    0.000 {method '__getitem__' of 'list' objects}
   400000    0.024    0.000    0.024    0.000 {method 'append' of 'list' objects}
   100002    0.005    0.000    0.005    0.000 {method 'copy' of 'list' objects}
        1    0.000    0.000    0.000    0.000 {method 'disable' of '_lsprof.Profiler'␣
↪objects}
        1    0.010    0.010    0.533    0.533 {method 'extend' of 'list' objects}
   600002    0.096    0.000    0.096    0.000 {method 'get' of 'dict' objects}
   100000    1.310    0.000    1.310    0.000 {method 'insert' of 'list' objects}
   100000    8.559    0.000    8.559    0.000 {method 'pop' of 'list' objects}
```

We see that blist package is really recommended because the general performnce is much better (3 times quicker). But we have to look in the details:

- *__init__()* - instancing is quicker with normal *list*
- *copy()* - is much quicker with *list*
- *insert()* - is much slower with *list*
- *__delitem__()*- is extremely slow with *list*

And for item acces the classes are on same level but even that it is not checked in this analysis we must mention that slicing is much quicker in *blist*. We found that slicing *[:]`is even quicker then `copy()* in *blist*-objects.

We can summarize: It's highly recommended to istnall *blist`package if itertree is used. With `list* only the object still runs smooth (in some cases quicker) as long as the user avoids mass-opertion related to deletion or insertion of items.

6.2 Iteration-Generators and filters

An investigation in other packages showed that search algorithms for specific items are sometimes very slow. Even xml.ElementTree which shows overall a very good performance but it is not very fast when using the *find_all()* method. The xpath syntax is relative powerful but sometimes difficult to use (e.g. try to target the text property). But we found that using iterators and build-in *filter()* function might be quicker and easier to use.

In itertree we have the possibility to define filter functionalities for nearly all the in-depth iteration-generators. We support here the *filter_method* parameter for hierarchical filterings. This means in case the parent does not match we will not iter over the children too. Via external filtering (build-in *filter()*-method) the user can still filter inside the parents if required. But hierarchical filtering is much easier to realize if supported inside the generator itself.

The filter method is fed by the *iTree*-item and must deliver a True/False after the analysis of the item is done.

The itertree package contains predefined filters in the *itree_filter.py* file and they can be reached via *Filter.**** in the code.

Because we are using generators the filtering is very effective. The filters can be combined and so the user can create queries like in a database to catch all information out of the tree and selected the matching items.

The resulting filtered-iterator-object is instanced very quick and it is totally independent from the tree size. After all filtering is combined the iterator can be consumed and in maximum we will iterate only one time over the whole tree. We do not waised any time in typecasts to lists inbetween. This is very memory effective and we avoid unnecessary iterations.

To avoid RecursionErrors all internal and external iterations an done in an iterative and not recursive way. We made tests and most often recursive algorithms will raise the RecursionError exceptions at tree depth >200 levels. The user can extend this by changing Pythons recursion-limit. But it is not required for *iTree*. We also tested the iterative implementation against the recursive ones and we did not find large differences.

6.3 File storage and serializing

The standard format for serializing and storage is a JSON format. It contains a header with environmental information like file (interface) version and the checksum. The data content is represented by a flatten list of items. We store the depth information for each item which allows us to reconstruct the whole tree if the file is read in. If we would store here a nested list we risk that the json-parser may raise RecursionError for deep trees.

The standard serializer can handle a large number of objects and serialize them into the JSON format (numpy.arrays supported too). If the user has additional objects that should be serialized he may extend the serialize or use his own serializer. The serializer is independent from *iTree* itself and another serializer can be defined easily. Please use the *itree_serializer* parameter in the related methods (the used serializer is stored after first usage and will be reused for future operations).

If the user likes to have other output formats (e.g. xml or MessagePack) he must also create his own serializer.

We allow already the packing and hashing of the data before we store it onto a file. Packing helps to keep the files small but the cost of calculation time must be considered and sometimes it's better to use the unpacked files and combine many of those files into an archive afterwards (independent from itertree). Therefore all these options (packing, hashing)vare optional and can switch off if required.

6.4 Data Structure and Data Models

The structure the user store data in the *iTree.value* is totally free any object can be used. In case of list or dict like objects (all objects with a *__getitem__()* method) the user can also use a key or index based access to the items in the structure.

If the user likes to determine which data can be stored in the *ìTree.value`he can store a data model first. If the provided data-model from itertree is used the `set_value()* method will set the value inside the model automatically.

This works also for the key related setter *set_key_value()* in this case the user can store multiple data models related to the given key or index in the *iTree.value* and again the method will exchange the value inside the model.

You might have a look in the examples/itree_data_models.py file to get a better idea what a data-model is in this contents.

6.5 Short words about the licencing

This Software and it's artifacts are licenced under MIT licence with an extension that protects human lives.

Therefore the condition:

" Human protect patch: The program and its derivative work will neither be modified or executed to harm any human being nor through inaction permit any human being to be harmed."

was added to the licence.

The author is aware about the situation that in practices this might not be controlled or even judged.

But it should be clear that from the point of view of the author such an usage is illegal. The author is not willing to spend his lifetime and creativity for the propose of killing people. We think the user should respect the intense of the author if he uses his knowledge and objectives.

Of course the point can be discussed and we respect here other meanings but please consider and respect this as a personal opinion.

In practices people will always find good arguments for utilizing things also to harm people (e.g. control terrorism). But from our point of view in a modern, enlightened society we should find better answers.

E.g. we also think that the possibility of winning a war is a total illusion. Beside raising fears the lie of winning wars is most often used to utilize large amount of resources which make few people richer or more powerful. But all the targeted people on both sides of such conflicts loose lives, freedom, truth, etc.. To protect us from such situations we need globally respected rules and less national intentions. This is the direction were we must put our resources and effort.

At least: People should take responsibility for the objectives they are delivering and publishing. They should give conditions for usage. And the global law should respect such conditions given by the authors even if it is against the national interests and the interests of the majority.

So please respect the authors meaning even if you have other opinion about this content.

CHAPTER

SEVEN

ITERTREE - INTRODUCTION

Do you have to store data in a tree like structure?
Do you need good performance and a reach feature set in the tree object?
You like to serialize and store the structure in files?
And is it helpful for you if you can link subtrees from other trees and add local items in this "inherited" parts?

Please give itertree package a try!

The main class for construction of the trees is the *iTree*-class. Here is a simple representation of a itertree:

```
iTree('root', value='xyz')
> iTree('subitem', value='abc')
> iTree(('tuple', 'tag'), value={'dict': 'value'})
.   > iTree('subtag', value=1)
.   > iTree('subtag', value=2)
> iTree('tag', value=[1, 2, 3])
```

Every node in the itertree (*iTree*-object) contains two main parts:

- First the related sub-structure (*iTree*-children)
- Second the item related value attribute were any kind of object can be stored in

The itertree solution can be compared with nested lists or dicts. Other packages that targeting in the in the same direction are *anytree*, *(l)xml.ElementTree*, *PyToolingTree*. In detail the feature-set and functional focus of *iTree* is a bit different. An overview of the advantage and disadvantages related to the other packages is given in the chapter *Comparison*.

7.1 Status and compatibility information

Version | 1.0.5| has been released!

Be sure to read the *changelog* before upgrading!

Please use the github issues to ask questions report problems.

The original implementation is done in Python 3.9 and it is tested under Python 3.5, 3.6 and 3.9. The package should work for all Python >= 3.4 environments.

The actual development status is "*released*" and stable.

The Software and all related documents are published under MIT license extended by a human protect patch (see *Background Licence*).

7.2 Feature Overview

The main features of the itertree package can be summarized with:

- trees can be structured in different levels (nested trees: parent - children - sub-children - ....)
- the identification tag (key) can be any kind of hashable object
- tags must not be unique (same tags are enumerated and collect in a tag-family)
- item access is possible via tag-index-pair, absolute index, slices, index-lists or filters
- the *iTree*-object keeps the order of the added children
- an *iTree*-object can contain linked/referenced items (linking to other internal tree parts or to an external itertree file is supported)
- in a linked iTree specific items can be *localized* and they can *cover* linked elements (overloading)
- supports standard serialization via export/import to JSON (incl. numpy and OrderedDict data serialization)
- designed for performance (huge trees with hundreds of levels and over a million of items)
- helper functions and data models which can be used to specify the valid values are delivered too
- it's a pure python package (should be easy usable in all environments)
- in general the *iTree*-class can be seen as a functional mix of lists and dicts with deeper levels and references

Here is very simple example of itertree usage:

```
>>> from itertree import * # required for all examples shown in the documentation
>>> # Create root item:
>>> root = iTree('root', value={'mykey': 0})
>>> # Append children:
>>> root.append(iTree('sub', value={'mykey': 1}))
iTree('sub', value={'mykey': 1})
>>> root.append(iTree('sub', value={'mykey': 2}))
iTree('sub', value={'mykey': 2})
>>> root.append(iTree('sub', value={'mykey': 3}))
iTree('sub', value={'mykey': 3})
>>> # Show tree content:
>>> root.render()
iTree('root', value={'mykey': 0})
 > iTree('sub', value={'mykey': 1})
 > iTree('sub', value={'mykey': 2})
 > iTree('sub', value={'mykey': 3})
>>> # Address item via tag-index-pair (key):
>>> root['sub', 1]
iTree('sub', value={'mykey': 2})
>>> # Address item via absolute-index and check stored value:
>>> root[1].value
{'mykey': 2}
```

7.3 Documentation Content

- *Introduction* - Short introduction to the itertree package (this page)
- *Tutorial* - A detailed Tutorial including functional sorted reference description
- *API Reference* - API Description of all containing classes and methods of itertree
- *Usage Examples* - itertree usage examples
- *Comparison* - Compare itertree with other packages
- *Background information* - Some background information about itertree and the target of the development

7.4 Getting started, first steps

7.4.1 Installation and dependencies

The package is a pure python package and does not have any dependencies. But we have two recommendations which give the package additional performance:

- blist - *Highly recommended!* This will speedup the iTree performance in huge trees especially for inserting and lefthand side operations
  - package link: https://pypi.org/project/blist/
  - documentation: http://stutzbachenterprises.com/blist/.

  -> in case the package is not found normal list object will be used instead -> depending on the size blist is especially better for *insert()* operations and slicing

  For Python 3.10 and 3.11 we created a package based on: https://github.com/stefanor/blist/tree/python3.11 and some additional adaptions. The package can be found under: https://github.com/BR1py/itertree/tree/main/dist We did not test the package in detail but the itertree testsuite runs without issues.

  ..note :: We recommend to use it only for the newer Python versions. For older versions Python <=3.9 use the original package from PyPI.
- orjson - A quicker json parser that used to create the JSON structures during serializing/deserializing

  -> in case orjson is not found, standard json package will be used

To install the itertree package just run the command:

```
pip install itertree
```

Inside the installed package the user can find a folder "examples" which might be a good starting point to learn the functionalities.

7.4.2 First steps

All important classes of the package are published by the package *__init__.py* file so that the functionality of itertree can be reached by importing:

```
>>> from itertree import *
```

Note: This import is a precondition for all shown code examples in this documentation.

The itertree trees are build by adding *iTree*-objects to a *iTree*-parent-object. This means we do not have an external tree generator the tree is build by using the appending functionalities of the objects itself.

We start now building an itertree with the recommended method for adding items *append()*. The user might use the lazy way via += operator (*__iadd__()*) too. Both operations will add a child item at the end of the parent sub-tree (like *append()* in lists).

```
>>> root = iTree('root')  # first we create a root element (parent)
>>> root.append(iTree(tag='child', value=0))  # add a child append method
iTree('child', value=0)
>>> root.append(iTree((1, 2, 3), 1))  # add next child (the given tag is tuple, any␣
↪hashable object can be used as tag)
iTree((1, 2, 3), value=1)
>>> root += iTree(tag='child2', value=2)  # next child could be added via += operator␣
↪too
>>> root.render()  # show the created tree
iTree('root')
 > iTree('child', value=0)
 > iTree((1, 2, 3), value=1)
 > iTree('child2', value=2)
```

Each *iTree*-object has a tag which is the main part of the identifier of the object. For tags you can use any type of hashable objects.

Different than the keys in dictionaries the given tags must not be unique! The user should understand that in general *iTree*-objects behave more like nested lists than nested dicts:

```
>>> root.append(iTree('child', 5))
iTree('child', value=5)
>>> root.append(iTree('child', 6))
iTree('child', value=6)
>>> root.render()
iTree('root')
 > iTree('child', value=0)
 > iTree((1, 2, 3), value=1)
 > iTree('child2', value=2)
 > iTree('child', value=5)
 > iTree('child', value=6)
```

In the *iTree* object equal tags are enumerated in a tag-family and they can be targeted via a tag-index-pair (family-tag,family-index). In the "wording" of *iTree* this pair is named a **key** because it is unique like the keys in dicts. To summarize the items in an *iTree* can be accessed via absolute index (like in lists) or they can be reached by giving the key (tag-index-pair) which is comparable to the key in dicts (both ways are very quick).

```
>>> print(root['child', 1])  # target via key -> tag_idx pair
iTree('child', value=5)
```

(continues on next page)

(continued from previous page)

```
>>> print(root[3])  # target via absolute index
iTree('child', value=5)
```

E.g.: To add sub-items we can address the child item also by absolute index and add a sub-item.

```
>>> root[0].append(iTree('subchild'))
iTree('subchild')
>>> print(root[0][0])
iTree('subchild')
```

After the tree is generated we can iterate over the tree:

```
>>> a = [i for i in root]
>>> len(a)
5
>>> print(a)
[iTree('child', value=0, subtree=[iTree('subchild')]), iTree((1, 2, 3), value=1),␣
↪iTree('child2', value=2), iTree('child', value=5), iTree('child', value=6)]
>>> b = list(root.deep)  # The list is build by iterating over all nested children
>>> len(b)  # The item: root[0][0] is considered in this iteration too
6
>>> print(b)
[iTree('child', value=0, subtree=[iTree('subchild')]), iTree('subchild'), iTree((1, 2,
↪ 3), value=1), iTree('child2', value=2), iTree('child', value=5), iTree('child',␣
↪value=6)]
```

As shown in the example we have the possibility to iterate over the first level only (children) or we use the internal class absolute index (like in lists) or they can be reached by giving the key (tag-index-pair) which is comparable to the key in dicts (both ways are very quick).

```
>>> print(root['child', 1])  # target via key -> tag_idx pair
iTree('child', value=5)
>>> print(root[3])  # target via absolute index
iTree('child', value=5)
```

E.g.: To add sub-items we can address the child item also by absolute index and add a sub-item.

```
>>> root[0].append(iTree('subchild'))
iTree('subchild')
>>> print(root[0][0])
iTree('subchild')
```

Many iterable methods have a *filter_method* parameter in which a filtering method can be placed that targets specific properties of the items.

```
>>> # ----> filtering method can be placed that targets specific properties of the␣
↪items.
>>> a = [i for i in root.deep.iter(filter_method=lambda i: type(i.value) is int and i.
↪value % 2 == 0)]  # search even data items
>>> print(a)
[iTree('child', value=0, subtree=[iTree('subchild'), iTree('subchild')]), iTree(
↪'child2', value=2), iTree('child', value=6)]
```

In case no value is given the *iTree* will take automatically the *itertree.NoValue* object as value. In case an *iTree* is instanced without tag the tag value *itertree.NoTag* will be used.

```
>>> empty_item = iTree()
>>> print(empty_item)
iTree()
>>> print(empty_item.tag)
<class 'itertree.itree_helpers.NoTag'>
>>> print(empty_item.value)
<class 'itertree.itree_helpers.NoValue'>
```

At least the itertree can be stored and reconstructed from a file. We can also link an item to a specific item in a file (external link) or create internal links.

```
>>> root.dump('dt.itz',overwrite=True)  # itz is the recommended file ending for the␣
↪zipped dataset file
9cd3a9a644af51ea94c82f64ca4ccf745b4a1dd717958beec0cfeb9b0647ba73
>>> root2 = iTree().load('dt.itz')  # loading a iTree from a file
>>> print(root2 == root)
True
>>> root += iTree('link', link=iTLink('dt.itz',[('child', 0)]))  # The node item will␣
↪integrate the children of the linked item.
```

7.5 iTree-Generators vs. lists

As the package name itertree suggests we have several possibilities to iterate over the tree items. The related functions are realized internally via generators. We have generators targeting the children only (level 1) and we have others which ran in-depth into the whole tree structure targeting all the internal items (children, sub-children,...). The provided generators can be easily casted into real iterators via build-in *iter()*-method (most often the cast is not required, if target method takes generators (uses *__iter__()*)).

If *mytree* is an *iTree*-object e.g. you can iterate via:

- iter(mytree) - level 1 iterator over all children delivers the items
- iter(mytree.keys()) - level 1 iterator over all children delivers the tag-idx of the items
- iter(mytree.values()) - level 1 iterator over all children delivers the values of the items
- iter(mytree.items()) - level 1 iterator over all children delivers the (tag_idx,item) pair of the items
- iter(mytree.deep) - flatten iterator over all in-depth items in the tree delivers the items
- iter(mytree.deep.tag_idx_path()) - flatten iterator over all in-depth items delivers the (tag-idx-path,item) pair
- iter(mytree.deep.idx_path()) - flatten iterator over all in-depth items delivers the (abs. index,item) pair

- mytree.get.iter(*target_path) - delivers an iterator over all items targeted via target_path (multi item target)

The usage of generators (iterators) give some big advantages over the usage of lists related to performance and memory consumption. The main idea is to combine all the filtering and iterable options together before you start the final iteration (consume the iter-generator). The instancing of generators/iterators is is very quick and independent from the number of items the object wil iter over. E.g. if the user would casts the inbetween results of multiple operations into 'list'-objects it would take relative long time and the memory consumption would be much more. Therefore it is recommended to build (cascade) all required operations based on the given generator/iterator object. And only at the very end we should consume the generator/iterator. If the code is build like this it is very quick and needs less memory. So please avoid type casts to lists in between the operations. It is very helpful if the user have a look at the powerful *itertools*-package which can be utilized for those proposes.

If the user really wants to to end-up in a *list*-object he can easy cast the generator by using the *list()* statement (The cast might be needed for list related functionalities like *len()*):

Related to generators/iterators the user should know:

- The StopIteration exception must be handled in case of empty generators.
- An generator can be consumed only one time. To reuse an generator multiple times you may have a look at *itertools.tee()*.

Here are some possible usages of the iteration functions in itertree (imagine large trees for small trees the example operations are equivalent):

```
>>> myresultlist = list(root.deep)  # this is quick even for huge number of items
>>> first_item = list(root.deep)[0]  # but this is slower (list-type-cast)  as:
>>> first_item = next(iter(root.deep)) # create an iterator from the generator object
>>> fifth_item = list(root.deep)[4]  # and this is slower as:
>>> import itertools
>>> fifth_item = next(itertools.islice(root.deep, 4, None))
```

PYTHON MODULE INDEX

i

itertree.itree_data, 112
itertree.itree_filters, 126
itertree.itree_getitem, 111
itertree.itree_helpers, 139
itertree.itree_indepth, 112
itertree.itree_main, 89
itertree.itree_mathsets, 130
itertree.itree_serializer.itree_json_converter, 138
itertree.itree_serializer.itree_json_serialize, 135
itertree.itree_serializer.itree_render_dot, 138
itertree.itree_serializer.itree_renderer, 137

INDEX

Symbols

__delitem__() (*itertree.itree_main.iTree method*), 102
__eq__() (*itertree.itree_main.iTree method*), 105
__getitem__() (*itertree.itree_main.iTree method*), 104
__init__() (*itertree.itree_main.iTree method*), 89
__iter__ (*itertree.itree_main.iTree attribute*), 92
__setitem__() (*itertree.itree_main.iTree method*), 100

A

accu_iterator() (*in module itertree.itree_helpers*), 140
ALL_TYPE (*itertree.itree_serializer.itree_json_serialize.iTStdJSONSerializer2 attribute*), 135
Any (*class in itertree.itree_helpers*), 141
ANY (*itertree.itree_serializer.itree_json_serialize.iTStdJSONSerializer2 attribute*), 135
append() (*itertree.itree_main.iTree method*), 98
appendleft() (*itertree.itree_main.iTree method*), 98
ArgTuple (*class in itertree.itree_helpers*), 141

B

built-in function
    itertree.iTree.deep.__contains__(), 46
    itertree.iTree.deep.__iter__(), 60
    itertree.iTree.deep.__len__(), 49
    itertree.iTree.deep.count(), 48
    itertree.iTree.deep.filtered_len(), 49
    itertree.iTree.deep.idx_paths(), 61
    itertree.iTree.deep.index(), 47
    itertree.iTree.deep.is_in(), 47
    itertree.iTree.deep.is_tag_in(), 48
    itertree.iTree.deep.iter(), 60
    itertree.iTree.deep.iter_family_items(), 63
    itertree.iTree.deep.reverse(), 24
    itertree.iTree.deep.sort(), 24
    itertree.iTree.deep.tag_idx_paths(), 62
    itertree.iTree.get(), 28
    itertree.iTree.get.by_idx(), 33
    itertree.iTree.get.by_idx_list(), 34
    itertree.iTree.get.by_idx_slice(), 33
    itertree.iTree.get.by_level_filter(), 36
    itertree.iTree.get.by_tag(), 36
    itertree.iTree.get.by_tag_idx(), 34
    itertree.iTree.get.by_tag_idx_list(), 35
    itertree.iTree.get.by_tag_idx_slice(), 35
    itertree.iTree.get.by_tags(), 36
    itertree.iTree.get.iter(), 31
    itertree.iTree.get.single(), 30
by_idx() (*in module itertree.itree_getitem._iTreeGetitem*), 33
by_idx_list() (*in module itertree.itree_getitem._iTreeGetitem*), 34
by_idx_slice() (*in module itertree.itree_getitem._iTreeGetitem*), 33
by_level_filter() (*in module itertree.itree_getitem._iTreeGetitem*), 36
by_tag() (*in module itertree.itree_getitem._iTreeGetitem*), 36
by_tag_idx() (*in module itertree.itree_getitem._iTreeGetitem*), 34
by_tag_idx_list() (*in module itertree.itree_getitem._iTreeGetitem*), 35
by_tag_idx_slice() (*in module itertree.itree_getitem._iTreeGetitem*), 35
by_tags() (*in module itertree.itree_getitem._iTreeGetitem*), 36
BYTE_TYPE (*itertree.itree_serializer.itree_json_serialize.iTStdJSONSerial attribute*), 135

C

cardinality() (*itertree.itree_mathsets.mSetCombine property*), 134

cardinality() (*itertree.itree_mathsets.mSetInterval property*), 132
cardinality() (*itertree.itree_mathsets.mSetRoster property*), 133
check() (*itertree.itree_data.iTDataModel method*), 121
check_and_cast_single_item() (*itertree.itree_data.iTAnyValueModel method*), 116
check_and_cast_single_item() (*itertree.itree_data.iTASCIIStrModel method*), 120
check_and_cast_single_item() (*itertree.itree_data.iTEnumerateModel method*), 121
check_and_cast_single_item() (*itertree.itree_data.iTFloatModel method*), 118
check_and_cast_single_item() (*itertree.itree_data.iTInt8Model method*), 117
check_and_cast_single_item() (*itertree.itree_data.iTIntModel method*), 116
check_and_cast_single_item() (*itertree.itree_data.iTRoundIntModel method*), 116
check_and_cast_single_item() (*itertree.itree_data.iTStrModel method*), 119
check_and_cast_single_item() (*itertree.itree_data.iTUTF16StrModel method*), 120
check_and_cast_single_item() (*itertree.itree_data.iTUTF8StrModel method*), 120
check_and_cast_single_item() (*itertree.itree_data.iTValueModel method*), 114
clear() (*itertree.itree_data.iTData method*), 123
clear() (*itertree.itree_data.iTDataModel method*), 121
clear() (*itertree.itree_data.iTDataReadOnly method*), 125
clear() (*itertree.itree_data.iTValueModel method*), 115
clear() (*itertree.itree_main.iTree method*), 103
contains() (*itertree.itree_data.iTValueModel property*), 115
convert() (*itertree.itree_serializer.itree_json_converter.Converter_1_1_1_to_2_0_0_Cls method*), 139
CONVERT_FROM_JSON_MAP (*itertree.itree_serializer.itree_json_serialize.iTStdJSONSerializer2 attribute*), 136
convert_from_json_obj() (*itertree.itree_serializer.itree_json_serialize.iTStdJSONSerializer method*), 136
convert_it_type() (*itertree.itree_serializer.itree_json_serialize.iTStdJSONSerializer method*), 136
CONVERT_MAP (*itertree.itree_serializer.itree_json_serialize.iTStdJSONSer attribute*), 136
convert_numpy() (*itertree.itree_serializer.itree_json_serialize.iTStdJS method*), 136
convert_single_itree_to_json_obj() (*itertree.itree_serializer.itree_json_serialize.iTStdJSONSerializer method*), 136
convert_single_itree_to_json_obj2() (*itertree.itree_serializer.itree_json_serialize.iTStdJSONSerializer method*), 136
convert_to_json_item() (*itertree.itree_serializer.itree_json_serialize.iTStdJSONSerializer method*), 136
Converter_1_1_1_to_2_0_0() (*in module itertree.itree_serializer.itree_json_converter*), 139
Converter_1_1_1_to_2_0_0_Cls (*class in itertree.itree_serializer.itree_json_converter*), 139
copy() (*itertree.itree_data.iTData method*), 123
copy() (*itertree.itree_main.iTree method*), 104
copy_keep_value() (*itertree.itree_main.iTree method*), 104
count() (*itertree.itree_main.iTree method*), 106
coupled_object() (*itertree.itree_main.iTree property*), 97
create_itree_from_raw() (*itertree.itree_serializer.itree_json_converter.Converter_1_1_1_to method*), 139
create_itree_from_raw() (*itertree.itree_serializer.itree_json_serialize.iTStdJSONSerializer method*), 136
create_itree_from_raw2() (*itertree.itree_serializer.itree_json_serialize.iTStdJSONSerializer method*), 136

D

DATA (*itertree.itree_serializer.itree_json_converter.Converter_1_1_1_to_2_ attribute*), 139
DATA_CONTAINER (*itertree.itree_serializer.itree_json_converter.Converte attribute*), 139
DATA_MODELL (*itertree.itree_serializer.itree_json_converter.Converter_1_ attribute*), 139
deep() (*itertree.itree_main.iTree property*), 94
deepcopy() (*itertree.itree_data.iTData method*), 124
deepcopy() (*itertree.itree_main.iTree method*), 105
del_key_value() (*itertree.itree_main.iTree method*), 97

del_value() (*itertree.itree_main.iTree method*), 97
delete_item() (*itertree.itree_data.iTData method*), 124
delete_item() (*itertree.itree_data.iTDataReadOnly method*), 125
description() (*itertree.itree_data.iTValueModel property*), 115
DICT_TYPE (*itertree.itree_serializer.itree_json_serialize.iTStdJSONSerializer2 attribute*), 135
DTYPE (*itertree.itree_serializer.itree_json_converter.Converter_1_1_1_to_2_0_0 attribute*), 139
dump() (*itertree.itree_main.iTree method*), 109
dump() (*itertree.itree_serializer.itree_json_serialize.iTStdJSONSerializer2 method*), 136
dumps() (*itertree.itree_main.iTree method*), 109
dumps() (*itertree.itree_serializer.itree_json_serialize.iTStdJSONSerializer2 method*), 136

E

equal() (*itertree.itree_main.iTree method*), 105
extend() (*itertree.itree_main.iTree method*), 99
extendleft() (*itertree.itree_main.iTree method*), 99

F

file_crc() (*itertree.itree_helpers.iTLink property*), 140
file_path() (*itertree.itree_helpers.iTLink property*), 140
filter() (*itertree.itree_mathsets.mSetCombine method*), 134
filter() (*itertree.itree_mathsets.mSetInterval method*), 132
filter() (*itertree.itree_mathsets.mSetRoster method*), 133
filtered_len() (*itertree.itree_main.iTree method*), 105
flags() (*itertree.itree_main.iTree property*), 94
flags_repr() (*itertree.itree_main.iTree method*), 94
force_cache_update() (*itertree.itree_main.iTree method*), 93
formatter() (*itertree.itree_data.iTDataModel method*), 122
formatter() (*itertree.itree_data.iTDataModelAny method*), 122
formatter() (*itertree.itree_data.iTValueModel property*), 115
formatter() (*itertree.itree_mathsets.mSetItem property*), 131
formatter_type() (*itertree.itree_mathsets.mSetItem property*), 131
fromkeys() (*itertree.itree_data.iTData method*), 124

G

get (*itertree.itree_main.iTree attribute*), 111
get() (*itertree.itree_data.iTData method*), 123
get() (*itertree.itree_data.iTDataModel method*), 121
get() (*itertree.itree_data.iTValueModel method*), 115
get_args() (*itertree.itree_helpers.iTLink method*), 140
get_init_args() (*itertree.itree_data.iTData method*), 124
get_init_args() (*itertree.itree_data.iTDataModel method*), 122
get_init_args() (*itertree.itree_data.iTDataModelAny method*), 123
get_init_args() (*itertree.itree_data.iTDataReadOnly method*), 126
get_init_args() (*itertree.itree_data.iTInt8Model method*), 117
get_init_args() (*itertree.itree_data.iTStrFnPatternModel method*), 119
get_init_args() (*itertree.itree_data.iTStrRegexPatternModel method*), 119
get_init_args() (*itertree.itree_data.iTValueModel method*), 116
get_init_args() (*itertree.itree_helpers.iTLink method*), 140
get_init_args() (*itertree.itree_main.iTree method*), 108
get_init_args() (*itertree.itree_mathsets.mSetCombine method*), 134
get_init_args() (*itertree.itree_mathsets.mSetInterval method*), 132
get_init_args() (*itertree.itree_mathsets.mSetItem method*), 131
get_init_args() (*itertree.itree_mathsets.mSetRoster method*), 133
get_key_value() (*itertree.itree_main.iTree method*), 97
GET_LOOK_UP_METHOD (*itertree.itree_data.iTData attribute*), 123
get_target_tree() (*itertree.itree_helpers.iTLink method*), 140
get_value() (*itertree.itree_main.iTree method*), 96
getitem_by_idx (*itertree.itree_main.iTree attribute*), 111
getter_to_list() (*in module itertree.itree_helpers*), 141

H

has_item_flags (*class in itertree.itree_filters*), 126
has_item_flags() (*in module itertree.itree_filters*), 75
has_item_tag_fnmatch (*class in itertree.itree_filters*), 127
has_item_tag_fnmatch() (*in module itertree.itree_filters*), 75
has_item_value (*class in itertree.itree_filters*), 127

has_item_value() (*in module itertree.itree_filters*), 75
has_item_value_dict_key (*class in itertree.itree_filters*), 129
has_item_value_dict_key() (*in module itertree.itree_filters*), 77
has_item_value_dict_key_fnmatch (*class in itertree.itree_filters*), 129
has_item_value_dict_key_fnmatch() (*in module itertree.itree_filters*), 77
has_item_value_dict_key_in (*class in itertree.itree_filters*), 130
has_item_value_dict_key_in() (*in module itertree.itree_filters*), 78
has_item_value_dict_value (*class in itertree.itree_filters*), 127
has_item_value_dict_value() (*in module itertree.itree_filters*), 76, 77
has_item_value_dict_value_fnmatch (*class in itertree.itree_filters*), 128
has_item_value_dict_value_fnmatch() (*in module itertree.itree_filters*), 76
has_item_value_dict_value_in (*class in itertree.itree_filters*), 128
has_item_value_fnmatch (*class in itertree.itree_filters*), 128
has_item_value_fnmatch() (*in module itertree.itree_filters*), 76
has_item_value_list_idx (*class in itertree.itree_filters*), 129
has_item_value_list_idx() (*in module itertree.itree_filters*), 77
has_item_value_list_item_fnmatch (*class in itertree.itree_filters*), 128
has_item_value_list_item_fnmatch() (*in module itertree.itree_filters*), 76
has_item_value_list_item_in (*class in itertree.itree_filters*), 129
has_item_value_list_value (*class in itertree.itree_filters*), 127
has_item_value_list_value() (*in module itertree.itree_filters*), 76, 77

I

IDX (*itertree.itree_serializer.itree_json_converter.Converter_1_1_1_to_2_0_0_Cls attribute*), 139
idx() (*itertree.itree_helpers.TagIdx property*), 141
idx() (*itertree.itree_main.iTree property*), 92
idx_path() (*itertree.itree_main.iTree property*), 93
index() (*itertree.itree_main.iTree method*), 106
insert() (*itertree.itree_main.iTree method*), 98
interval (*itertree.itree_data.iTInt16Model attribute*), 117
interval (*itertree.itree_data.iTInt32Model attribute*), 118
interval (*itertree.itree_data.iTInt64Model attribute*), 118
interval (*itertree.itree_data.iTInt8Model attribute*), 117
interval (*itertree.itree_data.iTUInt16Model attribute*), 118
interval (*itertree.itree_data.iTUInt32Model attribute*), 118
interval (*itertree.itree_data.iTUInt64Model attribute*), 118
interval (*itertree.itree_data.iTUInt8Model attribute*), 117
is_complement() (*itertree.itree_mathsets.mSetItem property*), 131
is_empty() (*itertree.itree_data.iTData property*), 124
is_empty() (*itertree.itree_data.iTDataModel property*), 121
is_empty_set() (*itertree.itree_mathsets.mSetCombine property*), 134
is_empty_set() (*itertree.itree_mathsets.mSetInterval property*), 132
is_empty_set() (*itertree.itree_mathsets.mSetRoster property*), 133
is_empty_set_complement() (*itertree.itree_mathsets.mSetCombine property*), 134
is_empty_set_complement() (*itertree.itree_mathsets.mSetInterval property*), 132
is_empty_set_complement() (*itertree.itree_mathsets.mSetRoster property*), 133
is_file_updated() (*itertree.itree_helpers.iTLink method*), 140
is_in() (*itertree.itree_main.iTree method*), 105
is_int_only() (*itertree.itree_mathsets.mSetInterval property*), 132
is_intersection() (*itertree.itree_mathsets.mSetCombine property*), 134
is_iTData() (*itertree.itree_data.iTData property*), 124
is_iTDataModel() (*itertree.itree_data.iTDataModel property*), 121
is_item_tag (*class in itertree.itree_filters*), 127
is_item_tag() (*in module itertree.itree_filters*), 75
is_item_value_in (*class in itertree.itree_filters*), 128
is_item_value_in() (*in module itertree.itree_filters*), 76
is_iterator_empty() (*in module itertree.itree_helpers*), 140

is_iTValueModel() (*itertree.itree_data.iTValueModel property*), 115
is_key_empty() (*itertree.itree_data.iTData method*), 124
is_link_cover() (*itertree.itree_main.iTree property*), 110
is_link_loaded() (*itertree.itree_main.iTree property*), 110
is_link_root() (*itertree.itree_main.iTree property*), 110
is_linked() (*itertree.itree_main.iTree property*), 110
is_loaded() (*itertree.itree_helpers.iTLink property*), 140
is_lower_closed() (*itertree.itree_mathsets.mSetInterval property*), 132
is_lower_open() (*itertree.itree_mathsets.mSetInterval property*), 132
is_mSetItem() (*itertree.itree_mathsets.mSetItem property*), 131
is_no_key_only() (*itertree.itree_data.iTData property*), 124
is_placeholder() (*itertree.itree_main.iTree property*), 110
is_root() (*itertree.itree_main.iTree property*), 92
is_tag_in() (*itertree.itree_main.iTree method*), 105
is_tree_read_only() (*itertree.itree_main.iTree property*), 94
is_union() (*itertree.itree_mathsets.mSetCombine property*), 134
is_upper_closed() (*itertree.itree_mathsets.mSetInterval property*), 132
is_upper_open() (*itertree.itree_mathsets.mSetInterval property*), 132
is_value_read_only() (*itertree.itree_main.iTree property*), 95
is_var() (*itertree.itree_mathsets.mSetItem property*), 131
IT_CONST_TYPE (*itertree.itree_serializer.itree_json_serialize.iTStdJSONSerialize2 attribute*), 135
IT_LINK_TYPE (*itertree.itree_serializer.itree_json_serialize.iTStdJSONSerialize2 attribute*), 135
IT_TYPE (*itertree.itree_serializer.itree_json_serialize.iTStdJSONSerialize2 attribute*), 135
iTAnyValueModel (*class in itertree.itree_data*), 116
iTASCIIStrModel (*class in itertree.itree_data*), 120
iTData (*class in itertree.itree_data*), 123
iTDataModel (*class in itertree.itree_data*), 121
iTDataModelAny (*class in itertree.itree_data*), 122
iTDataReadOnly (*class in itertree.itree_data*), 124
iTDataTypeError, 121
iTDataValueError, 121
items() (*itertree.itree_main.iTree method*), 106
items() (*itertree.itree_mathsets.mSetCombine method*), 134
items() (*itertree.itree_mathsets.mSetRoster method*), 133
iTEnumerateModel (*class in itertree.itree_data*), 121
iter_families() (*itertree.itree_main.iTree method*), 107
iter_family_items() (*itertree.itree_main.iTree method*), 107
iter_in() (*itertree.itree_mathsets.mSetCombine method*), 134
iter_in() (*itertree.itree_mathsets.mSetInterval method*), 132
iter_in() (*itertree.itree_mathsets.mSetRoster method*), 133
iter_items_over_filter_method() (*in module itertree.itree_filters*), 126
ITER_TYPE (*itertree.itree_serializer.itree_json_serialize.iTStdJSONSerialize2 attribute*), 135
itertree.iTree.deep.__contains__() built-in function, 46
itertree.iTree.deep.__iter__() built-in function, 60
itertree.iTree.deep.__len__() built-in function, 49
itertree.iTree.deep.count() built-in function, 48
itertree.iTree.deep.filtered_len() built-in function, 49
itertree.iTree.deep.idx_paths() built-in function, 61
itertree.iTree.deep.index() built-in function, 47
itertree.iTree.deep.is_in() built-in function, 47
itertree.iTree.deep.is_tag_in() built-in function, 48
itertree.iTree.deep.iter() built-in function, 60
itertree.iTree.deep.iter_family_items() built-in function, 63
itertree.iTree.deep.reverse() built-in function, 24
itertree.iTree.deep.sort() built-in function, 24
itertree.iTree.deep.tag_idx_paths() built-in function, 62
itertree.iTree.get() built-in function, 28
itertree.iTree.get.by_idx() built-in function, 33
itertree.iTree.get.by_idx_list() built-in function, 34

itertree.iTree.get.by_idx_slice() built-in function, 33
itertree.iTree.get.by_level_filter() built-in function, 36
itertree.iTree.get.by_tag() built-in function, 36
itertree.iTree.get.by_tag_idx() built-in function, 34
itertree.iTree.get.by_tag_idx_list() built-in function, 35
itertree.iTree.get.by_tag_idx_slice() built-in function, 35
itertree.iTree.get.by_tags() built-in function, 36
itertree.iTree.get.iter() built-in function, 31
itertree.iTree.get.single() built-in function, 30
itertree.itree_data module, 112
itertree.itree_filters module, 126
itertree.itree_getitem module, 111
itertree.itree_helpers module, 139
itertree.itree_indepth module, 112
itertree.itree_main module, 89
itertree.itree_mathsets module, 130
itertree.itree_serializer.itree_json_converter module, 138
itertree.itree_serializer.itree_json_serialize module, 135
itertree.itree_serializer.itree_render_dot module, 138
itertree.itree_serializer.itree_renderer module, 137
iTFLAG (*class in itertree.itree_helpers*), 140
iTFloatModel (*class in itertree.itree_data*), 118
iTInt16Model (*class in itertree.itree_data*), 117
iTInt32Model (*class in itertree.itree_data*), 118
iTInt64Model (*class in itertree.itree_data*), 118
iTInt8Model (*class in itertree.itree_data*), 117
iTIntModel (*class in itertree.itree_data*), 116
iTLink (*class in itertree.itree_helpers*), 140
iTree (*class in itertree*), 16
iTree (*class in itertree.itree_main*), 89
ITREE_ITEMS_DECODE (*itertree.itree_serializer.itree_json_converter.Converter_1_1_1_to_2_0_0_Cls attribute*), 139
ITREE_SERIALIZE_VERSION (*itertree.itree_serializer.itree_json_converter.Converter_1_1_1_to attribute*), 139
iTreeRender (*class in itertree.itree_serializer.itree_renderer*), 137
iTRoundIntModel (*class in itertree.itree_data*), 116
iTStdJSONSerializer2 (*class in itertree.itree_serializer.itree_json_serialize*), 135
iTStrFnPatternModel (*class in itertree.itree_data*), 119
iTStrModel (*class in itertree.itree_data*), 119
iTStrRegexPatternModel (*class in itertree.itree_data*), 119
iTUInt16Model (*class in itertree.itree_data*), 117
iTUInt32Model (*class in itertree.itree_data*), 118
iTUInt64Model (*class in itertree.itree_data*), 118
iTUInt8Model (*class in itertree.itree_data*), 117
iTUTF16StrModel (*class in itertree.itree_data*), 120
iTUTF8StrModel (*class in itertree.itree_data*), 120
iTValueModel (*class in itertree.itree_data*), 113
iTValueModel() (*in module itertree.Data*), 78

K

keys() (*itertree.itree_main.iTree method*), 106

L

last_except() (*itertree.itree_data.iTValueModel property*), 115
level() (*itertree.itree_main.iTree property*), 94
LINK (*itertree.itree_serializer.itree_json_converter.Converter_1_1_1_to_2 attribute*), 139
link_data() (*itertree.itree_helpers.iTLink property*), 140
link_item() (*itertree.itree_helpers.iTLink property*), 140
link_root() (*itertree.itree_main.iTree property*), 110
link_tag() (*itertree.itree_helpers.iTLink property*), 140
load() (*itertree.itree_main.iTree method*), 109
load() (*itertree.itree_serializer.itree_json_serialize.iTStdJSONSerializer2 method*), 137
LOAD_LINKS (*itertree.itree_helpers.iTFLAG attribute*), 141
load_links() (*itertree.itree_main.iTree method*), 110
loaded() (*itertree.itree_helpers.iTLink property*), 140
loads() (*itertree.itree_main.iTree method*), 109
loads() (*itertree.itree_serializer.itree_json_serialize.iTStdJSONSerialize method*), 136
lower_value() (*itertree.itree_mathsets.mSetInterval property*), 132

M

make_local() (*itertree.itree_main.iTree method*), 111

math_repr() (*itertree.itree_mathsets.mSetCombine method*), 134
math_repr() (*itertree.itree_mathsets.mSetInterval method*), 132
math_repr() (*itertree.itree_mathsets.mSetItem method*), 131
math_repr() (*itertree.itree_mathsets.mSetRoster method*), 133
max_depth() (*itertree.itree_main.iTree property*), 94
model_items() (*itertree.itree_data.iTData method*), 124
model_values() (*itertree.itree_data.iTData method*), 124
module
itertree.itree_data, 112
itertree.itree_filters, 126
itertree.itree_getitem, 111
itertree.itree_helpers, 139
itertree.itree_indepth, 112
itertree.itree_main, 89
itertree.itree_mathsets, 130
itertree.itree_serializer.itree_json_converter, 138
itertree.itree_serializer.itree_json_serialize, 135
itertree.itree_serializer.itree_render_dot, 138
itertree.itree_serializer.itree_renderer, 137
move() (*itertree.itree_main.iTree method*), 101
mSetCombine (*class in itertree.itree_mathsets*), 133
mSetCombine() (*in module itertree.itree_mathsets*), 81
mSetInterval (*class in itertree.itree_mathsets*), 131
mSetInterval() (*in module itertree.itree_mathsets*), 81
mSetItem (*class in itertree.itree_mathsets*), 130
mSetRoster (*class in itertree.itree_mathsets*), 133
mSetRoster() (*in module itertree.itree_mathsets*), 81

N

NO_KEY (*itertree.itree_serializer.itree_json_serialize.iTStdJSONSerializer2 attribute*), 135
NO_TAG (*itertree.itree_serializer.itree_json_serialize.iTStdJSONSerializer2 attribute*), 135
NO_VALUE (*itertree.itree_serializer.itree_json_serialize.iTStdJSONSerializer2 attribute*), 135
NoKey (*class in itertree.itree_helpers*), 141
not_linked_filter() (*itertree.itree_serializer.itree_json_serialize.iTStdJSONSerializer2 method*), 135
NoTag (*class in itertree.itree_helpers*), 141
NoTarget (*class in itertree.itree_helpers*), 141
NoValue (*class in itertree.itree_helpers*), 141
NP_TYPE (*itertree.itree_serializer.itree_json_serialize.iTStdJSONSerializer2 attribute*), 135

P

parent() (*itertree.itree_main.iTree property*), 92
pattern() (*itertree.itree_data.iTStrFnPatternModel property*), 119
pattern() (*itertree.itree_data.iTStrRegexPatternModel property*), 119
pop() (*itertree.itree_data.iTData method*), 123
pop() (*itertree.itree_data.iTDataReadOnly method*), 125
pop() (*itertree.itree_main.iTree method*), 103
post_item() (*itertree.itree_main.iTree property*), 94
pre_item() (*itertree.itree_main.iTree property*), 93

R

READ_ONLY_TREE (*itertree.itree_helpers.iTFLAG attribute*), 140
READ_ONLY_VALUE (*itertree.itree_helpers.iTFLAG attribute*), 141
remove() (*itertree.itree_main.iTree method*), 103
rename() (*itertree.itree_main.iTree method*), 101
render() (*itertree.itree_main.iTree method*), 108
renders() (*itertree.itree_main.iTree method*), 108
renders() (*itertree.itree_serializer.itree_renderer.iTreeRender method*), 137
reverse() (*itertree.itree_main.iTree method*), 101
rindex() (*in module itertree.itree_helpers*), 140
root() (*itertree.itree_main.iTree property*), 92
rotate() (*itertree.itree_main.iTree method*), 101

S

set() (*itertree.itree_data.iTDataModel method*), 121
set() (*itertree.itree_data.iTValueModel method*), 115
set_coupled_object() (*itertree.itree_main.iTree method*), 97
set_description() (*itertree.itree_data.iTValueModel method*), 115
set_formatter() (*itertree.itree_data.iTValueModel method*), 115
set_key_value() (*itertree.itree_main.iTree method*), 95
set_loaded() (*itertree.itree_helpers.iTLink method*), 140
set_source_path() (*itertree.itree_helpers.iTLink method*), 140
set_tags_and_keys() (*itertree.itree_helpers.iTLink method*), 140
set_tree_read_only() (*itertree.itree_main.iTree method*), 95
set_value() (*itertree.itree_main.iTree method*), 95

set_value_read_only() (*itertree.itree_main.iTree method*), 95
single() (*in module itertree.itree_getitem._iTreeGetitem*), 85
sort() (*itertree.itree_main.iTree method*), 102
source_path() (*itertree.itree_helpers.iTLink property*), 140
STR_TYPE (*itertree.itree_serializer.itree_json_serialize.iTStdJSONSerializer2 attribute*), 135

T

Tag (*class in itertree.itree_helpers*), 141
tag (*itertree.itree_helpers.Tag attribute*), 141
TAG (*itertree.itree_serializer.itree_json_converter.Converter_1_1_1_to_2_0_0_Cls attribute*), 139
tag() (*itertree.itree_helpers.TagIdx property*), 141
tag() (*itertree.itree_main.iTree property*), 92
tag_idx() (*itertree.itree_main.iTree property*), 93
tag_idx_path() (*itertree.itree_main.iTree property*), 93
tag_number() (*itertree.itree_main.iTree property*), 94
TagIdx (*class in itertree.itree_helpers*), 141
tags() (*itertree.itree_helpers.iTLink property*), 140
tags() (*itertree.itree_main.iTree method*), 107
target_path() (*itertree.itree_helpers.iTLink property*), 140
TRANSLATE_KEY2OBJ (*itertree.itree_serializer.itree_json_serialize.iTStdJSONSerializer2 attribute*), 135
TRANSLATE_OBJ2KEY (*itertree.itree_serializer.itree_json_serialize.iTStdJSONSerializer2 attribute*), 135
TREE (*itertree.itree_serializer.itree_json_converter.Converter_1_1_1_to_2_0_0_Cls attribute*), 139

U

unset_tree_read_only() (*itertree.itree_main.iTree method*), 95
unset_value_read_only() (*itertree.itree_main.iTree method*), 95
update() (*itertree.itree_data.iTData method*), 123
update() (*itertree.itree_data.iTDataReadOnly method*), 125
upper_value() (*itertree.itree_mathsets.mSetInterval property*), 132

V

validator() (*itertree.itree_data.iTDataModel method*), 122
validator() (*itertree.itree_data.iTDataModelAny method*), 122
value() (*itertree.itree_data.iTDataModel property*), 121
value() (*itertree.itree_data.iTValueModel property*), 115
value() (*itertree.itree_main.iTree property*), 95
value() (*itertree.itree_mathsets.mSetItem property*), 131
values() (*itertree.itree_main.iTree method*), 106