itertree Documentation

Release 0.8.2

B.R.

Jun 09, 2022

CONTENTS

CHAPTER

ONE

CHANGELOG

1.1 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!

1.2 Version 0.7.3

Bugfixes in repr() and render()

Extended examples

Still Beta SW -> but release candidate!

1.3 Version 0.7.2

Improved Interval class (dynamic limits in all levels)

Adapted some tests and the documentation

Still Beta SW -> but release candidate!

1.4 Version 0.7.1

Bigger bugfix on 0.7.0 which was really not well tested!

Still Beta SW -> but release candidate!

1.5 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 get_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!

1.6 Version 0.6.0

Improved interface and performance

Documentation is setup

Testing is improved

Examples still missing

Beta SW!

1.7 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

2.1 Status and compatibility information

The original implementation is done in python 3.5 and it is tested under python 3.5 and 3.9. It should work in all python 3 environments.

The actual development status is released.

2.2 Using the itertree package

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

In this chapter 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. And we hope this chapter orders the things in a much better way.

2.3 Construction of an itertree

The first step in the construction of a itertree is to instance the itertree `iTree class`.

class itertree.**iTree**(*tag*, *data=None*, *subtree=None*)

This is the main class related to iTrees.

This object is the parent of a sub-tree (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.

A iTree object can only be integrated in one tree (one parent only)!

Each iTree object contains a tag. In case your tags are stings it's recommended to use tag strings without wildcards "*","?" and without the standard separators "/" and "#". If you use these characters you might get confusing results in find, filter and match operations.

In general we allow all hashable objects to be used as a tag in the iTree objects (only search operation might be limited in this case). But we have two exceptions: We do not allow integers and TagIdx objects as tags because those objects used for direct item access.

Different than in dictionaries it is allowed to put multiple times the same tag inside the iTree. The items with the same tag are placed and ordered (enumerated) in the related tag-family. They can be reached via TagIdx objects by giving the tag, index pair (tag_idx).

Linked iTree objects will behave different. They have a read only structure (children) and they contain the children (tree) of the linked iTree. The "local" attributes like tag, data, ... can be set independent from the linked item (local properties). To change the tree structure of such an object you must manipulated the source object and reload the link.

Additionally a iTree object can contain:

- data - a iTData object to store any kind of python objects
- couple - you can couple the object to another one by giving a pointer
- is_temporary - you can mark it as temporary. Those iTree items behave like normal ones. But they will not be considered during encoding for storage, etc.

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[] (__getitem__()) call.

More complex access is available via find() and findall() methods. Have a look in the documentation related to each method.

The delivery of access related operations in the iTree objects is for unique targets an iTree object and for multi target operations an iterator over the matching items. We don't deliver something like a list.

If really needed an iterator can be easily converted into a list by list() method but this may take a long time for huge iterators. The iterator should only be used in the final step of the operation. It's recommended to have a look into itertools for better usage of the delivered iterators.

The design of the object is made to have best possible performance even that it is pure python. For more details you may run the performance tests in the test section (But you might have to install additional packages run the comparisons and to get the full picture.)

The function related to iterations iter; iter_children and find_all can be used with an item_filter. By this mechanism you can create queries regarding any property in an iTree.

To initialize the class the following parameters are available

Parameters

- **tag** – tag string or hashable object used for the iTree identification
- **data** – data dict or item to be stored in the node
- **subtree** – The subtree is a iterable structure that contains sub-items (iTree objects) that should be the children of this iTree.

> **Warning:** subtree: In case the given iTree objects have already a parent an implicit copy will be made.

__init__(*tag*, *data=None*, *subtree=None*)

Instance the *iTree* object:

```
>>> item1=iTree('item1') # itertree item with the tag 'item1'
>>> item2=iTree('item2', data={'mykey':1}) # instance a iTree-object with data content␣
↪(defined as a dict)
>>> item3=iTreeTemporary('temp_item') # instance a temporary iTree-object
>>> # instance a iTree-object containing a link:
>>> item4=iTreeLink('linked_item', data={'mykey':2}, link_file_path='dt.itz', link_key_
↪path=iTreeTagIdx('child',0), load_links=True)
```

iTreeTemporary objects can be filtered out and when dumping the whole *iTree* into a file the iTreeTemporary items are ignored and not stored.

In case a link is set by using the iTreeLink class will integrate the childs of the linked iTree-objects as it's own childs into the tree. The *iTree* object can have own properties like temporary or own data. But it can also contain own, local children (see *iTree linked sub-trees*).

To add or manipulate the children of an item we have several possibilities. The following direct operations are recommended for structural manipulations in the tree:

```
>>> root=iTree('root')
>>> root.append(iTree('child')) # append a child
>>> root[0]=iTree('newchild') # replace the child with index 0
>>> del root[iTreeTagIdx('newchild',0)] # deletes the child with matching iTreeTagIdx
```

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

itertree.iTree.**append**(*self*, *item*)

Append the given iTree object to the tree (new last child)

Except
raise TypeError in case iTree object has already a parent

Parameters
item – iTree object to be appended

Returns
True in case append was successful

itertree.iTree.**__iadd__**(*self*, *other*)

Implement self+=value.

itertree.iTree.**appendleft**(*self*, *item*)

Append the given iTree object to the left of the the tree (new first child)

Except
raise TypeError in case iTree object has already a parent

Parameters
item – iTree object to be appended

itertree.itree_main.iTree.**extend**(*self*, *extend_items*)

We extend the iTree with given items (multi append)

Note: In case the extend items have already a parent an implicit copy will be made. We do this because we might get an iTree-object as extend_items parameter and then the children will have automatically a parent even that the parent object might be a temporary one.

Parameters
extend_items – iterable object that contains iTree objects as items

Returns
True

itertree.itree_main.iTree.**extendleft**(*self*, *extend_items*)

We extend the iTree with given items in the beginning (multi appendleft)

Note: In case the extend items have already a parent an implicit copy will be made. We do this because we might get an iTree-object as extend_items parameter and then the children will have automatically a parent even that the parent object might be a temporary one.

Note: The *extendleft()* operation is a lot slower then the normal extend operation

Parameters
extend_items – iterable object that contains iTree objects as items

itertree.itree_main.iTree.**insert**(*self*, *insert_key*, *item*)
Insert an item before a specific position

Parameters
- **insert_key** – position key (integer index or TagIdx)
- **item** – item that should be inserted in the tree (new child)

itertree.itree_main.iTree.**move**(*self*, *insert_key*)
move the item in another position

Parameters
insert_key – item will be insert before this key

itertree.itree_main.iTree.**rename**(*self*, *new_tag*)
give the item a new tag

Parameters
new_tag – new tag object string or hashable object

itertree.itree_main.iTree.**pop**(*self*, *key=- 1*)
pop the item out of the tree, if no key is given the last item will be popped out

Parameters
key – specific identification key for an item (integer index, TagIdx)

Returns
popped out item (parent will be set to None)

itertree.itree_main.iTree.**popleft**(*self*)
pop the first item out of the tree

Returns
popped out item (parent will be set to None)

itertree.iTree.**clear**(*self*)
deletes all children and data! All flags stay unchanged!

The addition of iTrees is possible the result contains always the properties of the first added item and the children of the second added item are appended by creating a copy.

```
>>> a=iTree('a',data={'mykey':1},subtree=[iTree('a1'),iTree('a2')])
>>> b=iTree('b',subtree=[iTree('b1'),iTree('b2')])
>>> c=a+b
>>> c
iTree("'a'", data="{'mykey': 1}", subtree=[iTree("'a1'"), iTree("'a2'"), iTree("'b1'"),␣
↪iTree("'b2'")])
```

(continues on next page)

(continued from previous page)

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
>>> root=iTree('root')
>>> root.extend(itree_list) # we can extend an existing `iTree` with the list (add 1000␣
↪identical children)
True
```

2.4 item access

The items in the *iTree* can be accessed via __getitem__() method:

itertree.iTree.**__getitem__**()

Main getter for items

If given key targets to only one item we will deliver an iTree. If no matching item is found an IndexError or KeyError exception will be raised.

If the given key targets to multiple items (tag family, slice, iterable of single target keys) and iterator will be delivered.

Parameters

key – single target: index, TagIdx or tuple (tag, index) (not recommended) multi target: TagIdx_s; iMatch; slice or an iterable (like list) of these keys

Returns

iTree item or iterator (multi target)

```
>>> root=iTree('root')
>>> root+=iTree('child',data=0)
>>> root+=iTree('child',data=1)
>>> root+=iTree('child',data=2)
>>> root+=iTree('child',data=3)
>>> root+=iTree('child',data=4)
>>> root[1] # index access
iTree("'child'", data=1)
>>> root[TagIdx('child',1)] # TagIdx access (index targets the index in the tag␣
↪family!)
iTree("'child'", data=1)
>>> root[TagIdx('child',-1)] # TagIdx access with negative index
iTree("'child'", data=4)
>>> root[TagIdx('child',[0,2])] # TagIdx give index list -> result is an iterator!
<list_iterator object at 0x0000029E12F69B00>
>>> list(root[TagIdx('child',[0,2])]) # make ietartor content visible by casting␣
↪into a list
[iTree("'child'", data=0), iTree("'child'", data=2)]
>>> list(root[[0,2]]) # index list access (absolute indexes)
[iTree("'child'", data=0), iTree("'child'", data=2)]
>>> list(root[1:3]) # slices are allowed too
[iTree("'child'", data=1), iTree("'child'", data=2)]
```

The TagIdx class is used to address items that contains the same tag. The second argument of the TagIdx is the index that the item has in the related tag-family. But we can also give multiple indexes or a slice. As the given example shows is the result of not unique targets always an iterator object.

itertree.iTree.**get_deep**()

deep key access the function is a replacement for *self[key_list[0]][key_list[1]]...[key_list[-1]]* but you can also feed with an iterator

dives into the tree *key_list=[1,0,2]* -> second element level 1 -> first element level 2 -> third element level 3 -> same as *self[1][0][2]*

Note: Each key in the key list must target to a single item only! E.g. do not use tags here they deliver always a family iterator not a single item (the method will raise an exception). Use index integers or TagIdx objects instead

---

Parameters

key_list – list or iterator of keys (indexes,TagIdx, tuple(tag,index) -> only in case no tuple tags!

Returns

iTree object the key list targets

itertree.iTree.**find**()

The find function targets over multiple levels of the iTree, it returns single items only! This means in case the key_path targets to multiple items the default_return will be given. If the key_path targets to a family with only one item inside or the item_filter extracts only one item in a family the item will be given back as result. For multiple result utilize the *find_all()* method (which is slower).

Note: The method will deliver a default_return when ever in the whole key_path a match is not unique. This means iteration is stopped here and even that a deeper iteration with the defined filtering might deliver at least a unique result. To ensure to find this deeper results you must utilize the slower *find_all()* method.

The key_path parameter given is normally a list. This can be a list of keys or TagIdx objects. The function will search for the first item in the first level, fo next item in the next level and so on...

Absolut and relative key_paths:

If the first item is the separator (default: '/') the find search is like an absolute path and we start at the root of the iTree. For compatibility reasons with find_all we accept a leading "./" (or to be exact: ".%s"#str_path_separator) as absolute path indicator. If the first item is different, the key_path is relative and we start from the actual item and search the children and sub-children.

Single string key_path: If the user searches for string type tags he can use a string with a separator (default: '/') in between the tags (Those key_paths will be implicit translated in a list). An index separator (default = '#') in between the tag and the index can also be used in this case. If the argument is already a list the single keys will not be parsed regarding the str_path_separator.

Note: If iTree contains tags with characters that used for separators or the all match '*' character the find() result might contain that tagged item instead of the expected separated or wildcard match.

Note: Quickest find operations can be performed by giving a list containing index integers or TagIdx objects

---

The parameters in detail:

Parameters

- **key_path** – single key or list of keys identification path for the item/items to be searched. Possible keys: integer - behaves like normal __getitem__() -> itree_item[key] TagIdx- behaves like normal __getitem__() -> itree_item[key] iTreeTagSlice - select a tag sliced group of sub-elements iTMatch - search pattern can be used too, but keep in mind it must deliver a unique result! Slice - a slice of indexes (like a special index list) string - will be parsed by the separators, special string '*' is as interpreted as any match iterable list/tuple/deque,... - run over single items
- **item_filter** – filters the item content regarding NORMAL, TEMPORARY and LINKED flag or a given filtering method
- **default_return** – object will be return in case of no match (default = None)
- **str_path_separator** – separator character in case of strings for the search levels (default: "/")
- **str_index_separator** – separator character for given tag indexes (default: "#")

Returns
iTree single item

2.5 iTree other structure related commands

itertree.iTree.**__setitem__**()

put the item in the iTree for (re)setting a child

HINT: A iTree child can only be child of one iTree (one parent only) HINT2: Linked items cannot be changed change the linked item and reload the tree!

Parameters

- **key** – single identifier for the item can be integer index or TagIdx
- **value** – iTree object that should be child of called iTree

Returns
value

itertree.iTree.**__delitem__**()

delete an item in the tree

Parameters
key – key targeting the item to be deleted single target: iTree object (remove), index, TagIdx or tuple (tag, index) (not recommended) multi target: TagIdx_s or an iterable (like list) of these keys or a slice

Returns
deleted item

itertree.iTree.**clear**()

deletes all children and data! All flags stay unchanged!

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 copy of all data items too. In copy we just copy the iTData object not the items itself, they stay as pointers to the original objects.

The function is used internally in extend operations too. And we can see (profiler) that improvements in this method might have big impact.

Returns
copied iTree object

itertree.iTree.**reverse**()

reverse the order of all children in the iTree object

itertree.iTree.**rotate**()

rotate children of the iTree object n times (rotate 1 times means move last element to first position)

Parameters
n –

2.6 iTree compare items

itertree.iTree.**__eq__**()

A iTree object is always unique we test therefore just for matching object IDs This is needed for quick index findings! ..node:: To check if properties content is equal use equal() instead :param other: iTree object to compare with :return:

itertree.iTree.**equal**()

compares if the data content of another item matches with this item

Parameters

- **other** – other iTree
- **check_parent** – check if item has same parent object too? (Default False)
- **check_coupled** – check the couple object too? (Default False)

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

Because the __eq__() method (== operator) is internally used for same item object findings. We compare here based on the python object id. Therefore for the comparison of two non possibly not identical objects the equal() method should be used.

itertree.iTree.**__contains__**()

checks if an iTree object is part of the iTree :param item: iTree object we searching for :return:

itertree.iTree.**__hash__**()

The hash operation is available but not a quick operation! ..node::: We do here not consider, parent and coupled item :return: integer hash

itertree.iTree.**__len__**()

Return len(self).

Based on the *iTree* length the comparison operators <; <=; >; >= are available too.

itertree.iTree.**count**()

count the number of children that match to the given filter :: note: The operation is not very quick on huge iTrees and complicate filters!

Parameters
item_filter –

Returns
integer number of children matching to the filter

2.7 iTree properties

As we will see later on some properties of the *iTree* object can be modified by the related methods. Warning:: The user should NEVER modify any of the given properties directly. Especially the not discussed private properties (marked with the beginning underline). Direct modifications will normally lead into inconsistencies of the *iTree* object!

The *iTree* object contains the following general properties:

itertree.iTree.**root**()
property delivers the root item of the tree

Returns
iTree root item

itertree.iTree.**is_root**()
is this item a root item (has no parent)

Returns
True/False

itertree.iTree.**parent**()
property contains the parent item

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

itertree.iTree.**pre_item**()
delivers the pre item (predecessor) of this object

Returns
iTree predecessor or None (no match)

itertree.iTree.**post_item**()
delivers the post item (successor)

Returns
iTree successor or None (no match)

itertree.iTree.**depth_up**()
delivers the distance (number of levels) to the root element of the tree

Returns
integer

itertree.iTree.**max_depth_down**()
delivers the max_depth in the direction of the children

Returns
integer maximal children depth

itertree.iTree.**is_temporary**()
In contrast to iTreeTemporary class this is False

Returns
False

itertree.iTree.**is_read_only**()

In contrast to iTreeReadOnly class this is False

Returns
False

itertree.iTree.**is_linked**()

In contrast to iTreeLinked class this is False

Returns
False

Item identification properties:

itertree.iTree.**idx**()

Index of this object in the iTree

Returns
integer index

itertree.iTree.**tag_idx**()

Get the TagIdx object related to this object (contains the tag and the index of the object in the tag-family)

Returns
TagIdx

itertree.iTree.**idx_path**()

delivers the a list of indexes from the root to this item

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

itertree.iTree.**tag_idx_path**()

delivers the a list of TagIdx objects from the root to this item

Returns
list of TagIdx (here we do not deliver an iterator)

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

(continues on next page)

(continued from previous page)

```
0
>>> root[0][0].tag_idx
TagIdx('subchild', 0)
>>> root[0][0].idx_path
[0, 0]
>>> root[0][0].tag_idx_path
[TagIdx('child', 0), TagIdx('subchild', 0)]
>>> root[1].tag_idx
TagIdx((1, 2), 0)
>>> root[-1].tag_idx
TagIdx((1, 2), 1)
```

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 in the TagIdx object.

Beside those structural properties the *iTree* objects contains some more properties that might be modified by the related methods.

itertree.iTree.**coupled_object**()

The iTree object can be couple 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 element (tree-list item) in a GUI)

Returns
pointer to coupled object

itertree.iTree.**set_coupled_object**()

User can couple this object with others with the help of this attribute .. note:: E.g. this might be an object in a GUI that are related to this item

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

Different than the data the coupled_obj is just a pointer to another python object. E.g. by this you might couple the *iTree* to a graphical user interface object e.g. an item in a hypertreelist or it can be used to couple the *iTree* object to an item in a mapping dictionary. The property couple_obj is not managed by the *iTree* object it's just a place to store a pointer. In file exports or string exports this information will not be considered.

2.8 iTree data related methods

itertree.iTree.**data**()

delivers the data-attribute object of the item

Returns
data object of the item

This is the data property. The property contains the iData objects which behaves in general like a dict. But there are two excepetions that must be considered: * The (__NOKEY__) key is an implizit key that will be used in case the user gives only one value (no_key) to the d_set() method. Then the given parameter will be stored in the (__NOKEY__) item of the dict. * In case a dict item contains a iDataModel object the given value in *iTree*.d_set() will be checked against the data model.

To manipulate data you can use the functions of the *iTree*.data object or can use the quick access functions in *iTree* object (methods related to data access have all the prefix *d_*):

itertree.iTree.**d_get**(*self*, *key=('__iTree_NOKEY__',)*, *return_type=0*)

get function for a data attribute

In case the standard iTData object is used we have:

Parameters
key – key under which the data is stored, in case no key is given the "__NOKEY__" item will be returned

Returns
data attribute object

itertree.Data.iTData.**__getitem__**()

get a specific data item by key

Except
Will raise KeyError in case given key is unknown

Parameters

- **key** – key of the data item (if not given __NOKEY__ is used!
- **_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

  ..note :: The parameter is only used by the helper method *getitem()*
  and cannot be used by standard item access

Returns
requested value

itertree.iTree.**d_set**(*self*, **args*, ***kwargs*)

set function for a data-attribute

In case the standard iTData object is used we have:

Parameters

- **key** – give key under which the data will be stored, in case data is None the first key parameter is taken as data object and it is stored in the "__NOKEY__" item
- **value** – data value the object that should be stored in the data structure of this iTree

Returns
None

itertree.Data.iTData.**__setitem__**()

setter for the iTreeData object HINT: If no value is given the key item will be interpreted as value

and it will be stored as __NOKEY__-object.

Parameters

- **key** – key under which the given object is stored
- **value** – object that should be stored

Returns
None

itertree.iTree.**d_del**(*self*, **args*, ***kwargs*)

data related del (will delete the given key)

Returns
deleted value

itertree.Data.iTData.**__delitem__**()

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** – Internal parameter cannot be reached by standard access * 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

Returns
deleted value

itertree.iTree.**d_pop**(*self*, **args*, ***kwargs*)

data related pop (will delete the given key from data-attribute)

Returns
deleted value

itertree.Data.iTData.**pop**()

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

itertree.iTree.**d_update**(*self*, **args*, ***kwargs*)

update function data-attribute

In case the standard iTData object is used we have:

Parameters

- **key** – give key under which the data will be stored, in case data is None the first key parameter is taken as data object and it is stored in the "__NOKEY__" item
- **value** – data value the object that should be stored in the data structure of this iTree

Returns
None

itertree.Data.iTData.**update**()

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

Do not replace the *iTree*.data object with another object (iTree.data is just a property which is linking into the internal structure). You will destroy a part of the functionality, use *iTree*.data.clear() and *iTree*.data.update() instead.

2.9 iTree iterators and queries

The standard iterator for iTrees delivers all children of the opbject. Beside this we have some special iterators that contain specific filter possibilities.

itertree.iTree.**__iter__**()

standard iterator over all items in the iTree :param item_filter: ALL = default :return:

itertree.iTree.**iter_children**()

main iterator in children level

Parameters
item_filter – the items can be filtered by giving a filter constants or giving a filter method or iTFilter object

Returns
iterator

itertree.iTree.**iter_all**()

main iterator for whole tree runs in top-> down order e.g.

```
iTree('child')
 └──iTree('sub0')
     └──iTree('sub0_0')
     └──iTree('sub0_1')
     └──iTree('sub0_2')
     └──iTree('sub0_3')
 └──iTree('sub1')
     └──iTree('sub1_0')
```

will be iterated like:

```
iTree('child')
iTree('sub0')
iTree('sub0_0')
iTree('sub0_1')
iTree('sub0_2')
iTree('sub0_3')
iTree('sub1')
iTree('sub1_0')
```

Parameters

- **item_filter** – filter for filter the items you can give a filter constant or a method for filtering (should return True/False)
- **filter_or** –
  - True - we combine the filtering with or this means even if we have no match in the higher levels of the tree we will go deeper to find matches
  - False - filters are combined with and which means children will only be parsed in case the parent matches also to the filter condition

Returns

iterator

itertree.iTree.**iter_all_bottom_up**()

main iterator for whole tree runs in down-> top order (We start at the children and afterwards the parents: e.g.:

```
iTree('child')
 └──iTree('sub0')
     └──iTree('sub0_0')
     └──iTree('sub0_1')
     └──iTree('sub0_2')
     └──iTree('sub0_3')
 └──iTree('sub1')
     └──iTree('sub1_0')
```

Will be iterated:

```
iTree('sub0_0')
iTree('sub0_1')
iTree('sub0_2')
iTree('sub0_3')
iTree('sub0')
```

(continues on next page)

(continued from previous page)

```
iTree('sub1_0')
iTree('sub1')
iTree('child')
```

Parameters

- **item_filter** – filter method for filtering (should return True/False when fet with an item) or iTFilter object
- **filter_or** –
  - True - we combine the filtering with or this means even if we have no match in the higher levels of the tree we will go deeper to find matches
  - False - filters are combined with and which means children will only be parsed in case the parent matches also to the filter condition

Returns

iterator

itertree.iTree.**iter_tag_idxs**()

iter over all children and deliver the children TagIdx

Parameters

item_filter – the items can be filtered by giving a filter constants or giving a filter method or iTFilter object

Returns

iterator over the TagIdx of the children

itertree.iTree.**index**()

The index method allows to search for the index of the item in a parent object This is especially useful if you must use a item_filter. The delivered index is delivered relative to the given item filter!

For the item index of the item in the unfiltered tree (ALL) it's recommended to use the idx property instead: *(parent.index(item,ALL) == item.idx)*

Parameters

- **item** – item index should be delivered for
- **item_filter** – filter integer; method can not handle filter methods yet!

Returns

index integer of the item relative to the given filter

Beside the classical iterators we have the more query related find methods:

itertree.iTree.**find_all**()

The find all function works on all levels of the tree. The key_path given (e.g. a list of indexes) addresses the items into the depth first item first level, second item second level,....

The method returns always an iterator or in case of no match an empty list! If you target to unique objects you will get anyway an iterator containing this unique element.

Warning: It's possible to create invalid recursions when constructing the key_path. In these cases the recursion depth exceeded exception will be raised by the interpreter

In case the target in the upper keys is not unique, all matches will be delivered! e.g. The operation *my_tree.find_all(['child','sub_child'])* takes first all items in the "child" family:

TagIdx('child',0),TagIdx('child',1),...TagIdx('child',n) in an iterator and in the next step the function will go one level deeper and will cumulate all the 'sub_child' families in these items as the result:

This means we have something like this:

my_tree[TagIdx('child',0)][TagIdx('sub_child',0)],my_tree[TagIdx('child',0)][TagIdx('sub_child',1)],...,

my_tree[TagIdx('child',1)][TagIdx('sub_child',0)],my_tree[TagIdx('child',0)][TagIdx('sub_child',1)],...,

...

and in case of no match in the keys items are skipped.

Note: It's not at all the same as: my_tree['child']['sub_child'] -> this operation will raise an exception!

Note: When addressing a single item it's quicker (~10x faster depending on tree depth) to use the *get_deep()* method instead of the *find_all()* method.

The key_path parameter is very flexible in case of the objects you put in. We have several possibilities:

0. Special keys: We have the following special keys that might be used in the key_path:
    - "/" default path separator (might be changed by str_path_separator parameter) If this is the first key the *find_all()* search will be started in the root element not in the element the method is called.

      Note: Be careful with "//" or "/" placed not in the beginning of the path this will rollback the *find_all()* to the root which means anything in the key_path before this key will be ignored.

    - "*"-wildcard will iterate over all children of the item
    - "**"-wildcard will iterate over all items of the item. The item itself is the first element of the iterator delivered

      ---

      Note: *find_all('**')* creates an different iterator then *iter_all() list(my_tree.find_all('**')) = [my_tree] + list(my_tree.iter_all())*

    Warning: It's always recommended to avoid the usage of string tags containing functional characters like "**","*","/","#","?". E.g. In case the iTree contains a family with the tag "/" or "*" or "**" the related family will be delivered. The special functionality is blocked in this moment (for "/" you might use the str_path_separator parameter to keep the functionality). Also filtering via iTMatch objects is limited in this case.

1. Give normal keys like in *__getitem__()* method: normal keys can be:
    - index integers
    - tag strings
    - TagIdx,TagIdxStr,TagIdxBytes

- TagMultiIdx,slices
- for index lists you must give[[1,2,3,4]] because first level will be interpreted as
- a list targeting into the depth of the tree

e.g. by index

- *my_tree.find_all(1)* is same as *my_tree[1]*
- *my_tree.find_all('child')* is same as *my_tree['child']*
- *my_tree.find_all(TagIdx('child',1))`is same as `my_tree[TagIdx('child',1)]*

...

2. Give a list of normal keys:

   e.g. by index

   - `my_tree.find_all([1,2])`is same as my_tree[1][2]
   - ***my_tree.find_all(['child','sub_child'])*** **delivers an iterator over all "sub_child" families found**
     in all "child" families
   - *my_tree.find_all([TagIdx('child',1),TagIdx('sub_child',1)])`is same as `my_tree[TagIdx('child',1)][TagIdx('sub_child',1)]*

   ...

3. Give *iTMatch()* object or list of *iTMatch()* objects:

   An iterator of all matching tags will be created the matches will be combined with the and operation. You can also use an item_filter containing the Filter.iTFilterItemTagMatch to have the same functionality. In case a list is given the find_all() function is again going one level deeper for each element in the list.

   Parameters

   - **key_path** – iterable/iterator that addresses items in the tree (see above explanations and examples)
   - **item_filter** – item_filter method
   - **str_path_separator** – In case of string tags the user can give also strings that are internally casted into a list by using the str_path_separator (default="/") e.g.: "/child_tag/sub_child_tag" -> ["child_tag","sub_child_tag"]
   - **str_index_separator** – In case of string tags the user can give TgaIdx also by string definition this is the separator used to separate the index number from the tag (default="#") e.g. "child_tag#89" -> TagIdx("child_tag",89)

   Returns
   iterator over the matches or in case of no match found an empty list -> []

For filter creation we have some helper classes (itree_filter.py)

itertree.Filter.**iTFilterTrue**()

This filter might be useless but it delivers True for all items (or False if inverted).

Parameters

- **pre_item_filter** – Additional filter to combine with this filter (will always be calculated before this filter)

- **invert** – True - invert the result of the filter (not) False (default) - result of filter is kept unchanged
- **use_and** – True (default) - combine this filter with item_filter via and operator False - use or operator instead of and

itertree.Filter.**iTFilterItemType**()

Filter for iTree types (we have iTree,ITreeReadOnly,iTreeTemporary,iTreeLink types)

Parameters

- **item_type** – target type class
- **pre_item_filter** – Additional filter to combine with this filter (will always be calculated before this filter)
- **invert** – True - invert the result of the filter (not) False (default) - result of filter is kept unchanged
- **use_and** – True (default) - combine this filter with item_filter via and operator False - use or operator instead of and

itertree.Filter.**iTFilterItemTagMatch**()

Filter using the iTMatch object (have a look on th iTMatch for more details). In generalyou can use wild cards, etc. to find matching item tags

Parameters

- **match** – iTMatch object that checks the item for a match
- **pre_item_filter** – Additional filter to combine with this filter (will always be calculated before this filter)
- **invert** – True - invert the result of the filter (not) False (default) - result of filter is kept unchanged
- **use_and** – True (default) - combine this filter with item_filter via and operator False - use or operator instead of and

itertree.Filter.**iTFilterData**()

This is the main data filter that allows a large number of different filtering based on iTree.data content. It's the recommended filter for this proposes because different than the simpler data filters in this module we can filter based on combinations (key/value) related to the iTree.data items

Parameters

- **data_key** – Checks if the given data key exists in item.data in case iTMatch is given matching keys will be considered None - all keys will be considered
- **data_value** – Checks if the given data value exists in item.data in case iTMatch is given matching values will be considered, if iTInterval is given numerical values matching to interval will be considered. None - all values will be considered
- **pre_item_filter** – Additional filter to combine with this filter (will always be calculated before this filter)
- **invert** – True - invert the result of the filter (not) False (default) - result of filter is kept unchanged
- **use_and** – True (default) - combine this filter with item_filter via and operator False - use or operator instead of and

itertree.Filter.**iTFilterDataKey**()

Filters in all items for the data key given. Delivers all items that have the given key in there data

Parameters

- **data_key** – Checks if the given data key exists in item.data
- **pre_item_filter** – Additional filter to combine with this filter (will always be calculated before this filter)
- **invert** – True - invert the result of the filter (not) False (default) - result of filter is kept unchanged
- **use_and** – True (default) - combine this filter with item_filter via and operator False - use or operator instead of and

itertree.Filter.**iTFilterDataKeyMatch**()

Filters in all items for the data key which matches to the given pattern (fnmatch search is used) you can use wildcards here. This filter works only on string or byte keys in the item.data (not on other objects)

Parameters

- **match_pattern** – string/bytes that contains a match pattern
- **pre_item_filter** – Additional filter to combine with this filter (will always be calculated before this filter)
- **invert** – True - invert the result of the filter (not) False (default) - result of filter is kept unchanged
- **use_and** – True (default) - combine this filter with item_filter via and operator False - use or operator instead of and

itertree.Filter.**iTFilterDataValueMatch**()

Filters in all items for containing a matching data value to given pattern. (Works only on string and byte values

Parameters

- **match_pattern** – pattern fnmatch will search for (you can use wildcards here)
- **pre_item_filter** – Additional filter to combine with this filter (will always be calculated before this filter)
- **invert** – True - invert the result of the filter (not) False (default) - result of filter is kept unchanged
- **use_and** – True (default) - combine this filter with item_filter via and operator False - use or operator instead of and

Depending on the data stored in the *iTree*.data object the user might create own filters. In general just a method must be created that takes the item as an argument and that delivers True in case of a match and False in case of no match. We have also a base class (super-class) of the given filters available which might be used for own filters too.

itertree.Filter.**iTFilterBase**()

Base/Super class for all itertree filter classes might be used for user defined filters too

Parameters

- **filter_method** – method that is fet with an iTree item and that delivers True/False
- **pre_item_filter** – Additional filter to combine with this filter (will always be calculated before this filter)
- **invert** – True - invert the result of the filter (not) False (default) - result of filter is kept unchanged

- **use_and** – True (default) - combine this filter with item_filter via and operator False - use or operator instead of and

The fitering in *iTree* is very effective and quick. As an example one might execute the example script itree_usage_example1.py where the itertree.Filter.iTFilterData object is used.

2.10 iTree formatted output

itertree.iTree.**__repr__**()

create representation string from which the object can be reconstructed via eval (might not work in case of data that do not have a working repr method) :return: representation string

itertree.iTree.**renders**()

render the iTree into a string

Parameters
item_filter – the items can be filtered by giving a filter constants or giving a filter method or iTFilter object

Returns
Tree representation as string

itertree.iTree.**render**()

print the rendered the iTree string to the terminal

Parameters
item_filter – the items can be filtered by giving a filter constants or giving a filter method or iTFilter object

2.11 iTree file storage

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

Returns
True if file is stored successful

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

Returns
serialized string (JSON in case of default serializer)

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

Returns
iTree object loaded from file

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

Returns
iTree object loaded from file

The file storage methods and the rendering methods are initialized by:

itertree.iTree.**init_serializer**()

Method sets the exchange environment that should be used. If you leave the parameters as default, the standard objects will be used.

Note: The method logic is called only one time the first time serializing is needed.

Parameters

- **force** – False (Default) - do not reload in case we have already loaded the items
- **exporter** – exporter object for file export of iTree (dump, dumps)
- **importer** – importer object in ces a file import is done (load, loads)
- **serializer** – Object serializer (especially needed for data objects!)
- **renderer** – A renderer for pretty print output of the iTree object

Returns
None

This method is implicit executed and set to the default serializing functions of itertree. The user might load his own functionalities explicit by using this method or he might overload the *iTree* class and the init_serializer() method with his own functionality (e.g. an xml export/import might be realized by this).

2.12 iTree linked sub-trees

The *iTree* objects can be merged to one main tree from different source files by using the iTreeLink class. 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.

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 elements 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_self_local()* functions and we cannot *appendleft()* or *insert()* because this would mean we have to reorder the other elements. 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 element, 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 elements. The identification is always made via the TagIdx of the item. The local storage of the tree contains iTreePlaceholder elements which will be replaced by the linked in elements during the load process. Those placeholders are needed to create the matching tag-idx combination for the real elements that should be kept after reload. In case the loaded structure is changed and and no matching item is found the iTreePlaceHolder items will remain in the *iTree*. All appended local items which are outside of the linked structure will be found at the end of the itertree.

Local items can be manipuplated 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 element after deletion. This means in this case a delete of an item will not reduce the numbers of the elements. If the local item has no corresponding linked item the number of children will decrease as usual.

The linked items must be loaded by an explizit operation. They are not loaded automatically. The links must be loaded via the *load_links()* method which can be executed at any level of the tree and it will start loading all links in the subtree (use *load_links()* on the *iTree* root to be sure to load all links). The behavior in case of load erros can be switched between Exceptions or deleting invalid elements (delete_invalid_items parameter). In case of exceptions the *iTree* is in an incomplete load state and if the exception is kept this must be handled (e.g. copy original tree before loading and replace back). The commands for loading *iTree* files can be influenced by the *load_links* parameter (to activate or deactivate the link loading) during file load.

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 TagIdx.** If we miss items during load placeholders are used to keep the TagIdx of the "real" local items. Normally those artefacts will be replaced during the load with linked items (if found) but in case of missmatches they will stay in the tree. Using wild linking in between different *iTree* elements 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 iTreeLink objects (e.g. we do allow only linking on not already linked elements (protection for circular definitions); local items cannot be linked items or temporary items).

itertree.**iTreeLink**(*tag*, *data=None*, *subtree=None*, *link_file_path=None*, *link_key_path=None*, *load_links=True*)

This class is used to define linked subtrees in a iTree object. The target source can be a subtree in another iTree related file (external links) or internal links to a subtree of the already loaded subtree.

Linking has some functional limitations so is it not allowed to link to already linked objects (we must protect iTree from circular definitions).

The iTreeLink objects supports local items which can be added additional to the linked items. Furthermore there is also a mechanism so that local items can overlay the linked items in the tree. This is done by localizing the linked items with the *make_child_local()* or *make_self_local()* method. Afterwards the item can be manipulated

as a normal iTree object. Only exception is that after deleting such a overlaying item the linked item will come back into the iTree.

itertree.iTreeLink.**load_links**()

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

Returns

- True - success
- False - load failed

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

itertree.iTreeLink.**make_self_local**()

make the current linked object a local object This is only possible if the parent parent is a normal iTree object -> 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

itertree.iTreeLink.**make_child_local**()

make the item related to the given key a local object This is only possible if the parent of self is a normal iTree object -> 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

Parameters
key – identification key for the child item that should be converted in a local item

Returns
None

itertree.iTreeLink.**iter_locals**()

iterator that iterates only over the local elements

Parameters
add_placeholders – If this flag is set the (normally ignored) placeholder items are included in the iteration

Returns
iterator over local items

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 the current tree the iTreeLinked class is used.

```
>>> #We create a small iTree:
>>> root = iTree('root')
>>> root += iTree('A')
>>> root += iTree('B')
>>> # we add "B" tag two times to get an enumerated tag family
>>> 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 -> load_links=False):
>>> linked_element=iTreeLink('internal_link',link_key_path='/B',load_links=False)
>>> root.append(linked_element)
>>> print(root.render())
iTree('root')
     └──iTree('A')
     └──iTree('B')
     └──iTree('B')
         └──iTree('Ba')
         └──iTree('Bb')
         └──iTree('Bb')
         └──iTree('Bc')
    └──iTreeLink('internal_link', link=iTreeLink(file_path=None, key_path=['/',␣
↪TagIdx(tag='B', idx=1)]))
>>> # now we load the links:
>>> root.load_links()
>>> print(root.render())
iTree('root')
     └──iTree('A')
     └──iTree('B')
     └──iTree('B')
         └──iTree('Ba')
         └──iTree('Bb')
         └──iTree('Bb')
         └──iTree('Bc')
     └──iTreeLink('internal_link', link=iTreeLink(file_path=None, key_path=['/',␣
↪TagIdx(tag='B', idx=1)]))
         └──iTreeLink('Ba')
         └──iTreeLink('Bb')
         └──iTreeLink('Bb')
         └──iTreeLink('Bc')
```

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

```
>>> B +=iTree('B_post_append')
>>> print(root.render())
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')
     └──iTreeLink('internal_link', link=iTreeLink(file_path=None, key_path=['/',␣
↪TagIdx(tag='B', idx=1)]))
         └──iTreeLink('Ba')
         └──iTreeLink('Bb')
         └──iTreeLink('Bb')
         └──iTreeLink('Bc')
>>> root.load_links(force=True) # we must force the reloading, if not forced already␣
↪loaded trees will not be updated
>>> print(root.render())
iTree('root')
     └──iTree('A')
     └──iTree('B')
     └──iTree('B')
         └──iTree('Ba')
         └──iTree('Bb')
         └──iTree('Bb')
         └──iTree('Bc')
         └──iTree('B_post_append')
     └──iTreeLink('internal_link', link=iTreeLink(file_path=None, key_path=['/',␣
↪TagIdx(tag='B', idx=1)]))
         └──iTreeLink('Ba')
         └──iTreeLink('Bb')
         └──iTreeLink('Bb')
         └──iTreeLink('Bc')
         └──iTreeLink('B_post_append')
```

The toplevel iTreeLink object allows manipulations of the subtree. We can append elements and we can change existing subitems to a local item taht covers the linked item and that can contain diffrent data and different children.

```
>>> #get the linked element
>>> il=root[TagIdx('internal_link',0)]
>>> #append an item
>>> il.append(iTree('new'))
>>> #we make second element local and append a item in the subtree
>>> local=il.make_child_local(2)
>>> local+=iTree('sublocal')
>>> print(root.render())
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')
    └──iTreeLink('internal_link', link=iTreeLink(file_path=None, key_path=['/',␣
↪TagIdx(tag='B', idx=1)]))
        └──iTreeLink('Ba')
        └──iTreeLink('Bb')
        └──iTree('Bb')
            └──iTree('sublocal')
        └──iTreeLink('Bc')
        └──iTreeLink('B_post_append')
        └──iTree('new')
```

The element '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 il[TagIdx('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')
    └──iTreeLink('internal_link', link=iTreeLink(file_path=None, key_path=['/',␣
↪TagIdx(tag='B', idx=1)]))
        └──iTreeLink('Ba')
        └──iTreeLink('Bb')
        └──iTreeLink('Bb')
        └──iTreeLink('Bc')
        └──iTreeLink('B_post_append')
        └──iTree('new')
```

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 for the class concept.

2.13 iTree helpers classes

In the itertree helper module we have some helper classes that can be used to construct specific *iTree* objects.

We have the following helper classes available:

`itertree.itree_helpers.iTInterval()`

helper class that defines an interval for range definitions in Data Models or Filters

the class contains a check if a given value is in the defined interval or not

The class might be a little bit under estimated in all the itertree functionalities but its a short but very powerful implementation of an Interval class for python.

The class contains anything you might need in case of a Interval functionality. You can given open/closed interval definitions including infinite limits. The intervals can be combined to a mathematical set via the pre_interval parameter. And the check method allows to give other limits as defined. This is especially useful for dynamically calculated limits.

The interval definition is also possible via a mathematical string like: "(1,2)" or "[10,+inf)".

If you need a more advanced implementation you might have a look on the intervals/portion python package.

Note: For equal just set upper_limit to None (upper_open, lower_open parameter will be ignored in this case)

---

itertree.itree_helpers.iTInterval.**__init__**()

helper class that defines an interval for range definitions

the class contains a check if a given value is in the defined interval or not

---

Note: For equal you give lower_limit and set upper_limit to None (lower_open,upper_open parameters will be

---

ignored in this case). The math representation in this case is "== %s"%lower_limit

---

Note: The not_in=True can be given to invert the interval check result (match is anything outside the interval) in the math representation we add in this case a "!" before the interval

---

Note: Cascade interval definitions can be created the pre_interval definition e.g. math_repr= "(([1,5]) and [9,12]) and [100,200]' valid values: 1…5,9…12,100..200

Parameters

- **lower_limit** – lower limit of the interval
- **upper_limit** – upper limit of the interval
- **lower_open** – True - open interval (x>lower_limit) False - closed interval (x>=lower_limit)
- **upper_open** – True - open interval (x<upper_limit) False - closed interval (x<=upper_limit)
- **not_in** – False - check for in interval True - check for not in interval (outside)
- **pre_interval** – Interval object to be checked before this interval
- **pre_and** – True - combine the result of pre check with and this Interval check with the and operator False - combine the result of pre check with and this Interval check with the or operator
- **str_def** – instance the object from given math_repr string (other parameters will be ignored in this case)

itertree.itree_helpers.**iTMatch**()

The match object is used to defined match to elements in the DtaTree used in iterations over the DataTree The defined iMatch object can be used for checks against iTree objects (mainly for checks against the tag and also for string matches e.g. for finding iTree.data.keys() or .values() in filters.

itertree.itree_helpers.iTMatch.__init__()

Create a match pattern for different proposes. Depending on the type we have following functions:

- int - check for an index
- TagIdx - check for a TagIdx
- str - string pattern using fnmatch
- iterable like list, tuple, ... combine the given patterns with the combine key

Parameters

- **pattern** – give pattern
- **combine_or** – True - or ; False - and; combination of matches/match patterns

itertree.itree_helpers.**TagIdx**()

TagIdx(tag, idx)

itertree.itree_helpers.**TagIdxStr**()

Define a TagIdx by a sting with an index separator (default='#')

Example: "mytag#1" will be translated in the TagIdx("mytag",1)

Note: This makes only sense and can only be used if the tag is a string (not for other objects)

Parameters

tag_idx_str – string containing the definition

itertree.itree_helpers.**TagIdxBytes**()

Define a TagIdx by bytes with an index separator (default=b'#')

Example: b"mytag#1" will be translated in the TagIdx(b"mytag",1)

Note: This makes only sense and can only be used if the tag is a byte (not for other objects)

Parameters

tag_idx_bytes – bytes containing the definition

The other classes in itree_helpers are used internally in the *iTree* object and might be less interesting for the user.

Addinionally the user might have also a look in the other itertree modules like itertree_data.py or itertree_filter.py. Especially the class iTDataModel might be a good starting point for own data model definitions as it is also shown in examples/itertree_data_model.py.

itertree.itree_data.**iTDataModel**()

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)

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:

This part of code contains the main iTree object

class `itertree.itree_main.`**iTree**(*tag*, *data=None*, *subtree=None*)

Bases: `list`

This is the main class related to iTrees.

This object is the parent of a sub-tree (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.

A iTree object can only be integrated in one tree (one parent only)!

Each iTree object contains a tag. In case your tags are stings it's recommended to use tag strings without wildcards "*","?" and without the standard separators "/" and "#". If you use these characters you might get confusing results in find, filter and match operations.

In general we allow all hashable objects to be used as a tag in the iTree objects (only search operation might be limited in this case). But we have two exceptions: We do not allow integers and TagIdx objects as tags because those objects used for direct item access.

Different than in dictionaries it is allowed to put multiple times the same tag inside the iTree. The items with the same tag are placed and ordered (enumerated) in the related tag-family. They can be reached via TagIdx objects by giving the tag, index pair (tag_idx).

Linked iTree objects will behave different. They have a read only structure (children) and they contain the children (tree) of the linked iTree. The "local" attributes like tag, data, ... can be set independent from the linked item (local properties). To change the tree structure of such an object you must manipulated the source object and reload the link.

Additionally a iTree object can contain:

- data - a iTData object to store any kind of python objects
- couple - you can couple the object to another one by giving a pointer
- is_temporary - you can mark it as temporary. Those iTree items behave like normal ones. But they will not be considered during encoding for storage, etc.

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[] (__getitem__()) call.

More complex access is available via find() and findall() methods. Have a look in the documentation related to each method.

The delivery of access related operations in the iTree objects is for unique targets an iTree object and for multi target operations an iterator over the matching items. We don't deliver something like a list.

If really needed an iterator can be easily converted into a list by list() method but this may take a long time for huge iterators. The iterator should only be used in the final step of the operation. It's recommended to have a look into itertools for better usage of the delivered iterators.

The design of the object is made to have best possible performance even that it is pure python. For more details you may run the performance tests in the test section (But you might have to install additional packages run the comparisons and to get the full picture.)

The function related to iterations iter; iter_children and find_all can be used with an item_filter. By this mechanism you can create queries regarding any property in an iTree.

To initialize the class the following parameters are available

Parameters

- **tag** – tag string or hashable object used for the iTree identification
- **data** – data dict or item to be stored in the node
- **subtree** – The subtree is a iterable structure that contains sub-items (iTree objects) that should be the children of this iTree.

> **Warning:** subtree: In case the given iTree objects have already a parent an implicit copy will be made.

init_serializer(*force=False*, *exporter=None*, *importer=None*, *serializer=None*, *renderer=None*) → None

Method sets the exchange environment that should be used. If you leave the parameters as default, the standard objects will be used.

Note: The method logic is called only one time the first time serializing is needed.

Parameters

- **force** – False (Default) - do not reload in case we have already loaded the items
- **exporter** – exporter object for file export of iTree (dump, dumps)
- **importer** – importer object in ces a file import is done (load, loads)
- **serializer** – Object serializer (especially needed for data objects!)
- **renderer** – A renderer for pretty print output of the iTree object

Returns
None

sort(**arg*, ***kwargs*)

sort operation is not supported, method exists just because super class supports it. Here a TypeError will be raised.

property data

delivers the data-attribute object of the item

Returns
data object of the item

d_set(**args*, ***kwargs*)

set function for a data-attribute

In case the standard iTData object is used we have:

Parameters

- **key** – give key under which the data will be stored, in case data is None the first key parameter is taken as data object and it is stored in the "__NOKEY__" item
- **value** – data value the object that should be stored in the data structure of this iTree

Returns
None

d_get(*key=('__iTree_NOKEY__',)*, *return_type=0*)

get function for a data attribute

In case the standard iTData object is used we have:

Parameters
key – key under which the data is stored, in case no key is given the "__NOKEY__" item will be returned

Returns
data attribute object

d_update(**args*, ***kwargs*)

update function data-attribute

In case the standard iTData object is used we have:

Parameters

- **key** – give key under which the data will be stored, in case data is None the first key parameter is taken as data object and it is stored in the "__NOKEY__" item
- **value** – data value the object that should be stored in the data structure of this iTree

Returns
None

d_check(*value*, *key=('__iTree_NOKEY__',)*)

check if the given data-item can be stored under the given key. The check make only sense in case there is a iTreeDataModel or matching object is already stored under the key

Exception
check will raise an iDataValueError or iDataTypeError exception in case the value is not matching in case given key is not found a KeyError will be raised

Parameters

- **value** – data value the object that should be checked
- **key** – give key under which contains the DataModel, in case key is not given the "__NOKEY__" item will be used

Returns
valid value

d_pop(**args*, ***kwargs*)

data related pop (will delete the given key from data-attribute)

Returns
deleted value

d_del(**args*, ***kwargs*)

data related del (will delete the given key)

Returns
deleted value

property parent

property contains the parent item

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

property is_root

is this item a root item (has no parent)

Returns
True/False

property root

property delivers the root item of the tree

Returns
iTree root item

property is_read_only

In contrast to iTreeReadOnly class this is False

Returns
False

property is_temporary

In contrast to iTreeTemporary class this is False

Returns
False

property is_placeholder

In contrast to iTreePlaceholder class this is False

Returns
False

property is_linked

In contrast to iTreeLinked class this is False

Returns
False

property link_item

in case we have "covered" a linked item this property delivers the original linked item (mainly for internal use)

Returns
None - no linked item iTreeLink object the covered item

property pre_item

delivers the pre item (predecessor) of this object

Returns
iTree predecessor or None (no match)

property post_item

delivers the post item (successor)

Returns
iTree successor or None (no match)

property depth_up

delivers the distance (number of levels) to the root element of the tree

Returns
integer

property max_depth_down

delivers the max_depth in the direction of the children

Returns
integer maximal children depth

property idx_path

delivers the a list of indexes from the root to this item

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

property tag_idx_path

delivers the a list of TagIdx objects from the root to this item

Returns
list of TagIdx (here we do not deliver an iterator)

property tag_idx

Get the TagIdx object related to this object (contains the tag and the index of the object in the tag-family)

Returns
TagIdx

property tag

This objects tag

Returns
tag object

property idx

Index of this object in the iTree

Returns
integer index

property coupled_object

The iTree object can be couple 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 element (tree-list item) in a GUI)

Returns
pointer to coupled object

set_coupled_object(*coupled_object*)

User can couple this object with others with the help of this attribute .. note:: E.g. this might be an object in a GUI that are related to this item

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

equal(*other*, *check_parent=False*, *check_coupled=False*)

compares if the data content of another item matches with this item

Parameters

- **other** – other iTree
- **check_parent** – check if item has same parent object too? (Default False)
- **check_coupled** – check the couple object too? (Default False)

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

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 copy of all data items too. In copy we just copy the iTData object not the items itself, they stay as pointers to the original objects.

The function is used internally in extend operations too. And we can see (profiler) that improvements in this method might have big impact.

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 copy of all data items too. In copy we just copy the iTData object not the items itself, they stay as pointers to the original objects.

Returns
deep copied new iTree object

`count`(*item_filter=None*)

count the number of children that match to the given filter :: note: The operation is not very quick on huge iTrees and complicate filters!

Parameters
item_filter –

Returns
integer number of children matching to the filter

`count_all`(*item_filter=None*)

count deep the number of children and sub children the element has and that match to the given filter :: note: The operation is not very quick on huge iTrees and complicate filters!

Parameters
item_filter –

Returns
integer number of children matching to the filter

`get_deep`(*key_list*)

deep key access the function is a replacement for *self[key_list[0]][key_list[1]]...[key_list[-1]]* but you can also feed with an iterator

dives into the tree *key_list=[1,0,2]* -> second element level 1 -> first element level 2 -> third element level 3 -> same as *self[1][0][2]*

Note: Each key in the key list must target to a single item only! E.g. do not use tags here they deliver always a family iterator not a single item (the method will raise an exception). Use index integers or TagIdx objects instead

---

Parameters
key_list – list or iterator of keys (indexes,TagIdx, tuple(tag,index) -> only in case no tuple tags!

Returns
iTree object the key list targets

`clear`()

deletes all children and data! All flags stay unchanged!

`insert`(*insert_key*, *item*)

Insert an item before a specific position

Parameters

- **insert_key** – position key (integer index or TagIdx)
- **item** – item that should be inserted in the tree (new child)

append(*item*)

Append the given iTree object to the tree (new last child)

Except
raise TypeError in case iTree object has already a parent

Parameters
item – iTree object to be appended

Returns
True in case append was successful

appendleft(*item*)

Append the given iTree object to the left of the the tree (new first child)

Except
raise TypeError in case iTree object has already a parent

Parameters
item – iTree object to be appended

extend(*extend_items*)

We extend the iTree with given items (multi append)

Note: In case the extend items have already a parent an implicit copy will be made. We do this because we might get an iTree-object as extend_items parameter and then the children will have automatically a parent even that the parent object might be a temporary one.

Parameters
extend_items – iterable object that contains iTree objects as items

Returns
True

extendleft(*extend_items*)

We extend the iTree with given items in the beginning (multi appendleft)

Note: In case the extend items have already a parent an implicit copy will be made. We do this because we might get an iTree-object as extend_items parameter and then the children will have automatically a parent even that the parent object might be a temporary one.

Note: The *extendleft()* operation is a lot slower then the normal extend operation

---

Parameters
extend_items – iterable object that contains iTree objects as items

pop(*key=- 1*)

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

Parameters
key – specific identification key for an item (integer index, TagIdx)

Returns
popped out item (parent will be set to None)

popleft()

pop the first item out of the tree

Returns
popped out item (parent will be set to None)

remove(*item*)

remove the given item out of the tree (delete the child)

Parameters
item – iTree object that should be removed from the tree

Returns
removed item will be returned (parent is set to None)

move(*insert_key*)

move the item in another position

Parameters
insert_key – item will be insert before this key

rename(*new_tag*)

give the item a new tag

Parameters
new_tag – new tag object string or hashable object

reverse()

reverse the order of all children in the iTree object

rotate(*n*)

rotate children of the iTree object n times (rotate 1 times means move last element to first position)

Parameters
n –

iter_all(*item_filter=None*, *filter_or=True*)

main iterator for whole tree runs in top-> down order e.g.

```
iTree('child')
 └──iTree('sub0')
     └──iTree('sub0_0')
     └──iTree('sub0_1')
     └──iTree('sub0_2')
     └──iTree('sub0_3')
 └──iTree('sub1')
     └──iTree('sub1_0')
```

will be iterated like:

```
iTree('child')
iTree('sub0')
iTree('sub0_0')
```

(continues on next page)

(continued from previous page)

```
iTree('sub0_1')
iTree('sub0_2')
iTree('sub0_3')
iTree('sub1')
iTree('sub1_0')
```

Parameters

- **item_filter** – filter for filter the items you can give a filter constant or a method for filtering (should return True/False)
- **filter_or** –
  - True - we combine the filtering with or this means even if we have no match in the higher levels of the tree we will go deeper to find matches
  - False - filters are combined with and which means children will only be parsed in case the parent matches also to the filter condition

Returns

iterator

iter_all_bottom_up(*item_filter=None*, *filter_or=True*)

main iterator for whole tree runs in down-> top order (We start at the children and afterwards the parents: e.g.:

```
iTree('child')
 └──iTree('sub0')
     └──iTree('sub0_0')
     └──iTree('sub0_1')
     └──iTree('sub0_2')
     └──iTree('sub0_3')
 └──iTree('sub1')
     └──iTree('sub1_0')
```

Will be iterated:

```
iTree('sub0_0')
iTree('sub0_1')
iTree('sub0_2')
iTree('sub0_3')
iTree('sub0')
iTree('sub1_0')
iTree('sub1')
iTree('child')
```

Parameters

- **item_filter** – filter method for filtering (should return True/False when fet with an item) or iTFilter object
- **filter_or** –
  - True - we combine the filtering with or this means even if we have no match in the higher levels of the tree we will go deeper to find matches

- False - filters are combined with and which means children will only be parsed in case the parent matches also to the filter condition

Returns
iterator

`iter_children`(*item_filter=None*)

main iterator in children level

Parameters
`item_filter` – the items can be filtered by giving a filter constants or giving a filter method or iTFilter object

Returns
iterator

`iter_tag_idxs`(*item_filter=None*)

iter over all children and deliver the children TagIdx

Parameters
`item_filter` – the items can be filtered by giving a filter constants or giving a filter method or iTFilter object

Returns
iterator over the TagIdx of the children

`iter_tag_idxs_all`(*item_filter=None*)

Delivers an iterator over all items tag_idx_paths

Parameters
`item_filter` – the items can be filtered by giving a filter constants or giving a filter method or iTFilter object

Returns
iterator over tuples of tag_idxs_paths of all items

`iter_idxs_all`(*item_filter=None*)

Delivers an iterator over all items index path tuples .. note:: This method is mainly usd for internal proposes (max_depth_down)

Parameters
`item_filter` – item_filter filter method might be used

Returns
iterator over tuples of index paths of all items

`find_all`(*key_path*, *item_filter=None*, *str_path_separator='/'*, *str_index_separator='#'*)

The find all function works on all levels of the tree. The key_path given (e.g. a list of indexes) addresses the items into the depth first item first level, second item second level,....

The method returns always an iterator or in case of no match an empty list! If you target to unique objects you will get anyway an iterator containing this unique element.

> **Warning:** It's possible to create invalid recursions when constructing the key_path. In these cases the recursion depth exceeded exception will be raised by the interpreter

In case the target in the upper keys is not unique, all matches will be delivered! e.g. The operation *my_tree.find_all(['child','sub_child'])* takes first all items in the "child" family:

TagIdx('child',0),TagIdx('child',1),... TagIdx('child',n) in an iterator and in the next step the function will go one level deeper and will cumulate all the 'sub_child' families in these items as the result:

This means we have something like this:

my_tree[TagIdx('child',0)][TagIdx('sub_child',0)],my_tree[TagIdx('child',0)][TagIdx('sub_child',1)],...,

my_tree[TagIdx('child',1)][TagIdx('sub_child',0)],my_tree[TagIdx('child',0)][TagIdx('sub_child',1)],...,

...

and in case of no match in the keys items are skipped.

Note: It's not at all the same as: my_tree['child']['sub_child'] -> this operation will raise an exception!

Note: When addressing a single item it's quicker (~10x faster depending on tree depth) to use the *get_deep()* method instead of the *find_all()* method.

The key_path parameter is very flexible in case of the objects you put in. We have several possibilities:

0. Special keys: We have the following special keys that might be used in the key_path:

   - "/" default path separator (might be changed by str_path_separator parameter) If this is the first key the *find_all()* search will be started in the root element not in the element the method is called.

     Note: Be careful with "//" or "/" placed not in the beginning of the path this will rollback the *find_all()* to the root which means anything in the key_path before this key will be ignored.

   - "*"-wildcard will iterate over all children of the item
   - "**"-wildcard will iterate over all items of the item. The item itself is the first element of the iterator delivered

     ---

     Note: *find_all('**')* creates an different iterator then *iter_all() list(my_tree.find_all('**')) = [my_tree] + list(my_tree.iter_all())*

   Warning: It's always recommended to avoid the usage of string tags containing functional characters like "**","*","/","#","?". E.g. In case the iTree contains a family with the tag "/" or "*" or "**" the related family will be delivered. The special functionality is blocked in this moment (for "/" you might use the str_path_separator parameter to keep the functionality). Also filtering via iTMatch objects is limited in this case.

1. Give normal keys like in *__getitem__()* method: normal keys can be:

   - index integers
   - tag strings
   - TagIdx,TagIdxStr,TagIdxBytes

- TagMultiIdx,slices
- for index lists you must give[[1,2,3,4]] because first level will be interpreted as
- a list targeting into the depth of the tree

e.g. by index

- *my_tree.find_all(1)* is same as *my_tree[1]*
- *my_tree.find_all('child')* is same as *my_tree['child']*
- *my_tree.find_all(TagIdx('child',1))`is same as `my_tree[TagIdx('child',1)]*

...

2. Give a list of normal keys:

   e.g. by index

   - `my_tree.find_all([1,2])`is same as my_tree[1][2]
   - ***my_tree.find_all(['child','sub_child'])*** **delivers an iterator over all "sub_child" families found**
     in all "child" families
   - *my_tree.find_all([TagIdx('child',1),TagIdx('sub_child',1)])`is same as `my_tree[TagIdx('child',1)][TagIdx('sub_child',1)]*

   ...

3. Give *iTMatch()* object or list of *iTMatch()* objects:

   An iterator of all matching tags will be created the matches will be combined with the and operation. You can also use an item_filter containing the Filter.iTFilterItemTagMatch to have the same functionality. In case a list is given the find_all() function is again going one level deeper for each element in the list.

   Parameters

   - **key_path** – iterable/iterator that addresses items in the tree (see above explanations and examples)
   - **item_filter** – item_filter method
   - **str_path_separator** – In case of string tags the user can give also strings that are internally casted into a list by using the str_path_separator (default="/") e.g.: "/child_tag/sub_child_tag" -> ["child_tag","sub_child_tag"]
   - **str_index_separator** – In case of string tags the user can give TgaIdx also by string definition this is the separator used to separate the index number from the tag (default="#") e.g. "child_tag#89" -> TagIdx("child_tag",89)

   Returns
   iterator over the matches or in case of no match found an empty list -> []

find_all2(*key_path*, *item_filter=None*, *str_path_separator='/'*, *str_index_separator='#'*, *_initial=True*)

Method is outdated use find_all instead!

The find_all function targets over multiple levels of the datatree, it returns a list or iterator of the matching items!

The key_path parameter given is normally a list. This can be a list of keys or TagIdx objects. The function will search for the first item in the first level, fo next item in the next level and so on...

Absolut and relative key_paths:

If the first item is the separator (default: '/') the find search is like an absolute path and we start at the root of the datatree. If the first item is different, the key_path is relative and we start from the actual item and search the children and sub-children.

Single string key_path: If the user searches for string type tags he can use a string with a separator (default: '/') in between the tags (These type of key_paths will be implicit translated in a list in the function). An index separator (default = '#') in between the tag and the index can also be used to identify to identify items in the tag family. If the key_path argument is already a list the single keys will not be parsed regarding the str_path_separator anymore.

HINT: Quickest find operations can be performed by giving a list containing index integers or TagIdx objects

The items can be filtered regarding specific content, for this a look into the available filer constructors: create_xxx_item_filter() might be interesting. The filter method or the filter constant can be given in the item_filter parameter

The parameters in detail:

Parameters

- **key_path** – single key or list of keys identification path for the item/items to be searched. Possible keys: integer - behaves like normal __getitem__() -> itree_item[key] TagIdx- behaves like normal __getitem__() -> itree_item[key] iTreeTagSlice - select a tag sliced group of sub-elements iTMatch - search pattern can be used too, but keep in mind it must deliver a unique result! Slice - a slice of indexes (like a special index list) string - will be parsed by the separators iterable list/tuple/deque,... -

  run over single keys if sub_key is again an iterable it will be taken as an index list (e.g. [1,2,3] - will go deeper in the tree 1. item; 2. subitem; 3. subsubitem but [[1,2,3]] - will stay in the first level and deliver 1. item; 2. item; 3. item)
- **item_filter** – filters the item content regarding NORMAL, TEMPORARY and LINKED flag or a given filtering method
- **str_path_separator** – separator character in case of strings for the search levels (default: "/")
- **str_index_separator** – separator character for given tag indexes (default: "#")
- **_initial** – Internal flag that should protect against cyclic constructs

Returns

list or iterator of matching iTrees; in case of no match and empty list is returned

find(*key_path*, *item_filter=None*, *default_return=None*, *str_path_separator='/'*, *str_index_separator='#'*)

The find function targets over multiple levels of the iTree, it returns single items only! This means in case the key_path targets to multiple items the default_return will be given. If the key_path targets to a family with only one item inside or the item_filter extracts only one item in a family the item will be given back as result. For multiple result utilize the *find_all()* method (which is slower).

Note: The method will deliver a default_return when ever in the whole key_path a match is not unique. This means iteration is stopped here and even that a deeper iteration with the defined filtering might deliver at least a unique result. To ensure to find this deeper results you must utilize the slower *find_all()* method.

The key_path parameter given is normally a list. This can be a list of keys or TagIdx objects. The function will search for the first item in the first level, fo next item in the next level and so on...

Absolut and relative key_paths:

If the first item is the separator (default: '/') the find search is like an absolute path and we start at the root of the iTree. For compatibility reasons with find_all we accept a leading "./" (or to be exact: ".%s"#str_path_separator) as absolute path indicator. If the first item is different, the key_path is relative and we start from the actual item and search the children and sub-children.

Single string key_path: If the user searches for string type tags he can use a string with a separator (default: '/') in between the tags (Those key_paths will be implicit translated in a list). An index separator (default = '#') in between the tag and the index can also be used in this case. If the argument is already a list the single keys will not be parsed regarding the str_path_separator.

Note: If iTree contains tags with characters that used for separators or the all match '*' character the find() result might contain that tagged item instead of the expected separated or wildcard match.

Note: Quickest find operations can be performed by giving a list containing index integers or TagIdx objects

---

The parameters in detail:

Parameters

- **key_path** – single key or list of keys identification path for the item/items to be searched. Possible keys: integer - behaves like normal __getitem__() -> itree_item[key] TagIdx- behaves like normal __getitem__() -> itree_item[key] iTreeTagSlice - select a tag sliced group of sub-elements iTMatch - search pattern can be used too, but keep in mind it must deliver a unique result! Slice - a slice of indexes (like a special index list) string - will be parsed by the separators, special string '*' is as interpreted as any match iterable list/tuple/deque,... -

  run over single items

- **item_filter** – filters the item content regarding NORMAL, TEMPORARY and LINKED flag or a given filtering method
- **default_return** – object will be return in case of no match (default = None)
- **str_path_separator** – separator character in case of strings for the search levels (default: "/")
- **str_index_separator** – separator character for given tag indexes (default: "#")

Returns

iTree single item

index(*item*, *item_filter=None*)

The index method allows to search for the index of the item in a parent object This is especially useful if you must use a item_filter. The delivered index is delivered relative to the given item filter!

For the item index of the item in the unfiltered tree (ALL) it's recommended to use the idx property instead: *(parent.index(item,ALL) == item.idx)*

Parameters

- **item** – item index should be delivered for
- **item_filter** – filter integer; method can not handle filter methods yet!

Returns

index integer of the item relative to the given filter

load_links(*force=False*, *delete_invalid_items=False*)

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

Parameters

- **force** – True - linked items will be reloaded even that they are already loaded
- **delete_invalid_items** – In case a iTreeLink refers to an invalid item (internal exception) the related iTreeLink object will be deleted from teh tree

loads(*data_str*, *check_hash=True*, *load_links=True*)

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

Returns
iTree object loaded from file

load(*file_path*, *check_hash=True*, *load_links=True*)

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

Returns
iTree object loaded from file

dumps(*calc_hash=True*)

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

Returns
serialized string (JSON in case of default serializer)

dump(*target_path*, *pack=True*, *calc_hash=True*, *overwrite=False*)

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

Returns
True if file is stored successful

renders(*item_filter=None*)

render the iTree into a string

Parameters
item_filter – the items can be filtered by giving a filter constants or giving a filter method or iTFilter object

Returns
Tree representation as string

render(*item_filter=None*)

print the rendered the iTree string to the terminal

Parameters
item_filter – the items can be filtered by giving a filter constants or giving a filter method or iTFilter object

class itertree.itree_main.**iTreeReadOnly**(*tag*, *data=None*, *subtree=None*, *freeze_struct_only=False*)

Bases: *iTree*

This iTree object is read only the initial parameters given cannot be changed the object remains static in the tree and can only be changed when deleted and replaced

insert(**args*, ***kwargs*)

Except
PermissionError not possible on iTreeReadOnly objects

append(**args*, ***kwargs*)

Except
PermissionError not possible on iTreeReadOnly objects

appendleft(**args*, ***kwargs*)

Except
PermissionError not possible on iTreeReadOnly objects

extend(**args*, ***kwargs*)

Except
PermissionError not possible on iTreeReadOnly objects

extendleft(**args*, ***kwargs*)

Except
PermissionError not possible on iTreeReadOnly objects

rotate(**args*, ***kwargs*)

Except
PermissionError not possible on iTreeReadOnly objects

reverse(**args*, ***kwargs*)

Except
PermissionError not possible on iTreeReadOnly objects

pop(**args*, ***kwargs*)

Except
PermissionError not possible on iTreeReadOnly objects

popleft(**args*, ***kwargs*)

Except
PermissionError not possible on iTreeReadOnly objects

remove(**args*, ***kwargs*)

Except
PermissionError not possible on iTreeReadOnly objects

clear()

Except
PermissionError not possible on iTreeReadOnly objects

class itertree.itree_main.**iTreeTemporary**(*tag*, *data=None*, *subtree=None*)

Bases: *iTree*

This is a temporary item that will not be considered if the iTree is saved into a file.

class itertree.itree_main.**iTreeLink**(*tag*, *data=None*, *subtree=None*, *link_file_path=None*, *link_key_path=None*, *load_links=True*)

Bases: *iTree*

This class is used to define linked subtrees in a iTree object. The target source can be a subtree in another iTree related file (external links) or internal links to a subtree of the already loaded subtree.

Linking has some functional limitations so is it not allowed to link to already linked objects (we must protect iTree from circular definitions).

The iTreeLink objects supports local items which can be added additional to the linked items. Furthermore there is also a mechanism so that local items can overlay the linked items in the tree. This is done by localizing the linked items with the *make_child_local()* or *make_self_local()* method. Afterwards the item can be manipulated as a normal iTree object. Only exception is that after deleting such a overlaying item the linked item will come back into the iTree.

rotate(**args*, ***kwargs*)

Except
PermissionError not possible on iTreeReadOnly objects

reverse(**args*, ***kwargs*)

Except
PermissionError not possible on iTreeReadOnly objects

append(*item*)

append of items is allowed (items are appended as locals :param item: item to be appended :return:

extend(*items*)

extend of items is allowed, items are appended as locals :param items: items to be appended (iterator) :return: None

extendleft(*item*)

Except
PermissionError not possible on iTreeReadOnly objects

`appendleft`(*item*)

Except
PermissionError not possible on iTreeReadOnly objects

`insert`(*insert_key*, *item*)

Except
PermissionError not possible on iTreeReadOnly objects

`pop`(*key*)

pop the object out of the tree (only possible on local objects)

Except
In case a linked item is selected an PermissionError is raised

Parameters
key – identification key for the child that should be popped out

Returns
popped out item (parent set to None)

`popleft`()

pop the first child out of the tree (only possible on local object)

Except
In case a linked item is selected an PermissionError is raised

Returns
popped first item (parent set to None)

`remove`(*item*)

remove the given child item out of the tree (only possible on local object)

Except
In case a linked item is selected an PermissionError is raised

Parameters
item – item to be removed from the iTree

Returns
removed item (parent set to None)

`rename`(*item_tag*)

Except
PermissionError not possible on iTreeReadOnly objects

`property is_link_root`

Is this item the highest level linked element?

Returns
True/False

`property link_root`

delivers the highest level element that is linked in case item is not linked it delivers it self

Returns
highest level linked item found in the parents

property is_link_loaded

For linked iTree objects we deliver here the state of loading the links

Returns
True/False

make_self_local()

make the current linked object a local object This is only possible if the parent parent is a normal iTree object -> 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

make_child_local(*key*)

make the item related to the given key a local object This is only possible if the parent of self is a normal iTree object -> 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

Parameters
key – identification key for the child item that should be converted in a local item

Returns
None

iter_locals(*add_placeholders=False*)

iterator that iterates only over the local elements

Parameters
add_placeholders – If this flag is set the (normally ignored) placeholder items are included in the iteration

Returns
iterator over local items

get_last_local_idx(*tag*)

helper function which searches for local items in the tag family and delivers the last index of a local item found in the family. If no local item is found it delivers None.

iTreePlaceHolder items ignored in this operation!

Parameters
tag – tag to identify the family to be searched in

Returns
last local item idx in tag family or None (no local item found)

load_links(*force=False*, *delete_invalid_items=False*, *_items=None*)

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

Returns

- True - success

- False - load failed

clear(*local_only=False*)

We clear the object

Parameters

local_only –

- True - clear only the local items
- **False - clear whole object (The object is reset to the no links loaded state and locals** are deleted)

Returns

equal(*other*, *check_parent=False*, *check_coupled=False*, *check_link=False*)

compares if the data content of another item matches with this item

Parameters

- **other** – other iTree
- **check_parent** – check the parent object too? (Default False)
- **check_coupled** – check the couple object too? (Default False)
- **check_link** – check the internal link variable too? (Default False)

Returns

boolean match result (True match/False no match)

class itertree.itree_main.**iTreePlaceHolder**(*tag*)

Bases: *iTreeReadOnly*

place holder item that helps to keep items name in the overloading mechanism

3.4 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:

This part of code contains the helper functions related to the iTree data attribute

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=('__iTree_NOVALUE__',)*)

Bases: 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*, *_it_data_model_identifier=None*)

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
- **_it_data_model_identifier** – internal parameter used for identification of the set method in special cases, no functional impact

property value

the stored value :return: object stored in value

clear(*_it_data_model_identifier=None*)

clears (deletes) the current value content and sets the state to "empty"

Parameters
_it_data_model_identifier – internal parameter used for identification of the set method in special cases, no functional impact

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

class itertree.itree_data.**iTDataModelAny**(*value=('__iTree_NOVALUE__',)*)

Bases: *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

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>>, 2: <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=('__iTree_NOKEY__',)*, *default=('__iTree_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=('__iTree_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=('__iTree_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

`class itertree.itree_data.iTDataReadOnly(seq=None, **kwargs)`

Bases: *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

3.5 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:

This part of code contains the helper functions related to the iTree data attribute

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=('__iTree_NOVALUE__',)*)

Bases: 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*, *_it_data_model_identifier=None*)

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
- **_it_data_model_identifier** – internal parameter used for identification of the set method in special cases, no functional impact

property value

the stored value :return: object stored in value

clear(*_it_data_model_identifier=None*)

clears (deletes) the current value content and sets the state to "empty"

Parameters
_it_data_model_identifier – internal parameter used for identification of the set method in special cases, no functional impact

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

class itertree.itree_data.**iTDataModelAny**(*value=('__iTree_NOVALUE__',)*)

Bases: *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

`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>>, 2: <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=('__iTree_NOKEY__',)*, *default=('__iTree_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=('__iTree_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=('__iTree_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

`class itertree.itree_data.iTDataReadOnly(seq=None, **kwargs)`

Bases: *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

3.6 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:

This part of code contains the standard iTree serializers (JSON and rendering)

class itertree.itree_serialize.**iTStdObjSerializer**

Bases: object

This class converts objects to raw objects (that can be converted by standard JSON serializer) and back conversion is also included

TREE = 'iT'

DATA = 'DT'

LINK = 'LK'

TAG = 'TG'

IDX = 'IDX'

DATA_MODELL = 'DM'

DTYPE = 'TP'

DATA_CONTAINER = 'DC'

ITREE_ITEMS_DECODE = {'iT': <class 'itertree.itree_main.iTree'>, 'iTI': <class 'itertree.itree_helpers.TagIdx'>, 'iTPH': <class 'itertree.itree_main.iTreePlaceHolder'>, 'iTRO': <class 'itertree.itree_main.iTreeReadOnly'>, 'iTl': <class 'itertree.itree_main.iTreeLink'>}

ITREE_ITEMS_ENCODE = {<class 'itertree.itree_main.iTree'>: 'iT', <class 'itertree.itree_main.iTreeReadOnly'>: 'iTRO', <class 'itertree.itree_main.iTreeLink'>: 'iTl', <class 'itertree.itree_main.iTreePlaceHolder'>: 'iTPH', <class 'itertree.itree_helpers.TagIdx'>: 'iTI'}

OTHER_ITEMS_DECODE = {'D': <class 'itertree.itree_data.iTData'>, 'DR': <class 'itertree.itree_data.iTDataReadOnly'>, 'OD': <class 'collections.OrderedDict'>, 'd': <class 'dict'>}

OTHER_ITEMS_ENCODE = {<class 'dict'>: 'd', <class 'itertree.itree_data.iTData'>: 'D', <class 'itertree.itree_data.iTDataReadOnly'>: 'DR', <class 'collections.OrderedDict'>: 'OD'}

encode(*o*)

encode the given object to a list or dict (unordered objects) :param o: object :return: list

decode(*raw_o*)

decode the given raw_object back to the original object :param raw_o: raw_object (dict or list) :param load_links: load the links of the linked iTree objects :return: constructed object

class itertree.itree_serialize.**iTStdJSONSerializer**(*obj_serializer=None*)

Bases: object

This is the standard serializer for DataTree which translates the structure into the JSON format. Users might implement there own serializers using the interface methods defined in this serializer

dumps2(*o*, *add_header=True*, *calc_hash=True*)

new dump not yet working still in development! should be iterative and only one iteration over all items should be done (not two like in the current solution) :param o: :param add_header: :param calc_hash: :return:

dumps(*o*, *add_header=True*, *calc_hash=True*)

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

Returns

string containing the serialized data

dump(*o*, *file_path*, *pack=True*, *calc_hash=True*, *overwrite=False*)

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

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

class itertree.itree_serialize.**iTStdRenderer**

Bases: object

Standard renderer fr the iTree object for creating a very simple pretty print output

render2(*itree_object*, *item_filter=None*, *_level=0*)

prints a pretty output of the iTree object

Parameters

- **itree_object** – iTree object to be converted
- **item_filter** – item filter method or filter-constant to filter specific items out
- **_level** – internal parameter for recursive calls (do not use)

Returns

string containing the pretty print aoutput

renders2(*itree_object*, *item_filter=None*, *_level=0*)

creates a pretty print string from iTree object This is the recursive version which might be a bit quicker

Parameters

- **itree_object** – iTree object to be converted
- **item_filter** – item filter method or filter-constant to filter specific items out
- **_level** – internal parameter for recursive calls (do not use)

Returns

string containing the pretty print aoutput

render(*itree_object*, *item_filter=None*)

creates a pretty print from iTree object and prints it stdout

Note:: Filtered renderings contains always the root object and the added children might have
confusing indentation levels because the parent elements might be filtered out

Parameters

- **itree_object** – iTree object to be converted
- **item_filter** – 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

Returns

renders(*itree_object*, *item_filter=None*)

creates a pretty print string from iTree object ad returns it in a string

Note:: Filtered renderings contains always the root object and the added children might have confusing indentation levels because the parent elements might be filtered out

Parameters

- **itree_object** – iTree object to be converted
- **item_filter** – 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

Returns

string containing the pretty print output

3.7 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:

The MIT License (MIT) 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:

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 :param iterator: iterator to be checked :return: tuple (True,iterator) - empty

(False, iterator) - item inside

class itertree.itree_helpers.**iTInterval**(*lower_limit='inf'*, *upper_limit='inf'*, *lower_open=True*, *upper_open=True*, *not_in=False*, *pre_interval=None*, *pre_and=False*, *str_def=None*)

Bases: object

helper class that defines an interval for range definitions in Data Models or Filters

the class contains a check if a given value is in the defined interval or not

The class might be a little bit under estimated in all the itertree functionalities but its a short but very powerful implementation of an Interval class for python.

The class contains anything you might need in case of a Interval functionality. You can given open/closed interval definitions including infinite limits. The intervals can be combined to a mathematical set via the pre_interval parameter. And the check method allows to give other limits as defined. This is especially useful for dynamically calculated limits.

The interval definition is also possible via a mathematical string like: "(1,2)" or "[10,+inf)".

If you need a more advanced implementation you might have a look on the intervals/portion python package.

Note: For equal just set upper_limit to None (upper_open, lower_open parameter will be ignored in this case)

INF = 'inf'

property is_equal

check(*value*, *use_limits=None*, *return_iterator=False*)

main check function :param value: value to be check if in interval or not (you might give iterables too! :param use_limits: You can replace the static limits in the interval with dynamic ones given in the check, any

nested iterable can be used here (do not use iterators!). None - use static limit (lower,_limit, upper_limit) - replace limits in highest level interval if lower_limit or

upper_limit is None the static one is used

(((lower_limit_l2,upper_limit_l2),(lower_limit_l1,upper_limit_l1)),(lower_limit_l0,upper_limit_l0)) - use nested tuples to give replacement limits to deeper levels (use None for using static ones)

Returns

True/False or iterator over single value check use any() to get a summary!

math_repr()

mathematical string representation of the interval :return: string

from_str(*interval_str*)

create the interval from a math representation string .. note:: Give inf for infinity :param interval_str: math string representation :return:

class itertree.itree_helpers.**iTLink**(*file_path=None*, *key_path=None*, *link_item=None*)

Bases: object

Definition of a link to an element in another DataTree

property loaded

property is_loaded

property link_item

property file_path

property key_path

property is_iTLink

property link_tag

property link_data

property source_path

set_source_path(*path*)

set_loaded(*tag=None*, *data=None*)

dict_repr()

class itertree.itree_helpers.**iTMatch**(*pattern*, *combine_or=True*)

Bases: object

The match object is used to defined match to elements in the DtaTree used in iterations over the DataTree The defined iMatch object can be used for checks against iTree objects (mainly for checks against the tag and also for string matches e.g. for finding iTree.data.keys() or .values() in filters.

property is_iTMatch

check(*item*, *item_filter=None*)

class itertree.itree_helpers.**TagIdx**(*tag*, *idx*)

Bases: tuple

idx

Alias for field number 1

tag

Alias for field number 0

class itertree.itree_helpers.**TagIdxStr**(*tag_idx_str*, *tag_separator='#'*)

Bases: *TagIdx*

Define a TagIdx by a sting with an index separator (default='#')

Example: "mytag#1" will be translated in the TagIdx("mytag",1)

Note: This makes only sense and can only be used if the tag is a string (not for other objects)

Parameters
tag_idx_str – string containing the definition

property is_TagIdxStr

class itertree.itree_helpers.**TagIdxBytes**(*tag_idx_bytes*, *tag_separator=b'#'*)

Bases: *TagIdxStr*

Define a TagIdx by bytes with an index separator (default=b'#')

Example: b"mytag#1" will be translated in the TagIdx(b"mytag",1)

Note: This makes only sense and can only be used if the tag is a byte (not for other objects)

Parameters
tag_idx_bytes – bytes containing the definition

property is_TagIdxBytes

class itertree.itree_helpers.**TagMultiIdx**(*tag*, *idxs*)

Bases: *TagIdx*

Define a TagMultiIdx

Parameters

- **tag** – item tag (can be any hashable object)
- **idxs** – This parameter can be: list of integer indexes any iterable or iterator containing index integers

  slice object

property is_TagMultiIdx

3.8 Subpackages

CHAPTER

FOUR

ITERTREE EXAMPLES PACKAGE

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.

In the file itree_usage_example.py a larger *iTree* is build ans manipulated and it is shown how the items in the tree can be reached. The example is in our opinion self explaining and we do not any more hints here.

In itree_data_examples.py we focus a bit on possible data models that might be used in *iTree*. We do not use any external packages in the examples but we recommend portion package for range definitions and also the Pydantic package might be 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.

4.1.1 Modules

4.1.2 itertree usage example

itertree.examples.itree_usage_example1.py

In this script we read in a part of the file system and we create an itertree which contains the some file information. Afterwards we filter on some conditions like filesize or modification times. Depending on the number of files found in the folder the first step (iTree creation) might need a short while.

After executing the script the output might look like this:

```
We read a part of the filesystem ('c:/ProgramData') into an itertree
Number of items read in 15633
The load in tree has a depth of 15
How many files are bigger then 1000000 Bytes?
Number of Matches: 934
How many files are in size 9000 ~ 10000 Bytes?
Number of Matches: 170
How many files are touched (modified) during the last day?
Number of Matches: 297
```

(continues on next page)

(continued from previous page)

```
How many files are touched (modified) during the last minute?
Number of Matches: 2
iTree('root')
            └──iTree('SelfElectController.log', data=iTData({'ACCESS': True, 'TYPE':
↪'FILE', 'EXT': 'log', 'CTIME': 1619959866.8760855, 'ATIME': 1620021514.1237395, 'FULL_
↪PATH': 'c:/ProgramData\\LANDesk\\Log\\SelfElectController.log', 'MTIME': 1620021514.
↪1237395, 'SIZE': 21081}))
                └──iTree('macompatsvc_VSL9GMPW.log', data=iTData({'ACCESS': True, 'TYPE
↪': 'FILE', 'EXT': 'log', 'CTIME': 1602947283.061818, 'ATIME': 1620021485.8823195,
↪'FULL_PATH': 'c:/ProgramData\\McAfee\\Agent\\logs\\macompatsvc_VSL9GMPW.log', 'MTIME':␣
↪1620021485.8823195, 'SIZE': 960698}))
```

4.1.3 itertree data models example

itertree.examples.itree_data_models.py

During the execution of the module we build an itertree and we fill the *iTree* objects with the data module and in a second step with the data values. Some exceptions are generated for non matching values and the formatted string representation of the data model is printed out. The script delivers the following output:

```
Run itertree data_model.py example
We build a tree for the following information:
signal_catalog
 - signal_category
   - signal
Each level in the tree contains several attributes that are stored in the data model
Build iTData structure for signal_catalog:
iTData({'creation_time': TimeModel(), 'name': StringModel(match=None, max_length=20)})
Build iTData structure for signal_category
iTData({'description': StringModel(match=None, max_length=200)})
Build iTData structure for signal
iTData({'type': StringModel(match=None, max_length=20), 'raw_data': ArrayModel(item_
↪type=FloatModel(range_interval=iTInterval(lower_limit=-10, upper_limit=10, lower_
↪open=False, upper_open=False), digits=2), max_len=None), 'gain': FloatModel(range_
↪interval=iTInterval(lower_limit=inf, upper_limit=inf, lower_open=True, upper_
↪open=True), digits=4), 'offset': FloatModel(range_interval=iTInterval(lower_limit=inf,␣
↪upper_limit=inf, lower_open=True, upper_open=True), digits=4), 'io_type':␣
↪EnumerationModel(enum_iterable_dict={1: 'INPUT', 2: 'OUTPUT'}), 'buffer_size':␣
↪IntegerModel(range_interval=iTInterval(lower_limit=0, upper_limit=1024, lower_
↪open=False, upper_open=False), representation=2), 'address': IntegerModel(range_
↪interval=iTInterval(lower_limit=0, upper_limit=inf, lower_open=False, upper_
↪open=False), representation=1)})
Build the tree
Type check example
Enter int as name and catch exception
Exception caught: iDataValueError('Given value of wrong type')
Enter creation time
Creation time value: 1649524264.1243489
Creation time string representation: 2022-04-09 19:11:04.124349
Create a category
Enter a to long description and catch exception
```

(continues on next page)

(continued from previous page)

```
Exception caught: iDataValueError('Given value contains to many characters (max_
↪length=200)')
Enter a array item out of range
Exception caught: iDataValueError('Given sub_value (index: 8)-> Value: 10.1 not in␣
↪range: [-10,10]')
raw_data_string ['1.00', '2.00', '3.00', '4.00', '5.00', '6.00', '7.00', '8.00', '9.90']
gain (see number of digits=4!) 1.0230
Enter invalid enumerate number
Exception caught: iDataValueError('Value: 3 not in enumeration definition')
io_type enum string INPUT
address as hex representation 0xff1234
Enter invalid buffer_size in update()
Exception caught: iDataValueError("Item ('buffer_size',-1): Value: -1 not in range: [0,
↪1024]")
CONSTRUCTED TREE:
iTree('signal_catalog', data=iTData({'creation_time': TimeModel(value= 1649524264.
↪1243489), 'name': StringModel(value= 'my signal catalog', match=None, max_length=20)}))
    └──iTree('analog signals', data=iTData({'description': StringModel(value= 'Digital␣
↪signals (switches and state inputs)', match=None, max_length=200)}))
        └──iTree('power voltage', data=iTData({'type': StringModel(value= 'analog input
↪', match=None, max_length=20), 'raw_data': ArrayModel(value= [1, 2, 3, 4, 5, 6, 7, 8,␣
↪9.9], item_type=FloatModel(range_interval=iTInterval(lower_limit=-10, upper_limit=10,␣
↪lower_open=False, upper_open=False), digits=2), max_len=None), 'gain':␣
↪FloatModel(value= 1.023, range_interval=iTInterval(lower_limit=inf, upper_limit=inf,␣
↪lower_open=True, upper_open=True), digits=4), 'offset': FloatModel(value= 0.0183,␣
↪range_interval=iTInterval(lower_limit=inf, upper_limit=inf, lower_open=True, upper_
↪open=True), digits=4), 'io_type': EnumerationModel(value= 1, enum_iterable_dict={1:
↪'INPUT', 2: 'OUTPUT'}), 'buffer_size': IntegerModel(value= 256, range_
↪interval=iTInterval(lower_limit=0, upper_limit=1024, lower_open=False, upper_
↪open=False), representation=2), 'address': IntegerModel(value= 16716340, range_
↪interval=iTInterval(lower_limit=0, upper_limit=inf, lower_open=False, upper_
↪open=False), representation=1)}))
        └──iTree('power current', data=iTData({'type': 'analog input', 'raw_data': [1,␣
↪2, 3, 4], 'gain': 1, 'offset': 0, 'io_type': 1, 'buffer_size': 100, 'address': 123}))
        └──iTree('power control', data=iTData({'type': 'analog output', 'raw_data':␣
↪ArrayModel(value= [1, 2, 3, 4, 5, 6, 7, 8, 9.9], item_type=FloatModel(range_
↪interval=iTInterval(lower_limit=-10, upper_limit=10, lower_open=False, upper_
↪open=False), digits=2), max_len=None), 'gain': 1.0, 'offset': 0, 'io_type': 2, 'buffer_
↪size': IntegerModel(value= 256, range_interval=iTInterval(lower_limit=0, upper_
↪limit=1024, lower_open=False, upper_open=False), representation=2), 'address': 456}))
    └──iTree('digital signals', data=iTData({'description': StringModel(value= 'Digital␣
↪signals (switches and state inputs)', match=None, max_length=200)}))
        └──iTree('power switch', data=iTData({'type': 'digital output', 'raw_data':␣
↪ArrayModel(value= [1, 2, 3, 4, 5, 6, 7, 8, 9.9], item_type=FloatModel(range_
↪interval=iTInterval(lower_limit=-10, upper_limit=10, lower_open=False, upper_
↪open=False), digits=2), max_len=None), 'gain': FloatModel(value= 1.023, range_
↪interval=iTInterval(lower_limit=inf, upper_limit=inf, lower_open=True, upper_
↪open=True), digits=4), 'offset': FloatModel(value= 0.0183, range_
↪interval=iTInterval(lower_limit=inf, upper_limit=inf, lower_open=True, upper_
↪open=True), digits=4), 'io_type': 2, 'buffer_size': IntegerModel(value= 256, range_
↪interval=iTInterval(lower_limit=0, upper_limit=1024, lower_open=False, upper_
↪open=False), representation=2), 'address': 789}))
```

4.1.4 itertree link example

itertree.examples.itree_link_example1.py

This example file should show the user how links can be used and how the links are stored.

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')
     └──iTreeLink('internal_link', link=iTreeLink(file_path=None, key_path=['/',␣
↪TagIdx(tag='B', idx=1)]))

`iTree` with linked element with links loaded:
iTree('root')
     └──iTree('A')
     └──iTree('B')
     └──iTree('B')
         └──iTree('Ba')
         └──iTree('Bb')
         └──iTree('Bb')
         └──iTree('Bc')
     └──iTreeLink('internal_link', link=iTreeLink(file_path=None, key_path=['/',␣
↪TagIdx(tag='B', idx=1)]))
         └──iTreeLink('Ba')
         └──iTreeLink('Bb')
         └──iTreeLink('Bb')
         └──iTreeLink('Bc')

iTree('root')
     └──iTree('A')
     └──iTree('B')
     └──iTree('B')
         └──iTree('Ba')
         └──iTree('Bb')
         └──iTree('Bb')
         └──iTree('Bc')
         └──iTree('B_post_append')
     └──iTreeLink('internal_link', link=iTreeLink(file_path=None, key_path=['/',␣
↪TagIdx(tag='B', idx=1)]))
         └──iTreeLink('Ba')
         └──iTreeLink('Bb')
         └──iTreeLink('Bb')
         └──iTreeLink('Bc')
`iTree` with updated linked element but no reload of the links:

iTree('root')
```

(continues on next page)

(continued from previous page)

```
     └──iTree('A')
     └──iTree('B')
     └──iTree('B')
         └──iTree('Ba')
         └──iTree('Bb')
         └──iTree('Bb')
         └──iTree('Bc')
         └──iTree('B_post_append')
     └──iTreeLink('internal_link', link=iTreeLink(file_path=None, key_path=['/',␣
↪TagIdx(tag='B', idx=1)]))
         └──iTreeLink('Ba')
         └──iTreeLink('Bb')
         └──iTreeLink('Bb')
         └──iTreeLink('Bc')

`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')
     └──iTreeLink('internal_link', link=iTreeLink(file_path=None, key_path=['/',␣
↪TagIdx(tag='B', idx=1)]))
         └──iTreeLink('Ba')
         └──iTreeLink('Bb')
         └──iTreeLink('Bb')
         └──iTreeLink('Bc')
         └──iTreeLink('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')
     └──iTreeLink('internal_link', link=iTreeLink(file_path=None, key_path=['/',␣
↪TagIdx(tag='B', idx=1)]))
         └──iTreeLink('Ba')
         └──iTreeLink('Bb')
         └──iTree('Bb')
             └──iTree('sublocal')
         └──iTreeLink('Bc')
         └──iTreeLink('B_post_append')
         └──iTree('new')
```

(continues on next page)

(continued from previous page)

```
`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')
     └──iTreeLink('internal_link', link=iTreeLink(file_path=None, key_path=['/',␣
↪TagIdx(tag='B', idx=1)]))
         └──iTreeLink('Ba')
         └──iTreeLink('Bb')
         └──iTreeLink('Bb')
         └──iTreeLink('Bc')
         └──iTreeLink('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 TagIdx of the local item Bb[1]
iTree('root')
     └──iTree('A')
     └──iTree('B')
     └──iTree('B')
         └──iTree('Ba')
         └──iTree('Bb')
         └──iTree('Bb')
         └──iTree('Bc')
         └──iTree('B_post_append')
     └──iTreeLink('internal_link', link=iTreeLink(file_path=None, key_path=['/',␣
↪TagIdx(tag='B', idx=1)]))
         └──iTreePlaceHolder('Bb')
         └──iTree('Bb')
             └──iTree('sublocal')
         └──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')
     └──iTreeLink('internal_link', link=iTreeLink(file_path=None, key_path=['/',␣
↪TagIdx(tag='B', idx=1)]))
         └──iTreeLink('Ba')
```

(continues on next page)

(continued from previous page)

```
        └──iTreeLink('Bb')
        └──iTree('Bb')
            └──iTree('sublocal')
        └──iTreeLink('Bc')
        └──iTreeLink('B_post_append')
        └──iTree('new')
```

4.1.5 itertree editor example

This program is using Tkinter to create a GUI to create and manipulate *itertree* objects. The program should show how an 'iTree'-item can be coupled with an item in a tree structure of the GUI (use the coupled_object functionality of *iTree*). THe editor allows diffrent manipulations on the tree structure from renaming and ordering to manipulations of the data (add some models, etc) and the creation of additional items (*iTree* / *iTreeLink* , *iTreeReadOnly*). The context menus are also created via a defintions based on *iTree* objects. The GUI code example is splittet in an controller and the GUI itself. This allows better testing of the functionalities if required and brings a lot more advantages. But we should not dive in the discussions of GUI related architecture here, as long we have here just another example for the usage of itertree.

Note: Please do not report issues related to the editor on GIT. We know that a lot of corner-cases are not covered

and that the editor functionalities are incomplete. It's just an example and not an application we provide here.

4.1.6 itertree performance example

There are two performance tests found under examples. They are not created for learning proposes furthermore the user can see how we have run some performance test against other solutions that targeting in same direction as the itertree package. For more details have a look on the *Comparison* of the documentation.

4.1.7 itertree profiler example

The example contains one of the profiling we did to optimize the *iTree* class for the main operations.

Based on those analysis you can see which operations needs the most calculation time. The output looks like:

```
Running on itertree version: 0.8.0
         5400015 function calls (5300015 primitive calls) in 4.132 seconds

   Ordered by: standard name

   ncalls  tottime  percall  cumtime  percall filename:lineno(function)
        1    0.000    0.000    4.132    4.132 <string>:1(<module>)
   700003    0.520    0.000    0.520    0.000 itree_data.py:203(__init__)
   400001    0.280    0.000    0.456    0.000 itree_data.py:220(__copy__)
   100000    0.180    0.000    0.228    0.000 itree_main.py:1087(insert)
   100000    0.130    0.000    0.187    0.000 itree_main.py:1127(append)
        1    0.000    0.000    0.897    0.897 itree_main.py:1178(extend)
        1    0.000    0.000    0.000    0.000 itree_main.py:1434(iter_children)
   500003    0.766    0.000    1.780    0.000 itree_main.py:148(__init__)
```

(continues on next page)

(continued from previous page)

```
200003/100003    0.463    0.000    1.237    0.000 itree_main.py:2073(_load_subtree)
   200000    0.088    0.000    0.121    0.000 itree_main.py:255(__getitem__)
   100000    0.084    0.000    0.124    0.000 itree_main.py:288(__delitem__)
        1    0.001    0.001    1.062    1.062 itree_main.py:373(__mul__)
        1    0.028    0.028    0.916    0.916 itree_main.py:385(<listcomp>)
        1    0.000    0.000    0.000    0.000 itree_main.py:409(__iter__)
   200000    0.760    0.000    1.607    0.000 itree_main.py:957(__copy__)
   200000    0.024    0.000    0.024    0.000 itree_main.py:972(<listcomp>)
        1    0.393    0.393    4.132    4.132 itree_profile.py:41(performance_dt)
        1    0.086    0.086    0.402    0.402 itree_profile.py:51(<listcomp>)
        1    0.000    0.000    4.132    4.132 {built-in method builtins.exec}
   200001    0.017    0.000    0.017    0.000 {built-in method builtins.isinstance}
   100000    0.009    0.000    0.009    0.000 {built-in method builtins.iter}
   100000    0.008    0.000    0.008    0.000 {built-in method builtins.len}
   400001    0.036    0.000    0.036    0.000 {function iTData.copy at␣
↪0x000001F206B4DA60}
   100000    0.013    0.000    0.013    0.000 {function `iTree`.__getitem__ at␣
↪0x000001F206BBC940}
   699998    0.069    0.000    0.069    0.000 {function `iTree`.append at␣
↪0x000001F206BC9CA0}
   100000    0.026    0.000    0.026    0.000 {function `iTree`.insert at␣
↪0x000001F206BC9C10}
   100000    0.032    0.000    0.032    0.000 {function `iTree`.pop at␣
↪0x000001F206BC9EE0}
   499997    0.077    0.000    0.077    0.000 {method '__contains__' of 'dict' objects}
   399998    0.040    0.000    0.040    0.000 {method '__getitem__' of 'dict' objects}
        1    0.000    0.000    0.000    0.000 {method 'disable' of '_lsprof.Profiler'␣
↪objects}
```

and for second profiling script:

```
       6934372 function calls in 3.239 seconds

 Ordered by: standard name

 ncalls  tottime  percall  cumtime  percall filename:lineno(function)
      1    0.000    0.000    3.239    3.239 <string>:1(<module>)
1050804    0.215    0.000    2.915    0.000 itree_helpers.py:56(accu_iterator)
 535806    1.338    0.000    2.520    0.000 itree_main.py:1504(find_all)
 525402    0.229    0.000    2.700    0.000 itree_main.py:1688(<lambda>)
 535806    0.312    0.000    0.446    0.000 itree_main.py:2133(__extract_first_iter_
↪items)
 535806    0.212    0.000    0.323    0.000 itree_main.py:2163(__build_find_all_result)
 535806    0.290    0.000    0.349    0.000 itree_main.py:255(__getitem__)
      1    0.206    0.206    3.239    3.239 itree_profile2.py:77(performance_it_find_all_
↪by_idx)
      1    0.000    0.000    3.239    3.239 {built-in method builtins.exec}
 535806    0.081    0.000    0.081    0.000 {built-in method builtins.hasattr}
 535806    0.052    0.000    0.052    0.000 {built-in method builtins.isinstance}
 535806    0.060    0.000    0.060    0.000 {built-in method builtins.iter}
 535806    0.053    0.000    0.053    0.000 {built-in method builtins.len}
  10404    0.068    0.000    2.937    0.000 {built-in method builtins.next}
```

(continues on next page)

(continued from previous page)

```
 525402    0.064    0.000    0.064    0.000 {built-in method from_iterable}
 535806    0.059    0.000    0.059    0.000 {function `iTree`.__getitem__ at␣
↪0x0000029A579A2AF0}
    102    0.000    0.000    0.000    0.000 {method 'append' of 'list' objects}
      1    0.000    0.000    0.000    0.000 {method 'disable' of '_lsprof.Profiler'␣
↪objects}
```

CHAPTER

FIVE

COMPARISON

In this chapter we compare the itertree package with other packages which are targeting in the same direction.

Each package is develop with a specific focus and therefore a comparison is always a bit misleading. Finally the comparison remarks you find in this chapter are no not at all a judgement of the other packages. Especially the performance tests can also be misleading because we may not have utilized the other packages in the right way.

In this chapter we compare *iTree* also with the standard types like dict and lists. Additionally we have a look on xml.ElementTree, sorted_dict (from sorted containers) and the anytree package.

In the design paradigms of the itertree package can be summarized by the following topics. They will be highlighted and compared:

1. We can add any type of tag in the *iTree* as long as it is hashable and we can add the same tag multiple times in the *iTree*. Some of the comparable package support only string-type tags (like anytree or xml.ElementTree). Other allow only unique tags like the keys in dicts (using same key will overwrite the already existing tag in this case).

2. In *iTree* the item access via index and tag (or TagIdx) is possible. As you will see in the performance tests later many of the other packages are focus on one type of access only and the second type is then much slower (optimized for key or optimized index access only). It is part of the design paradigm related to the classes (E.g. it's quite clear that index access on huge dicts or key access in huge list will be very slow). But even the search mechanisms in specialized packages are very often really slow (compared to iTree).

3. The access of multiple items via index list is possible *my_itree[[1,4,5,6,9]]* will deliver the indexed items in an iterator (The access via index-list is in most packages not supported).

4. As in the introduction already explained the results when running filter queries in *iTree* will be delivered very quick because we delivering always iterators. But this might make the coding from the point of usage sometimes a bit more complicate because if you need index access to a specific element you must cast the iterator in a list (by *list(my_iterator)*) or you use or use *itertools.is_slice()* operation. It's always recommended to address the target items via the available iTree methods (*find()*, *find_all()* or even *iTree[my_identifier]*) directly. You can also use the item_filter and matches to reach your results as good as possible.

5. We can link multiple source files into one *iTree* object or even create link inside the tree itself, also we can cover linked items by local items. Most of the packages do not support links and we do not know any other package supporting item covering (overloading). Even the load and storage into files is most often not supported especially considering the data object stored in the item must be serialized in this case too. (Users can always create serializers but this can be sometimes very difficult considering all data-types stored in a tree). *iTree* delivers out of the box the possibility to store several data-types into a JSON file (e.g. also numpy arrays if needed).

6. At least the data in the *iTree* objects can be combined with a data model that checks that the give data values matching with the ones expected by the data model. The defined data models allows much more then just a check of the data type. E.g. one can also define ranges or intervals in which an integer value must fit in. This functionality makes *iTree* objects very attractive for the storage of certain configuration data.

Finally we must also mention that sorting of items is not in focus of this package.

Under examples you can find the "itree_performance.py" file which contains a short performance test regarding other comparable packages. The following results are create under Python 3.9 and blist package installed. Please feel free to adapt the first line regarding the tree size and the number of repetitions when you run your own tests. In case no blist package is installed you may skip the insert operation of *iTree* which is slowed down a lot.

The measured times given are always relative to one operation.

The tests are only performed in case the needed package is available in the local installation if the module is not found the test is skipped. The user can find some experimental not published packages imported in the code, this should be ignored.

Running the test on a tree with 5000 items delivers the following result on my PC under python 3.9.

```
>>>python iter_performance.py
We run for treesizes: 5000 with 4 repetitions
Python:  3.9.2 (tags/v3.9.2:1a79785, Feb 19 2021, 13:44:55) [MSC v.1928 64 bit (AMD64)]
blist package is available and used
itertree version: 0.8.0
A relative values >1 related to `iTree` means the other object is faster
(relative values <1 means `iTree` is faster)
Exectime time itertree build: 0.014122499999999996
Exectime time itertree build: with subtree list comprehension: 0.011322650000000004
Exectime time itertree build (with insert): 0.014225149999999992
Exectime time itertree tag access: 0.0018194249999999995
Exectime time itertree tag index access: 0.0037113499999999883
Exectime time itertree tag index tuple access: 0.0025402999999999953
Exectime time itertree index access: 0.0018937750000000003
Exectime time itertree convert iter_all iterator to list: 0.0024619749999999913
Exectime time itertree save to file: 0.018669550000000007
Exectime time itertree load from file: 0.030752224999999994
Loaded `iTree` is equal: True
-- Standard classes -----------------------------------
Exectime time dict build: 0.0013473249999997883 ~ 10.482x faster as iTree
Exectime time dict key access: 0.0010821000000000858 ~ 1.681x faster as iTree
Exectime time dict index access: 0.1515084499999999 ~ 0.012x faster as iTree
Exectime time list build (via comprehension): 0.0007731250000000411 ~ 14.645x faster as␣
↪iTree
Exectime time list build (via append): 0.0010298500000001098 ~ 13.713x faster as iTree
Exectime time list build (via insert): 0.007580025000000212 ~ 1.877x faster as iTree
Exectime time list index access: 0.00014650000000004937 ~ 12.927x faster as iTree
Exectime time list key access: 0.1558285 ~ 0.012x faster as iTree
Exectime time OrderedDict build: 0.0010928749999998821 ~ 12.922x faster as iTree
Exectime time OrderedDict key access: 0.0007975499999999247 ~ 0.002x faster as iTree
Exectime time deque build (append): 0.0009357999999999311 ~ 15.091x faster as iTree
Exectime time deque build (insert): 0.0012067749999999933 ~ 11.788x faster as iTree
Exectime time deque index access: 0.00027527499999990823 ~ 6.880x faster as iTree
-- SortedDict ----------------------------------
Exectime time SortedDict build: 0.03194094999999986 ~ 0.442x faster as iTree
Exectime time SortedDict key access: 0.0011537749999999125 ~ 1.577x faster as iTree
Exectime time SortedDict index access: 0.005190625000000004 ~ 0.365x faster as iTree
-- xml ElementTree ------------------------------------
Exectime time xml ElementTree build: 0.0017907750000001332 ~ 7.886x faster as iTree
Exectime time xml ElementTree key access: 0.16554792499999982 ~ 0.011x faster as iTree
Exectime time xml ElementTree index access: 0.00016492499999998245 ~ 11.032x faster as␣
↪iTree
```

(continues on next page)

(continued from previous page)

```
-- anytree ---------------------------------
Exectime time Anytree build: 0.6392311500000001 ~ 0.022x faster as iTree
Exectime time Anytree key access (no cache): 20.658143574999997 ~ 0.000088x faster as␣
↪iTree
Exectime time Anytree index access: 0.06119849999999971 ~ 0.031x faster as iTree
```

Running the test on a tree with a depth of 150 levels and 22500 items delivers the following result on my PC under python 3.5.

```
>>>python iter_performance2.py
We run for deep tree sizes: depth of 150 with 22500 items and 4 repetitions
Python:  3.9.2 (tags/v3.9.2:1a79785, Feb 19 2021, 13:44:55) [MSC v.1928 64 bit (AMD64)]
blist package is available and used
itertree version: 0.8.0
A relative values >1 related to `iTree` means the other object is faster
(relative values <1 means `iTree` is faster)
Exectime time itertree build append: 0.053359225
Exectime time itertree build (with insert): 0.06587992499999999
Max tree depth 150
Exectime time itertree get max_depth_down~iter_all(): 0.0105537
Exectime time itertree get deep indexes access (all items iterated): 0.5943447749999999
Exectime time itertree get find_all by indexes access (all items iterated): 4.701620525
Exectime time itertree find all by deep tag list (one deep search last item): 0.
↪08802357500000024
-- Standard classes -----------------------------------
Exectime time dict build: 0.007973800000000253 ~ 6.692x faster as iTree
Exectime time dict key access: 0.11559847499999965 ~ 0.761x faster as iTree
Exectime time list build (via comprehension): 0.006427750000000287 ~ 8.301x faster as␣
↪iTree
Exectime time list index access: 0.04177927499999967 ~ 14.226x faster as iTree
-- SortedDict ----------------------------------
Exectime time SortedDict build: 0.14082195000000003 ~ 0.379x faster as iTree
Exectime time SortedDict key access: 0.13243777499999965 ~ 0.665x faster as iTree
-- xml ElementTree -----------------------------------
Exectime time xml ElementTree build: 0.00898362499999994 ~ 5.940x faster as iTree
Exectime time xml ElementTree key access: 2.8548865250000004 ~ 0.031x faster as iTree
Exectime time xml ElementTree index access: 0.05549647499999999 ~ 10.710x faster as iTree
-- anytree ---------------------------------
Exectime time Anytree build: 0.3895624249999994 ~ 0.137x faster as iTree
Anytree key access skipped -> slow
Exectime time Anytree index access: 1.0371582999999998 ~ 0.573x faster as iTree
```

I have following comments on the findings:

1. *iTree* objects behave ~ 8-16 times slower then the build in objects like dict, lists, etc. Reason is mainly that *iTree* is a pure python package which does not has the the speed advantage of an underlying C-Layer. Anyway a 20 times slower execution is really not an issue from our point of view. Please consider the wide range of functionalities found in *iTree* objects.
2. For untypical access of dict per idx or list per key the builtin objects perform ~ 100 times slower than *iTree*.
3. The other tree like packages are on par or slower then *iTree* (in some cases incredible slower). An exception is the package xml-ElementTree which incredible fast in case of index access (quicker then builtin lists).

On a large tree of 500000 we have the following findings:

```
We run for treesizes: 500000 with 4 repetitions
Python:  3.9.2 (tags/v3.9.2:1a79785, Feb 19 2021, 13:44:55) [MSC v.1928 64 bit (AMD64)]
blist package is available and used
itertree version: 0.8.0
A relative values >1 related to `iTree` means the other object is faster
(relative values <1 means `iTree` is faster)
Exectime time itertree build: 1.4585138
Exectime time itertree build: with subtree list comprehension: 1.317420325
Exectime time itertree build (with insert): 1.5535431249999996
Exectime time itertree tag access: 0.23381625000000028
Exectime time itertree tag index access: 0.5307640249999999
Exectime time itertree tag index tuple access: 0.4094945000000001
Exectime time itertree index access: 0.21780237500000066
Exectime time itertree convert iter_all iterator to list: 0.27708437500000027
Exectime time itertree save to file: 2.1980745499999994
Exectime time itertree load from file: 2.7010892500000008
Loaded `iTree` is equal: True
-- Standard classes -----------------------------------
Exectime time dict build: 0.15743670000000165 ~ 9.264x faster as iTree
Exectime time dict key access: 0.11920657499999976 ~ 1.961x faster as iTree
Exectime time dict index access: skipped incredible slow
Exectime time list build (via comprehension): 0.07432719999999904 ~ 17.725x faster as␣
↪iTree
Exectime time list build (via append): 0.09793205000000071 ~ 14.893x faster as iTree
Exectime time list build (via insert): Skipped very slow
Exectime time list index access: 0.025543875000000327 ~ 8.527x faster as iTree
Exectime time list key access: Skipped incredible slow
Exectime time OrderedDict build: 0.17470362499999936 ~ 8.349x faster as iTree
Exectime time OrderedDict key access: 0.11788422500000095 ~ 0.234x faster as iTree
Exectime time deque build (append): 0.10968872499999804 ~ 13.297x faster as iTree
Exectime time deque build (insert): 0.1312096000000018 ~ 11.840x faster as iTree
Exectime time deque index access: 7.638674499999997 ~ 0.029x faster as iTree
-- SortedDict -----------------------------------
Exectime time SortedDict build: 3.445377900000004 ~ 0.423x faster as iTree
Exectime time SortedDict key access: 0.1740121499999958 ~ 1.344x faster as iTree
Exectime time SortedDict index access: 1.105328924999995 ~ 0.197x faster as iTree
-- xml ElementTree -----------------------------------
Exectime time xml ElementTree build: 0.20869660000000323 ~ 6.989x faster as iTree
xml ElementTree key access skipped -> too slow
Exectime time xml ElementTree index access: 0.019160849999998675 ~ 12.203x faster as␣
↪iTree
-- anytree -----------------------------------
Exectime time Anytree build: 5641.44443335 ~ 0.000x faster as iTree
Anytree key access skipped -> incredible slow
Exectime time Anytree index access: not working
```

Some of the steps are skipped because very bad performance (some functions need hours).

Insertion of elements in lists is very slow. This might only be a minor corner case because filling a list might always be done by append() or even better with a list comprehension. The *iTree* insertion mechanism (based on blist) works much quicker and is nearly on the speed of append(). But we also recommend list comprehension mechanism for quickest filling of *iTree* objects too. The mayor time in filling an *iTree* goes into instance the object (*__init__*) and if needed in the internal *copy()* of *iTree* items (e.g. see *extend()* method).

5.1 iTree vs. dict / collections.OrderedDict

For the base functionality storing data paired with hashable objects as keys in a data structure where one can find the data by giving the key the dict is quicker then iTree (10x quicker for the building of the structure and 2x quicker for the item access). But we have a lot of limitations. We cannot store one and the same hashable object (key) multiple times in the dict (item will always be overwritten). You can build nested dicts by putting sub dicts into dict keys (building nested structures is only 7x quicker). But the access to this nested structure is very limited no deep iterations are available out of the box. Also search queries must be programmed outside the dict structure. The normal dict does not support ordered storage in older python versions, only the OrderedDict extension does this. At least we do not have access to the order by index we always must create an iterator that can be misused for index access.

Summary: It's not surprising that the main functional target (key based operations) of the build-in dict object are quicker compared with the key (tag) based operations we have on *iTree*. But the a dict is a flat unordered structure and there is no build-in functionality related to trees. Considering the overall functionality of *iTree* in all highlighted directions the speed difference even compared with the "core" functions of a dict are still more than acceptable from our point of view.

5.2 iTree vs. list / collections.deque

For lists and nested list we can found the same pros and cons we described for dicts in the last chapter except that the access in list is focused on index and not by keys. We can say that index access in iTrees is also the most performant way to access items (quicker then tag or TagIdx based access). Insert operations in lists can be also very slow. For huge trees we recommend to install blist package which out-performances lists in a lot of circumstances (We still don't understand why the blist implementation is not used as standard list in python as proposed by the author). Beside the tag based access *iTree* objects can also be reached via index lists (not available in lists). The deque object behave in general as lists. We can quicker insert elements (link-list extension is easy) but get an items index() works much slower as in normal lists.

Summary: For the core functions lists and deque are 10-18 times quicker than *iTree*. But key access is very limited.

5.3 iTree vs. xml ElementTree

The xml ElementTree package goes very much in the same direction as the *iTree* package. The performance regarding any list related action is very good and much better than *iTree* can deliver (C-Layer).

But the handling of ElementTrees is totally different. Trees are normally build by external factory functions even that an internal build interface is available too (list like behavior). The same tag can be stored multiple times in an ElementTree (same as in itertree). As the naming tells the package is mainly build to provide all xml related data structures and functionalities. And the storage and loading into/from files is widely support. By the way serializing of none string objects in the tree must be managed and organized by the user. The item identification is made via string only tags and you can't use hashable object as tags (like in iTree). Even the string usage is limited to the xml naming convention (e.g. no spaces are allowed). For queries in the tree one can use the powerful xpath syntax. But we think the *iTree* filter functions are comparable and because we use filter objects we are more flexible especially very special filter conditions.

Beside the pure index access *iTree* is for any operation quicker than the ElementTree (which is surprising because ElementTree is a c-based implementation). Especially when searching for specific tags and filtering we see bigger advantages for *iTree* (not all seen in the performance test). Serialization and storage in *iTree* is more efficient than in ElementTree. But *iTree* does not have all the xml powered higher level functionalities like schemata, etc. which are support by ElementTree (which is really not the target of iTree). As last remark we can say an xml-serialization of *iTree* objects might be easy implemented if needed.

5.4 iTree vs. sorted_dict

The sorted_dict package from sorted_containers might be used for the same proposes *iTree* is build for. But the architecture for realization is a bit different. Sorted_dict supports key and index based access. But one cannot store same key multiple times (behavior is here the same as in normal dicts). The *iTree* object has not the target of sorting items in different ways. Furthermore *iTree* tries to realize filtered access to the items by keeping the original order. In one first approach the author tried to realize the *iTree* functionalities with an underlying sorted_dict. But the performance of the approach was worse and we changed the strategy. iTree does not support the grouping function (union, intersection, etc.) supported by sorted-dicts. The performance of sorted-dicts regarding the design paradigms of *iTree* is less good. Especially building a instance of sorted-dict objects of a huge number is 2 times slower than for *iTree* objects. Key access is on par with normal dicts and 2 time quicker than in *iTree*.

5.5 iTree vs. anytree

The anytree packages gains mostly in the same direction as itertree. You can find nearly comparable serialization possibilities. The rendering found in *iTree* is a simple "copy" of what you can get in anytree. As in *iTree* objects you can combine children of same name with a parent in anytree too. But there are 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

Before the itertree package was developed we thought anytree is the solution to go for and there is no need for a new package like itertree. But the results of the anytree package tests we did where very ambiguous. Anytree has a very huge feature-set but also really poor performance. This was also shortly discussed with the author: https://github.com/c0fec0de/anytree/issues/169.

At least we came to the conclusion that anytree seems not match to our requirements for tree structured storage and access. From description it should match, but in practice the package did not work for us as expected.

Summary: For small trees anytree might be an alternative to *iTree* but when getting to bigger structures (more elements deeper levels) or when effective filtering is needed *iTree* has very huge advantages.

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 is in Python 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.x 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 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:

- *iTree* (list) -> main list of items
- _map (dict) -> dict containing the family list (key is tag)
- _data (iTData) -> data object that stores all the data attributes related to the *iTree* 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).

The profiling of the package done by running over 100000 base operations gives the following result based on blist: :: Running on itertree version: 0.6.1 100003 0.161 0.000 0.342 0.000 itree_main.py:111(__init__) 100000 0.044 0.000 0.059 0.000 itree_main.py:269(__getitem__) 100000 0.090 0.000 0.383 0.000 itree_main.py:302(__delitem__) 100000 0.239 0.000 0.258 0.000 itree_main.py:870(append) 100000 0.269 0.000 0.286 0.000 itree_main.py:829(insert) 100000 0.160 0.000 0.891 0.000 itree_main.py:725(__copy__) 1 0.154 0.154 0.977 0.977 itree_main.py:919(extend) 100000 0.067 0.000 0.089 0.000 itree_main.py:622(idx)

We can see that creating copies is the most time consuming operation and it is the reason why the one extend() operation takes so long.

Running the same profiling actions without blist package (using normal list) we get: :: 100003 0.161 0.000 0.320 0.000 itree_main.py:111(__init__) 100000 0.052 0.000 0.060 0.000 itree_main.py:269(__getitem__) 100000 0.094 0.000 1.266 0.000 itree_main.py:302(__delitem__) 100000 0.140 0.000 0.161 0.000 itree_main.py:870(append) 100000 0.228 0.000 1.895 0.000 itree_main.py:829(insert) 100000 0.129 0.000 0.701 0.000 itree_main.py:725(__copy__) 1 0.149 0.149 0.914 0.914 itree_main.py:919(extend) 100000 0.082 0.000 0.097 0.000 itree_main.py:622(idx)

Especially the index based searches in the lists are take much longer. And especially the *insert()* take exceptionally much longer but one may see this as a corner case only because the filling of a huge tree will normally always be done by appending or even better by extending elements. Inserting a single item is absolutely no issue! Please consider we talk here about a very huge number of *insert()* operations (100000). Same arguments can be made for the *__delitem__()* operation nobody will delete all the items step by step it's much easier to delete or clear the parent instead.

We can summarize: Except from the told corner cases the itertree package runs with the same speed (sometimes a bit faster) even that the blist package is not installed.

6.2 Special *iTree* objects

In an itertree person might need temporary items or they like to combine the tree from different sources (files). Or they might like to protect specific items from writing (read only). For this proposes we can integrate special iTree objects in the itertree.

Besides the normal *iTree* object we have three other types of *iTree* objects available:

- *iTreeLink* - Link to another *iTree* file/key so that an itertree can be created from different source files. Also internal linking to other branches of the root object supported. The children and sub children of these linked objects are read only. But they can be localized and by this you can cover the orignal linked items by a new structure.
- *iTreeReadOnly* - An read_only object that allows no changes in the *iTree* structure (properties (like data or coupled_object) can be changed)
- *iTreeTemporary* - a temporary *iTree* item (These items behave like normal *iTree* items except that they are not stored in a file. If *dump()* is called these items are filtered out.
- *iTreePlaceHolder* - an internally used object that is used to keep the indexing of localized items during storage.

For data protection a *iTDataReadOnly* class is available too.

6.3 Iterators 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 is not very fast when using the *find_all()* method. Beside this the string based xpath syntax is sometimes also a bit difficult and not as powerful and flexible as it might be needed for complex data structures and data objects different from strings.

In itertree we have the possibility to define filter functionalities for all the iterators delivered by the *iter_children()*, *iter_all()* or *find()* and *find_all()* methods. These methods contain a *item_filter* parameter where the user can give a filter method or class. Those objects can be cascaded to create complex filters (and/or logic supported).

The filter method is fed by the 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.iTFilter*** in the code.

Because we are using iterators 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 iterator is delivered very quick 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.

6.4 File storage and serializing

At the moment we serialize to JSON and the speed (with *orjson* module) is comparable with *pickle*. But we see that there is still room for improvements and we might get quicker results in the future. Also we might consider other output formats like *MessagePack* or *xml*.

Anyway 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 same into an archive afterwards (independent from itertree). Therefore all these options (packing, hashing) are optional and can switch off if required.

6.5 Data Structure and Data Models

The data structure of a *iTree* is not ordered (do not confuse data with the tree structure). It behaves like a normal dict (We do not see why we should create a second ordered structure here). If the user really needs this he can add any type of object into the data structure (e.g. also *OrderedDict`s). And for newer Python versions the `dict* is ordered anyway. But to be honest in this case it might be better to create a deeper *iTree* containing all the items of the *OrderedDict* in an *iTree* branch instead.

To create a better usability the data structure can be fed directly with only one data object. Alternatively the user can store also multiple objects by giving key,value pairs. Internally the *iTData* object is an overloaded dictionary.

The itertree package contains a concept for data models for the attributes stored in the data structure of the *iTree*. By this the user can determine what kind of data can be stored in a specific attribute. The *iTDataModel* is just a basic structure which can be used to create more advanced models. You might have a look in the examples/itree_data_models.py file to get a better idea.

In general the data model allows to define the target data type but furthermore also the dimension, the range, etc. Also the formatting of the data when casted into a string can be defined. E.g. we can define the following data models:

- We can define an integer in the range 0-255 in a 1 dimensional array (*list*) of maximum length 8. Additionally we like to have a hex representation when converted into a string.
- We define a float value in the range in between -250 and 250 and we like to have a string representation of maximum 3 digits and added by a unit string "V" ("%.3f V").

If a data model is stored in the data structure we can put only values into the related attribute that are matching to the model. In case of no matching values the set command will raise an *iDataValueError* exception.

Note: If define your own data_models and or *iData* classes ensure that you create a matching interface! E.g. the *check()* and *_validator()* methods must deliver the value as return (needed for recursive operations).

CHAPTER

SEVEN

ITERTREE - INTRODUCTION

Do you have to store data in a tree like structure? Do you need good performance, a reach feature set especially in case of filtered access to all data and the possibility to serialize and store the structure in files? Or do you like to use links to sub-trees and to cover items from a linked structure with new data?

Give itertree package a try!

The main class for construction of the itertree trees is the iTree class. The class allows the construction of trees like this:

```
iTree('root',data='xyz')
└─iTree('subitem1',data='abc')
    └─iTree('subsubitem1',data={'a':'b','b':'c'})
└─iTree('subitem2',data={1:2})
└─iTree('subitem2',data={2:3})
```

Every node in the itertree (iTree object) contains two parts of stored information:

- First the related sub-structure (iTree-children)
- Second the item related data attribute were any kind of object can be stored in

The itertree solution can be compared with nested dicts or lists. Other packages that targeting in the in the same direction are anytree, xml.ElementTree, sorted_containers. 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 Package Comparison.

7.1 Status and compatibility information

Version | 0.8.2| has been released!

Be sure to read the *changelog* before upgrading!

Please use the github issues to ask questions report problems.

Please do not email me directly.

The original implementation is done in Python 3.5 and it is tested under Python 3.5 and 3.9. It should work in all Python 3 environments.

The actual development status is "*beta - release candidate*".

7.2 Feature Overview

The main features of the iTree object in itertree can be summarized in:

- trees can be structured in different levels (nested trees: parent - children - sub-children - ....)
- the identification tag can be a string or any kind of object that is hashable
- tags must not be unique (same tags are enumerated and collect in a tag-family)
- item access is possible via tag, tag-index, index, slices
- iTree keeps the order of the added children
- the item related data is stored in a protected data structure where data models can be used to evaluate the given data values
- a iTree 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
- standard export/import to JSON (incl. numpy and OrderedDict data serialization)
- designed for performance (huge trees with hundreds of levels)
- it's a pure python package (should be therefore usable in all embedded environments)

Here is very simple example of itertree usage:

```
>>> from itertree import *
>>> root=iTree('root',data={'mykey':0})
>>> root+=iTree('sub',data={'mykey':1})
>>> root+=iTree('sub',data={'mykey':2})
>>> root+=iTree('sub',data={'mykey':3})
>>> root.append(iTree('sub',data={'mykey':4}))
>>> root.render()
iTree('root', data="{'mykey': 0}")
 └──iTree('sub', data="{'mykey': 1}")
 └──iTree('sub', data="{'mykey': 2}")
 └──iTree('sub', data="{'mykey': 3}")
 └──iTree('sub', data="{'mykey': 4}")
```

7.3 Documentation Content

- *Introduction* - Short introduction to the itertree package
- *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 lefthandside 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
- orjson - A quicker json parser that used to create the JSON structures during serializing/deserializing

  -> in case orjson is not found, ujson package is checked too

  -> in case both not found normal json package will be used

To install the itertree package just run the command:

```
pip install itertree
```

The structure of folder and files related to this package looks like this:

- itertree (main folder)
  - __init__.py
  - itree_main.py
  - itree_data.py
  - itree_filter.py
  - itree_helpers.py
  - itree_serialize.py
  - examples
    - itree_performance.py
    - itree_performance2.py
    - itree_profile.py
    - itree_profile2.py
    - itree_data_models.py
    - itree_usage_example1.py
    - itree_usage.py
    - itree_link_example1.py

7.4.2 First steps

All important classes of the package are published by the __init__.py file so that the functionality of itertree can be reached by simply 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.

We start now building a itertree with the recommended method for adding items. You can just use the += operator (*__iadd__()*) which adds a child item to the parent item left of =+ operator. The classical append() method is available too.

```
>>> root=iTree('root') # first we create a root element
>>> root+=iTree(tag='child', data=0) # add a child via += operator
>>> root+=iTree(tag=(1,2,3), data=1) # add next child (tag is tuple, a hashable object)
>>> root+=iTree(tag='child2', data=2) # add next child
>>> root.render() # show the current tree
iTree('root')
 └──iTree('child', data=0)
 └──iTree((1, 2, 3), data=1)
 └──iTree('child2', data=2)
```

Each iTree-object must have a tag which is the main part of the identifier of the object. For tags you can use any type of hashable objects except integers and *TagIdx* objects (these objects are used for index access and they are therefore not allowed as tags).

Different than the keys in dictionaries the given tags must not be unique:

```
>>> root+=iTree(tag='child', data=3)
>>> root+=iTree(tag='child', data=4)
>>> root.render()
iTree('root')
 └──iTree('child', data=0)
 └──iTree((1, 2, 3), data=1)
 └──iTree('child2', data=2)
 └──iTree('child', data=3)
 └──iTree('child', data=4)
```

In the iTree object equal tags are enumerated in a tag-family and they can be reached/identified via the helper object *TagIdx*.

```
>>> print(root[TagIdx('child',1)])
iTree(tag='child', data=3)
>>> print(root[3])
iTree(tag='child', data=3)
```

To add subitems we can address the child item also by index (or *TagIdx*) and add a sub-item.

```
>>> root[0]+=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.iter_children()] # iter over the children and put result in list
>>> print(a)
[iTree("'child'", data=0, subtree=[iTree("'subchild'")]), iTree("(1, 2, 3)", data=1),␣
↪iTree("'child2'", data=2), iTree("'child'", data=3), iTree("'child'", data=4)]
>>> b=[i for i in root.iter_all()] # iter over all items (all levels) and put them into␣
↪a list
>>> print(b)
[iTree("'child'", data=0, subtree=[iTree("'subchild'")]), iTree("'subchild'"), iTree("(1,
↪ 2, 3)", data=1), iTree("'child2'", data=2), iTree("'child'", data=3), iTree("'child'",
↪ data=4)]
```

The iterators and find functions of itertree have a *item_filters* parameter in which filter functions/objects can be placed in to search for specific properties. The provided filter objects can also be cascaded.

```
>>> result=root.find_all(['**'],item_filter=Filter.iTFilterDataValue(2)) # '**' is a␣
↪wildcard for any item; result is an iterator
>>> print(list(result))
[iTree(tag='child',data=2)]
```

The data handling can be done over set and get functions, if no specific key is given the *__NOKEY__* element will be addressed automatically. This is very helpful in case you want to store just one data object in the iTree-object.

```
>>> root=iTree('root')
>>> root.d_set(1)
>>> root.d_get()
1
>>> root.d_set('mykey',2)
>>> root.d_get() # the ("__NOKEY__") data item is untouched by the last operation
1
>>> root.d_get('mykey')
2
>>> item=iTree('root2',data={'A':'a','B':'b'})
>>> item.data
"{'A': 'a', 'B': 'b'}"
```

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.dtz') # dtz is the recommended file ending for the zipped dataset file
>>> root2=root.load('dt.dtz') # for loading a itertree any available iTree object can be␣
↪used
>>> print(root2==root)
True
>>> root+=iTree('link',link=iTLink(dt.dtz',iTreeTagIdx(child',0))) # The node item will␣
↪integrate the children of the linked item.
```

7.5 iterators vs. lists

We named the package itertree because when ever a iTree operation delivers multiple items the result will be an iterator (and not a list what the user might expect).

Iterators are very powerful objects especially if you have a huge number of items to be iterated over. Iterators can be created very fast and they can be combined. So you can create very effective filter functions. It's recommended to have a look in the powerful itertools and more_itertools packages to combine it with itertree

The main idea is to combine all the filtering and iterator options together before you start the final iteration (consume the iterator), which might at least end up in the expected list. By this mechanism we do at least only one unique iteration over the items and we must not do multiple typecasts and re-iterations in between even when we combine multiple filters.

If the user really wants to create a list he can easy cast the iterator by using the *list()* statement:

```
>>> myresultlist=list(root.iter_all()) #  this is quick even for huge number of items
>>> first_item=list(root.iter_all())[0] # Anyway this is much slower than:
>>> first_item=next(root.iter_all())
>>> fifth_item=list(root.iter_all())[4] # and this is much slower than:
>>> import itertools
>>> fifth_item=next(itertools.isslice(root.iter_all(),4,None))
```

As it is shown in the performance test the operation *list()* is very quick (less then 0.5 s on 1 million items (depending on you PC)). And using the index access afterwards is a very good readable code. But as shown here there are quicker solutions available on iterators only.

But we see also two downsides related to iterators:

- The StopIteration exception must be handled in case of empty iterators. To make the handling a bit easier iTree delivers in most cases an empty list if we have no match. But in some cases (e.g. filter operations) the user will get an empty iterator and not the empty list. In itree_helpers the user can find a check function for empty iterators that might help in this case: *is_iterator_empty(my_iterator)*.
- The user must also consider that an iterator can be consumed only one time. To reuse an iterator multiple times you may have a look on *itertools.tee()*.

To summarize this chapter:

We decided that the iTree methods should deliver only iterators (and not lists). This is made to give the user the possibility to utilize the whole iterator power afterwards. If he really needs a list (in most cases for index access) he can cast the iterator easy and quick via the *list()* statement. But if iTree would directly deliver lists by default we would have a performance drop in all itertree filter functions which is not acceptable from our point of view.

PYTHON MODULE INDEX

i

INDEX

Symbols

A

C

D

E

F

G

I

K

L

M

O

P

R

S

T

U

V