augmentedtree

Release 0.0a1

May 25, 2020

Contents:

augmentedtree enhances mappings and sequences (targeted for python dictionaries and lists) preserving their native behavior and access. The enhancement comes with getting values by single keys, human readable viewing, selecting and setting multiple values/items within the nested data at once, *or*-conditional selection of values/items. Also this package intends to prepare the nested data for pyQt. The augmentation provides methods and properties to be used for a *QAbstractItemModel* showing the nested data within a *QTreeView*.

CHAPTER 1

Installation

Installing the latest release using pip is recommended.

```
$ pip install augmentedtree
```

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

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

CHAPTER 2

Purpose of *augmentedtree*

The main purpose of this package is enhance nested data structure (mostly nested dictionaries) by keeping its functionality for methods working with these data types.

The targeted usage is to be able to write the following kind of code

```
# code where the nested data comes from
...

# gathering parameters
with AugmentedTree(nested_data) as tree:
   # simple selection
   first_value = tree.select("something", "here")[0]
   last_value = tree.select("something", "there")[-1]
   a_slice_of_values = tree.select("a", "l?t", "of")[3:6]

   # selection with refinement
   selection_of_values = tree.select("also/a", "lot", "of")
   narrowed_down = selection_of_values.where("this", "or", "that")[ALL_ITEMS]
    ...

if not tree.all_selections_succeeded:
    # break, exit or reacting to some value are not there
    ...

# code which is working with requested parameters
...
```

CHAPTER 3

Limitations

The augmented tree items are directly forwarding the nested data. Practically objects implementing collections.abc.Sequence and collection.abc.Mapping should work with this package. Nevertheless:

- This package was developed with using *list* and *dict*. Other classes weren't tested yet.
- Keep in mind that inserting new items directly into the nested data will mess up the augmentation. If *AnAugmentedTreeItem* should keep track of new items usage of this layer is mandatory.

The **development status** is **alpha**.

- Changes will come.
- Definition of types for fields in MappingTreeItem will be implemented.
- **Selection** items will be reworked. (SPARQL support is a desirable option.)

3.1 Basic Usage Examples

After a short example about the basic behavior of *AnAugmentedTreeItem* (*for more details*) examples will show how to

- *Select* items,
- working with these *selections*
- and *viewing* them afterwards.

3.1.1 Basic behavior of *AnAugmentedTreeItem*

An *'augmented'* dictionary or sequence will keep their native behavior, due to the goal of augmentedtree to not interfere with other python modules/packages working with native python mappings and sequences. *AnAugmentedTreeItem* enhances the nested data.

Accessing *AnAugmentedTreeItem* by index will return the nested data on basis of its *augmented path*.

[105]:
```python
from augmentedtree import AugmentedTree
nested_data = {"a": 1, "b": [1, 2, 3], "c": {"d": 3}}
atree = AugmentedTree(nested_data)
print(atree["a"], type(atree["a"]))
print(atree["b"], type(atree["b"]))
print(atree["c"], type(atree["c"]))
```

```
1 <class 'int'>
[1, 2, 3] <class 'list'>
{'d': 3} <class 'dict'>
```

Mappings and sequences can be used like before.

[106]:
```python
print(atree["a"])
print(atree["b"][1])
print(atree["c"]["d"])
```

```
1
2
3
```

3.1.2 The nested exemplary data

Within the following examples this nested data will be used.

[107]:
```python
nested_data = {
    "A1": {
        "B1": {
            "C1": {"x11": 11, "x21": 21, "x31": 31},
            "C2": {"x12": 12, "x22": 22},
            "C3": {"x13": 13},
        },
        "B2": {
            "C1": {"x11": 110, "x21": 210, "x31": 310},
            "C2": {"x12": 120, "x22": 220},
        },
        "B3": {
            "C1": {"x11": 1100, "x21": 2100, "x31": 3100}
        }
    },
    "A2": {
        "B1": {
            "C1": {"x11": 211, "x21": 221, "x31": 231},
            "C2": {"x12": 212, "x22": 222},
            "C3": {"x13": 213},
        }
    }
}
```

3.1.3 Examples on how to

Access values using select

Selecting items by using a single key.

```
[108]: atree = AugmentedTree(nested_data)
       all_x12_items = atree.select("x12")
       all_x12_items.print()
```

```
#0 12
#1 120
#2 212
```

Selecting items by using multiple keys.

Keys needs to be in order of their occurrence within the desired items *augmented path*.

```
[109]: atree = AugmentedTree(nested_data)
       all_x12_items_at_A2 = atree.select("A2", "x12")
       all_x12_items_at_A2.print()
```

```
#0 212
```

```
[110]: not_working = atree.select("x12", "A2")
       print("Returning nothing", not_working[:])
```

```
Returning nothing []
```

Selecting using UNIX wildcards

A selection using the well known and beloved UNIX filename search pattern is supported.

Using the questionmark

```
[111]: atree = AugmentedTree(nested_data)
       allitems_having_a_leading_1 = atree.select("x1?")
       allitems_having_a_leading_1.treeitems.print()
```

```
#0 /A1/B1/C1/x11
  11
#1 /A1/B1/C2/x12
  12
#2 /A1/B1/C3/x13
  13
#3 /A1/B2/C1/x11
  110
#4 /A1/B2/C2/x12
  120
#5 /A1/B3/C1/x11
  1100
#6 /A2/B1/C1/x11
  211
#7 /A2/B1/C2/x12
  212
#8 /A2/B1/C3/x13
  213
```

Using a range of numbers

Using a range of numbers

```
[112]: allitems_having_only_1_and_3 = atree.select("x[13][13]")
       allitems_having_only_1_and_3.treeitems.print()
```

```
#0 /A1/B1/C1/x11
  11
#1 /A1/B1/C1/x31
  31
#2 /A1/B1/C3/x13
  13
#3 /A1/B2/C1/x11
  110
#4 /A1/B2/C1/x31
  310
#5 /A1/B3/C1/x11
  1100
#6 /A1/B3/C1/x31
  3100
#7 /A2/B1/C1/x11
  211
#8 /A2/B1/C1/x31
  231
#9 /A2/B1/C3/x13
  213
```

Getting all unknown items of a specific item

new in release 0.2a0

Using a path delimiter with a trailing single asterisk wildcard "/*" selects only items at that level, instead of all sub items (default).

```
[113]: all_next_level_subitems_of_b2 = atree.select("B2/*")
       all_next_level_subitems_of_b2.treeitems.print()
```

```
#0 /A1/B2/C1
  C1:
    x11: 110
    x21: 210
    x31: 310

#1 /A1/B2/C2
  C2:
    x12: 120
    x22: 220
```

If the star * wildcard is used as a stand-alone path part, it will return all sub tree item from its root(s).

```
[114]: all_subitems_of_b2 = atree.select("B2", "*")
       all_subitems_of_b2.treeitems.print()
```

```
#0 /A1/B2/C1
  C1:
    x11: 110
    x21: 210
    x31: 310

#1 /A1/B2/C1/x11
  110
#2 /A1/B2/C1/x21
  210
#3 /A1/B2/C1/x31
  310
#4 /A1/B2/C2
  C2:
    x12: 120
    x22: 220

#5 /A1/B2/C2/x12
  120
#6 /A1/B2/C2/x22
  220
```

Selecting using *regular expression*

With regular expression a powerful tool for selection is available.

[115]:
```
from augmentedtree import RegularExpressionParts as REPs
allitems_having_only_1 = atree.select(REPs("x[1]{2}"))
allitems_having_only_1.treeitems.print()
```

```
#0 /A1/B1/C1/x11
  11
#1 /A1/B2/C1/x11
  110
#2 /A1/B3/C1/x11
  1100
#3 /A2/B1/C1/x11
  211
```

Select with *Or* condition

Wrapping multiple of path parts with a *tuple*, a *list*, an *UnixFilePatternPart* or a *RegularExpressionPart* makes these parts behave like an *or* condition in between them.

```
select("A", ("B", "C"), "D")
            is equal to
        A and (B or C) and D
```

[116]:
```
from augmentedtree import RegularExpressionParts as REPs
all_x11_items_of_B1_or_Y1 = atree.select(REPs("B1", "Y1"), "x11")
all_x11_items_of_B1_or_Y1.treeitems.print()
```

```
#0 /A1/B1/C1/x11
  11
#1 /A2/B1/C1/x11
  211
```

3.1.4 Working with selections

Accessing single items

[117]:
```
atree = AugmentedTree(nested_data)
all_x12_items = atree.select("x12")
second_found_value = all_x12_items[1]
print(second_found_value)
```

```
120
```

[118]:
```
second_item = all_x12_items.treeitems[1]
print(second_item)
```

```
ValueTreeItem(x12: 120)
```

Accessing slices

Selections can be accessed using slices. For a better readability of scripts *AugmentedTree.ALL_ITEMS* can be used.

[119]:
```
from augmentedtree import ALL_ITEMS

atree = AugmentedTree(nested_data)
all_x12_items = atree.select("x12")
all_values = all_x12_items[ALL_ITEMS]
print(all_values)
```

```
[12, 120, 212]
```

[120]:
```
all_except_last = all_x12_items[:-1]
print(all_except_last)
```

```
[12, 120]
```

Setting multiple values at different locations at once

[121]:
```
from copy import deepcopy
temp_nested_data = deepcopy(nested_data)
atree = AugmentedTree(temp_nested_data)

# Select items
all_x11_items = atree.select("x11")

# Set all items to a new values
all_x11_items[ALL_ITEMS] = "My key is x11."

atree.print()
```

```
{..}
  A1:
    B1:
      C1:
        x11: My key is x11.
        x21: 21
        x31: 31
      C2:
        x12: 12
        x22: 22
      C3:
        x13: 13
    B2:
      C1:
        x11: My key is x11.
        x21: 210
        x31: 310
      C2:
        x12: 120
        x22: 220
    B3:
      C1:
        x11: My key is x11.
        x21: 2100
        x31: 3100
  A2:
    B1:
      C1:
        x11: My key is x11.
        x21: 221
        x31: 231
      C2:
        x12: 212
        x22: 222
      C3:
        x13: 213
```

[122]:
```
all_x11_items[1:3] = "A value set by using slicing."

atree.print()
```

```
{..}
  A1:
    B1:
      C1:
        x11: My key is x11.
        x21: 21
        x31: 31
      C2:
        x12: 12
        x22: 22
      C3:
        x13: 13
    B2:
      C1:
        x11: A value set by using slicing.
```

(continues on next page)

(continued from previous page)

```
        x21: 210
        x31: 310
      C2:
        x12: 120
        x22: 220
    B3:
      C1:
        x11: A value set by using slicing.
        x21: 2100
        x31: 3100
  A2:
    B1:
      C1:
        x11: My key is x11.
        x21: 221
        x31: 231
      C2:
        x12: 212
        x22: 222
      C3:
        x13: 213
```

3.1.5 Viewing treeitems

AnAugmentedTreeItem will give you a different output in whether you use `print(treeitem)` or `treeitem.print()`.

The standard output resembles the nested data wrapped by *AnAugmentedTreeItem*. An exception is a ValueTreeItem in which its key (primekey) is also print out for more convenience.

[123]:
```
root_level = {
    "ridiculous-level": {
        "ludicrous-level": "They've gone into plaid."
    }
}

spacetree = AugmentedTree(root_level)
ridiculous_item = spacetree.children["ridiculous-level"]
ludicrous_item = ridiculous_item.children["ludicrous-level"]
print(spacetree)
print(ridiculous_item)
print(ludicrous_item)
```

```
AugmentedTree({'ridiculous-level': {'ludicrous-level': "They've gone into plaid."}})
MappingTreeItem({'ludicrous-level': "They've gone into plaid."})
ValueTreeItem(ludicrous-level: They've gone into plaid.)
```

Using *AnAgumentedTreeItem*'s print method gives a different output.

[124]:
```
print("# spacetree")
spacetree.print()
print("# ridiculous_treeitem")
ridiculous_item.print()
print("# ludicrous_treeitem")
ludicrous_item.print()
```

```
# spacetree
{..}
  ridiculous-level:
    ludicrous-level: They've gone into plaid.

# ridiculous_treeitem
ridiculous-level:
  ludicrous-level: They've gone into plaid.

# ludicrous_treeitem
They've gone into plaid.
```

3.1.6 Sorting tree item selections

new in release 0.2a0

The **main purpose** of the *sort*-method is to **sort selections**. If the nested data should be sorted, its more efficient to use specific packages and re-augment the sorted result. In many cases a sorted result of a selection is desired.

Tree items can be sorted using the *sort* method of *AugmentedTreeItem* or *AugmentedItemSelection*. By default the tree items are sorted in regard of their *augmented_path*.

The current implementation does **not** sort the nested data itself. It sorts the tree items. It can be compared to a reorganized view, while the original data keeps its order.

Exemplary behavior

This example shows a simple nested structure with 6 leafs.

[125]:
```
from augmentedtree import AugmentedTree

data = {
    "b": {
        "a-1": 1,
        "b-1": 2
    },
    "a": {
        "a-2": 3,
        "a-13": 4,
        "b-2": 5,
        "b-13": 6,
    }
}

a_tree = AugmentedTree(data)
a_tree.print()
```

```
{..}
  b:
    a-1: 1
    b-1: 2
  a:
    a-2: 3
    a-13: 4
    b-2: 5
```

(continues on next page)

(continued from previous page)

```
    b-13: 6
```

The tree items inherits the order of the given structure.

[126]:
```
a_tree.treeitems.print()
```

```
#0 /b
  b:
    a-1: 1
    b-1: 2

#1 /b/a-1
  1
#2 /b/b-1
  2
#3 /a
  a:
    a-2: 3
    a-13: 4
    b-2: 5
    b-13: 6

#4 /a/a-2
  3
#5 /a/a-13
  4
#6 /a/b-2
  5
#7 /a/b-13
  6
```

Sorting the tree affects the global order of the tree items, but not the local order or the nested data itself.

[127]:
```
sorting_tree = a_tree.sort()
sorting_tree.print()
```

```
{..}
  b:
    a-1: 1
    b-1: 2
  a:
    a-2: 3
    a-13: 4
    b-2: 5
    b-13: 6
```

[128]:
```
sorting_tree.treeitems.print()
```

```
#0 /a
  a:
    a-2: 3
    a-13: 4
    b-2: 5
    b-13: 6

#1 /a/a-13
```

(continues on next page)

(continued from previous page)

```
  4
#2 /a/a-2
  3
#3 /a/b-13
  6
#4 /a/b-2
  5
#5 /b
  b:
    a-1: 1
    b-1: 2

#6 /b/a-1
  1
#7 /b/b-1
  2
```

Basic usage of sorting

The main purpose is to sort selections. The default sorting order is based on the augmented path.

[129]:
```
all_bees = a_tree.select("b-*").sort()
all_bees.treeitems.print()
```

```
#0 /a/b-13
  6
#1 /a/b-2
  5
#2 /b/b-1
  2
```

A different sorting order might be desired, which can be achieved using the *Callable[[PathMapItem], int]* interface.

[130]:
```
from augmentedtree.treeitemselection import PathMapItem

def sort_by_trailing_item_number(path_map_item: PathMapItem) -> int:
    number_characters = path_map_item.primekey.split("-")[1]
    try:
        return int(number_characters)
    except TypeError:
        return 0

all_bees = a_tree.select("b-*").sort(sorting_method=sort_by_trailing_item_number)
all_bees.treeitems.print()
```

```
#0 /b/b-1
  2
#1 /a/b-2
  5
#2 /a/b-13
  6
```

3.2 Usage of *augmentedtree* - detailed examples

This chapter shows more detailed examples than within the chapter *Basic Usage*. These simplified examples are taken from existing projects and resembles the major issues *augmentedtree* was written for.

Accessing values - Example on how to retrieve nested values. - Using the *nested example data* - items are *selected* using a single key. - *Refining the selection* retrieves exact values.

Usage of Schemas - Getting a quick view on the relevant (by your definition) values. - How to *define schemas* - How the data *looks without schemas* - How the data *looks with schemas* - How schemas effect *selecting* items

Use-case of the *or*-conditional selection in combination with setting multiple values.

3.2.1 Accessing values - 'Where did I put it again?'

Did you ever wondered 'Where did i put it?'? You were about to write an analysis script and put the configuration values into a JSON dump beforehand by another preliminary process. Or you can't remember the structure of the nested output and its key names?

Nested example data

The exemplary data can be found within a json-file.

```
[1]: import json
     from dicthandling import read_from_json_file

     # load and show the nested data
     nested_data = read_from_json_file("resources/nested_data_of_examples.json", "detailed/
     ↪example-1")
     print(json.dumps(nested_data, indent="  "))
```

```
{
  "section-name": {
    "7h3-P4r7-Y0u-C4n7-R3m-3mb3r": {
      "metatype": "my-man",
      "type": "worker",
      "metadata-1": 24,
      "metadata-2": "value needed for a function",
      "name-of-this-item": "Gerry",
      "tasks": {
        "class": "TaskCollection",
        "description": "Tasks 'Gerry' should do.",
        "items": [
          {
            "metatype": "task",
            "class": "WorkerTask",
            "name": "prepare-task",
            "arg-1": "Not this one.",
            "arg-2": "Not this one either."
          },
          {
            "metatype": "task_of_gerry",
            "class": "WorkerTask",
            "name": "get-cracking",
            "arg-1": "This one you want."
          }
```

(continues on next page)

(continued from previous page)

```
        ]
      },
      "another-parameter": [
        1,
        2,
        3
      ]
    }
  },
  "etc.": "..."
}
```

Selecting specific items

In this example all items with a key *arg-1* are selected using `select(*path_parts)`. The desired value can be obtained by knowing the item's index.

[2]:
```
from augmentedtree import AugmentedTree, ALL_ITEMS

# augment the nested data
atree = AugmentedTree(nested_data)

# get a selection of all items with the key 'arg-1'
taskarg1_selection = atree.select("arg-1")

# take a look on the selected items using the explicit print method of
# the selection
taskarg1_selection.print()

# in this example get all values from the selection using a slice (ALL_ITEMS)
# equivalent to [:] (using ALL_ITEMS makes the code more understandable)
taskarg1_of_alltasks = taskarg1_selection[ALL_ITEMS]

# here it is known the desired arg-1 is at the end.
taskarg1_of_getcracking = taskarg1_of_alltasks[-1]

print("\narg-1 of 'get-cracking': {}".format(taskarg1_of_getcracking))
```

```
#0 Not this one.
#1 This one you want.

arg-1 of 'get-cracking': This one you want.
```

Refining the selection

In the prior example knowledge of the items occurrence is required to obtain the desired value. In cases where you don't know the position of the item a refinement of the selection comes handy. Exact values/items can be retrieved using `where(*path_parts)`.

[3]:
```
arg1_selection = atree.select("arg-1")
# here: using 'where' on the prior made selection returns only one value
crackpoint = arg1_selection.where("get-cracking")

crackpoint.print()
```

(continues on next page)

(continued from previous page)

```
# which can be accessed using the first index
taskarg1_of_getcracking = crackpoint[0]

print("\narg-1 of 'get-cracking': {}".format(taskarg1_of_getcracking))
```

```
#0 This one you want.

arg-1 of 'get-cracking': This one you want.
```

3.2.2 Usage of Schemas - Getting a quick view on the relevant values

With *schemas* a more semantic like behavior can be applied to the nested data. *Schemas* are defined using dictionaries and it planned to implement JSON-schemas.

Interpretation of *metadata* within this package

By using schemas values can be classified as *metadata*. In this context values are classified as metadata

- if these values are not essential for the impression of your data, therefore can be hidden from the view.
- Additional not essential values *which* can be understood as *attributes* of an entity. These will be used selecting values by *where*.
- Besides *attributes* there can be data, which is not directly related for the entity but is used for process control. E.g. an unique identifier generated at runtime.

The distinction is based on your interpretation. What do you want to tell the viewer?

Example: A *blue tennis ball* will be enough data for the majority to depict such an object. Since the default color of tennis balls is yellow, the color *blue* in that case is essential to forward this information, to make the deviation of the standard clear. *Diameter, mass, manufacturer, production date, etc.* are additional *attributes* of a tennis ball, but are needed for specific occasions only. ("Pass me the blue ACME tennis ball, which was made before 2020.02.02"). An 'runtime metadata' of a tennis ball is the store's article number for the cashier system.

Using schemas

Schemas are defined by a dictionary with specific entries. The recommend way to define a schema is by using the *construct*-method of MappingSchemaBuilder. For further explanation see the section *Schemas*.

```
schema = MappingSchemaBuilder.construct(
    identifier=("key-within-the-target-mapping", "identifier"),
    primarykey="key-which-value-is-used-as-primekey",
    primaryname="key-which-value-is-used-as-primename",
    additional_metafieldkeys=["keys", "treadend", "as", "metadata"]
}
```

In the example below three *schemas* are used for the nested data, giving the output a different meaning.

Schemas have to be explicitly defined for usage using the `use_MappingSchema_schema()` method.

```
[4]: from augmentedtree import MappingSchema, use_mappingtree_schemas, MappingSchemaBuilder

schemas_as_kwargs = read_from_json_file(
    "resources/nested_data_of_examples.json",
    "detailed/example-1-schemas"
)
schemas = MappingSchemaBuilder.construct_from_collection(schemas_as_kwargs)

# How an example schema definition looks like.
GERRY_SCHEMA = {
                MappingSchema.IDENTIFIER: ("metatype", "my-man"),
                MappingSchema.PRIMARYKEY: "type",
                MappingSchema.PRIMARYNAME: "name-of-this-item",
                MappingSchema.METAFIELDKEYS: [
                    "metatype",
                    "type",
                    "name-of-this-item",
                    "metadata-1",
                    "metadata-2"
                ]
            }

use_mappingtree_schemas(GERRY_SCHEMA, *schemas)
```

Representation without schemas

This output shows the nested data in its 'natural' occurrence.

```
[5]: tree_like_it_is = AugmentedTree(nested_data, use_schemas=False)
tree_like_it_is.print()
```

```
{..}
  section-name:
    7h3-P4r7-Y0u-C4n7-R3m-3mb3r:
      metatype: my-man
      type: worker
      metadata-1: 24
      metadata-2: value needed for a function
      name-of-this-item: Gerry
      tasks:
        class: TaskCollection
        description: Tasks 'Gerry' should do.
        items:
          0.
            metatype: task
            class: WorkerTask
            name: prepare-task
            arg-1: Not this one.
            arg-2: Not this one either.
          1.
            metatype: task_of_gerry
            class: WorkerTask
            name: get-cracking
            arg-1: This one you want.
      another-parameter: [1, 2, 3]
```

(continues on next page)

(continued from previous page)

```
  etc.: ...
```

Representation using schemas

Using *schemas* can reduce the needed vertical space and increase readability.

[6]:
```
tree_using_schemas = AugmentedTree(nested_data)
tree_using_schemas.print()
```

```
{..}
  section-name:
    worker: Gerry
      tasks: Tasks 'Gerry' should do.
        prepare-task: WorkerTask
          arg-1: Not this one.
          arg-2: Not this one either.
        get-cracking: WorkerTask
          arg-1: This one you want.
      another-parameter: [1, 2, 3]
  etc.: ...
```

Impact of schemas on selecting items

Using schemas also makes selecting items more natural to he user.

[7]:
```
atree = AugmentedTree(nested_data)

# get a selection of all 'arg-1' items with 'get-cracking' in the path
taskarg1_of_getcracking = tree_using_schemas.select("get-cracking", "arg-1")

taskarg1_of_getcracking.print()
```

```
#0 This one you want.
```

Schemas can be redefined setting the *override_existing* parameter of *use_MappingSchema_schema* to *True*. In the following example the field *arg-1* is added to the metadata and will be hidden from the standard view (compare to prior output above). Metadata can be additionally listed using the *additional_columns* parameter of the tree item's *print*-method.

[8]:
```
TASK_SCHEMA = MappingSchemaBuilder.construct(
    identifier=("class", "WorkerTask"),
    primarykey="name",
    primaryname="class",
    additional_metafieldkeys=["metatype", "arg-1"]
)

use_mappingtree_schemas(TASK_SCHEMA, override_existing=True)

tree_using_schemas = AugmentedTree(nested_data)
tree_using_schemas.print(additional_columns=["@arg-1", "arg-2"])
```

```
                                                        @arg-1                arg-2
{..}                                  '                     '
  section-name:                       '                     '
    worker: Gerry                     '                     '
      tasks: Tasks 'Gerry' should do. '                     '
        prepare-task: WorkerTask      ' Not this one.       ' Not this one either.
          arg-2: Not this one either. ' Not this one.       ' Not this one either.
        get-cracking: WorkerTask      ' This one you want.  '
      another-parameter: [1, 2, 3]    '                     '
  etc.: ...                           '                     '
```

3.2.3 Use-case of the *or*-conditional selection

The next example combines the *'usage of schemas'* for shortening the view, and *setting multiple values* at once, where a specific set of values need to be changed.

Here some script went wrong. After the 'bug' was fixed and a lot of job files has to be 'reset' to a specific configuration.

The nested data is a broadly simplified example. It consists of a list with 4 dictionaries resembling what could be simple task definitions.

```
[9]: from augmentedtree import use_mappingtree_schemas, AugmentedTree, MappingSchema, ALL_
     ↪ITEMS
     from dicthandling import read_from_json_file


     task_list = read_from_json_file("resources/nested_data_of_examples.json", address=
     ↪"detailed/tasklist")
     import json
     print(json.dumps(task_list, indent="  "))
```

```
[
  {
    "metatype": "task",
    "state": "done",
    "args": "bread",
    "task": "buy"
  },
  {
    "metatype": "task",
    "state": "done",
    "args": "bread",
    "task": "take a slice of"
  },
  {
    "metatype": "task",
    "state": "done",
    "args": "sandwich",
    "task": "prepare"
  },
  {
    "metatype": "task",
    "state": "failed",
    "args": "sandwich",
    "task": "eat"
```

(continues on next page)

(continued from previous page)

```
  }
]
```

By using schemas the meaning is be made easier to read and needed vertical space shortened.

[10]:
```
schema_parameters = read_from_json_file("resources/nested_data_of_examples.json",␣
↪address="detailed/tasklist-schemas")
schemas = MappingSchemaBuilder.construct_from_collection(schema_parameters)

use_mappingtree_schemas(*schemas)

task_tree = AugmentedTree(task_list)
task_tree.print()
```

```
[..]
  buy. bread
    state: done
  take a slice of. bread
    state: done
  prepare. sandwich
    state: done
  eat. sandwich
    state: failed
```

Now we get to part, where we want to reset 3 specific tasks, because the first task doesn't need to be repeated. After the selection using the 'or' condition for the first path part, we check if the selection returned something.

[11]:
```
tasks_to_repeat = task_tree.select(("take", "prepare", "eat"), "state")
tasks_to_repeat[ALL_ITEMS] = "to-do"

task_tree.print()
```

```
[..]
  buy. bread
    state: done
  take a slice of. bread
    state: to-do
  prepare. sandwich
    state: to-do
  eat. sandwich
    state: to-do
```

3.3 An-augmented-Tree-Item

There are 3 basic types of tree items derived from the abstract base class `AnAugmentedTreeItem`. These 3 tree items represent values, sequences and mappings.

The following example shows nested data and the 'augmented' view on it. A tree item has always a **primekey**. Using schemas this **primekey** doesn't necessarily need to be equal to the real index/key within the nested data. The **primename** is just an association for the tree item. The **primevalue** of the tree item referees to the nested data the tree item is attached to. So the **primevalue** of the *root* tree item always shows the complete nested data.

```
[1]: from augmentedtree import AugmentedTree, PRIMARYVALUE_KEY
     import json

     nested_data = [{"a1": 1}, {"a2": 2, "a3": [3, 4, 5], }]

     pretty_printed_data = json.dumps(nested_data, indent="  ")
     print(pretty_printed_data)
```

```
[
  {
    "a1": 1
  },
  {
    "a2": 2,
    "a3": [
      3,
      4,
      5
    ]
  }
]
```

Above is a pretty print using *json.dumps* showing the how the nested data looks like.

The nested data within a 3 column view (*primekey, primename*) with primevalue (the data of *AnAugmentedTreeItem*) as an additional column.

```
[2]: tree = AugmentedTree(nested_data)
     tree.print(additional_columns=[PRIMARYVALUE_KEY])
```

```
                                       Primevalue
[..]               ' [{'a1': 1}, {'a2': 2, 'a3': [3, 4, 5]}]
  0.               ' {'a1': 1}
    a1: 1          ' 1
  1.               ' {'a2': 2, 'a3': [3, 4, 5]}
    a2: 2          ' 2
    a3: [3, 4, 5]  ' [3, 4, 5]
```

3.3.1 Value (& Sequence)

A value tree item is being considered any type not being a Mapping.

Sequences will be treated as a value if they do not possess a Mapping. A SequenceTreeItem will behave like a Sequence.

3.3.2 Mappings

The items of a Mapping can be divided into 2 groups. Values which should be shown and metadata which will be hidden. There are 2 possibilities to define this behavior.

Flat Mapping item

A flat mapping item has its values and metadata within the same level.

[3]:
```
from augmentedtree import MappingSchemaBuilder, AugmentedTree, use_mappingtree_schemas

flat_pizza = {
    "type": "flat-pizza",
    "name": "Salami",
    "number": 62,
    "tomato sauce": "1 scoop",
    "cheese": "A lot",
    "salami": "6 slices",
    "allergens": [10, 17, 24]
}

schema = MappingSchemaBuilder.construct(
    identifier=("type", "flat-pizza"),
    primarykey="type",
    primaryname="name",
    additional_metafieldkeys=["allergens", "number"]
)

use_mappingtree_schemas(schema)

AugmentedTree(flat_pizza).print()
```

```
flat-pizza  Salami
  tomato sauce: 1 scoop
  cheese: A lot
  salami: 6 slices
```

Nested Mapping item

A nested mapping item has is values within a additional level. Metadata is at the root level of this item. This 'rootlevel' is skipped in the view.

[4]:
```
nested_pizza = {
    "type": "nested-pizza",
    "name": "Salami",
    "number": 62,
    "ingredients": {
        "tomato sauce": "1 scoop",
        "cheese": "A lot",
        "salami": "6 slices",
    },
    "allergens": [10, 17, 24]
}

schema = MappingSchemaBuilder.construct(
    identifier=("type", "nested-pizza"),
    primarykey="type",
    primaryname="name",
    outervalues_key="ingredients"
)

use_mappingtree_schemas(schema)

AugmentedTree(nested_pizza).print()
```

```
nested-pizza  Salami
  tomato sauce: 1 scoop
```

(continues on next page)

(continued from previous page)

```
  cheese: A lot
  salami: 6 slices
```

3.3.3 Schemas

Schemas are the essential tool to give mapping a more semantic like view. Any dictionary containing the right set of keys of MappingSchema can be used as a schema. *IDENTIFIER* of MappingSchema is mandatory. Either *METAFIELDKEYS* or *OUTERVALUES* are required additionally. *OUTERVALUES* always overrules *METAFIELDKEYS*.

All other entries are optional and do have impact on the representation within the view, which will be shown in the following examples, starting with the 'flat-pizza' from above.

Overriding default metafieldkey definition keeps the identifier visible.

[5]:
```
schema = MappingSchemaBuilder.construct(
    identifier=("type", "flat-pizza"),
    metafieldkeys=["number", "allergens"]
)
use_mappingtree_schemas(schema, override_existing=True)
AugmentedTree(flat_pizza).print()
```

```
{..}
  type: flat-pizza
  name: Salami
  tomato sauce: 1 scoop
  cheese: A lot
  salami: 6 slices
```

Adding metafieldkeys to the defaults hides the identifier.

[6]:
```
schema = MappingSchemaBuilder.construct(
    identifier=("type", "flat-pizza"),
    additional_metafieldkeys=["number", "allergens"]
)
use_mappingtree_schemas(schema, override_existing=True)
AugmentedTree(flat_pizza).print()
```

```
{..}
  name: Salami
  tomato sauce: 1 scoop
  cheese: A lot
  salami: 6 slices
```

Supplied primarykey and primaryname are default metadatakeys hiding both items automatically.

[7]:
```
schema = MappingSchemaBuilder.construct(
    identifier=("type", "flat-pizza"),
    primarykey="type",
    primaryname="name",
    additional_metafieldkeys=["number", "allergens"]
)
use_mappingtree_schemas(schema, override_existing=True)
AugmentedTree(flat_pizza).print()
```

```
flat-pizza  Salami
  tomato sauce: 1 scoop
  cheese: A lot
  salami: 6 slices
```

Outervalues overrules metadatakeys. If an item of a mapping is defined to represent the real values (or children) of this collection, then automatically all root level items of this mapping are rendered to metadata.

```
[8]: schema = MappingSchemaBuilder.construct(
         identifier=("type", "nested-pizza"),
         metafieldkeys=["these", "are", "now", "irrelevant"],
         outervalues_key="ingredients"
     )
     use_mappingtree_schemas(schema, override_existing=True)
     AugmentedTree(nested_pizza).print(additional_columns=["@name", "@allergens"])
```

```
                          @name     @allergens
{..}                    ' Salami ' [10, 17, 24]
  tomato sauce: 1 scoop ' Salami ' [10, 17, 24]
  cheese: A lot         ' Salami ' [10, 17, 24]
  salami: 6 slices      ' Salami ' [10, 17, 24]
```

3.3.4 Paths within the augmentation

AnAugmentedTreeItem has 2 paths. It's *real path* is the location within the original nested datastructure the tree items are attached to. The tree item's *augmented path* depicts the location within the augmentation changed by *schemas*.

```
[9]: from dicthandling import read_from_json_file

     # get nested data and schema for the example
     nested_data = read_from_json_file("resources/nested_data_of_examples.json", "AATI/
     ↪path_example")

     tree_without = AugmentedTree(nested_data, use_schemas=False)
     tree_without.print()
```

```
{..}
  The story:
    type: example
    name: W.
    key: Richard
    items:
      0.
        type: example
        name: steed
        key: mighty
        items:
          0.
            type: example
            name: along
            key: rides
            items:
              into: the sunset
```

From the data above the last item with the *primekey* into is selected, the item's *TreePath* retrieved and both *real path* as well *augmented path* shown. Since no schemas were applied, both are identical.

```
[10]: real_and_augmented_path_is_identical = tree_without.select("into")
      items_treepath = real_and_augmented_path_is_identical.paths[0]

      print("     real path:", items_treepath.real_path)
      print("augmented path:", items_treepath.augmented_path)
```

```
     real path: /The story/items/0/items/0/items/into
augmented path: /The story/items/0/items/0/items/into
```

Now the whole process will be repeated, but this time a schema is applied to the nested data, resulting in a different view off it. Both paths differs in this example, due to the *schema*.

The exemplary data can be found within a json-file.

```
[11]: schema_kwargs = read_from_json_file("resources/nested_data_of_examples.json", "AATI/
      ↪path_example-schema")
      example_schema = MappingSchemaBuilder.construct(**schema_kwargs)

      use_mappingtree_schemas(example_schema)

      tree_with_schema = AugmentedTree(nested_data)
      tree_with_schema.print()

      augmented_path_differs_from_real = tree_with_schema.select("into")
      items_treepath = augmented_path_differs_from_real.paths[0]

      print("     real path:", items_treepath.real_path)
      print("augmented path:", items_treepath.augmented_path)
```

```
{..}
  Richard: W.
    mighty: steed
      rides: along
        into: the sunset

     real path: /The story/items/0/items/0/items/into
augmented path: /Richard/mighty/rides/into
```

```
[12]: # Using the origin data and navigating through it
      print(tree_with_schema["The story"]["items"][0]["items"][0]["items"]["into"])

      # Using the augmented path
      print(tree_with_schema["Richard/mighty/rides/into"])
```

```
the sunset
the sunset
```

3.4 package

3.4.1 Augmented tree items

Abstract Base Classes

class `augmentedtree.`**`AnAugmentedTreeItem`**

Bases: `abc.ABC`

It is mandatory to implement this abstract basic class for any kind of augmented tree item. This class defines the minimal, necessary set of properties and methods for a tree item within a two column tree view (like it is presented using *print_atree*).

class `augmentedtree.`**`AnAugmentedCollection`**

Bases: `augmentedtree.abstractbaseclasses.AnAugmentedTreeItem`

In addition to AnAugmentedTreeItem a tree item resembling a sequence or mapping has to implement *__getitem__*, *__setitem__* and *outervalues*.

Base tree item classes

class `augmentedtree.`**`ATreeItem`**(*real_key=None*, ***kwargs*)

Bases: `augmentedtree.abstractbaseclasses.AnAugmentedTreeItem`

ATreeItem set the entry point for subclassing *AnAugmentedTreeItem* for usage with *augment_datastructure*. It also implements properties and methods for convenience. Properties and methods which has to be overwritten will raise NotImplementedErrors.

Raises `TypeError` – This class cannot be instantiated directly. It has to be sub-classed.

`children`

Access to the children (tree items) this tree item possess. Also obligatory for a brasic QT implementation.

Returns

Sequence Sequence of *AnAugmentedTreeItem*

`has_primekey`(*key: Union[int, str]*) → bool

Returns it the tree item has a child with the requested key. Basic implementation for QT, where it's called hasKey().

Parameters **`key`** (*`Union[int, str]`*) – Requested key.

Returns bool

`primekey`

Represents the key this tree item is associated within its parent container. Also it is the first columns (left) value within a 2-column view (like when using *print_atree()*).

Returns Union[int, str]

`primename`

Represents the value this tree item is associated with. Also it is the second columns (right) value within a 2-column view (like when using *print_atree()*).

Returns Any

`primevalue`

The 'real' value/nested data this tree item is representing.

Returns Any

`real_key`

The real key within the nested data.

Returns Union[int, str]

class augmentedtree.**ACollectionTreeItem**(*primarykey: Union[augmentedtree.core.KeyLink, str, int] = None, primaryname: Union[augmentedtree.core.KeyLink, str, int] = None, primaryvalue: Union[Sequence[T_co], collections.abc.Mapping] = None, real_key: str = None, **kwargs*)

Bases: `augmentedtree.abstractbaseclasses.AnAugmentedCollection`, `augmentedtree.treeitems.ATreeItem`

outervalues

Returns this tree items origin nested data, which is considered to represent this tree items values. These doesn't need to be identical to *primevalue*, but *primevalue* always contains the values visible to the outside.

Returns Any

Tree item classes

class augmentedtree.**ValueTreeItem**(*primarykey=None*, ***metadata*)

Bases: `augmentedtree.treeitems.ATreeItem`

class augmentedtree.**SequenceTreeItem**(*primarykey=None*, *primaryname=None*, *primaryvalue=None*, *real_key=None*, ***kwargs*)

Bases: `collections.abc.MutableSequence`, `augmentedtree.treeitems.ACollectionTreeItem`

class augmentedtree.**MappingTreeItem**(*primarykey=None*, *primaryname=None*, *primaryvalue=None*, *outervaluekey=None*, *metadatakeys=None*, *real_key=None*, *field_types: Dict[str*, *Callable] = None*, *meta_attributes=None*, *keypairs: dict = None*)

Bases: `collections.abc.Mapping`, `augmentedtree.treeitems.ACollectionTreeItem`

class augmentedtree.**AugmentedTree**(*data: Union[augmentedtree.abstractbaseclasses.AnAugmentedCollection, Mapping[KT, VT_co], Sequence[T_co]], use_schemas: bool = True, pathmap: augmentedtree.treeitemselection.PathMap = None*)

Bases: `augmentedtree.abstractbaseclasses.AnAugmentedTreeItem`

This class is the recommended entry for augmenting nested data.

Parameters

- **data** (*Union[Mapping, Sequence]*) – Augments this given data.
- **use_schemas** (*bool*) – As default registered schemas are used. If turned to *false* this tree will represent the pure data structure.

augmentedtree.**augment_datastructure**(*nested_data: Union[Sequence[T_co], Mapping[KT, VT_co]], parent: augmentedtree.abstractbaseclasses.AnAugmentedCollection = None, augmentclasses: Dict[str, augmentedtree.abstractbaseclasses.AnAugmentedTreeItem] = None, use_schemas: bool = True*)

Augments nested data with *AnAugmentedTreeItem*.

Parameters

- **nested_data** (*Union[Sequence, Mapping]*) – nested data to be augmented with *AnAugmentedTreeItem*.
- **parent** (*AnAugmentedCollection, optional*) – parent of the given *nested_data*

- **augmentclasses** (*Dict[str, AnAugmentedTreeItem], optional*) – *AnAugmentedTreeItem*-classes to be used for augmentation
- **use_schemas** (*bool, optional*) – Defines whether schemas should be used or not. Default = *True*; schemas are used.

Returns AnAugmentedTreeItem

`augmentedtree.`**`print_atree`**(*treeitem: augmentedtree.abstractbaseclasses.AnAugmentedTreeItem, additional_columns: List[T] = None, show_hidden=False, indent=' ', prefix=''*)

Pretty prints a tree in a simple manner.

Notes

- if the item is within a Sequence the separator between primekey and primename will be a dot '.'; else a colon ':'
- by using schemas the default indexing by integers of Sequences can be changed to key-names of a Mapping.

Parameters

- **treeitem** (AnAugmentedTreeItem) – Tree item to be printed.
- **additional_columns** (*str*) – Additional columns which should be shown.
- **show_hidden** (*bool*) – If *True* leading underline keys will be shown. Default = *False*
- **indent** (*str*) – Indentation characters which will be used.
- **prefix** (*str*) – Additional string with which line begins.

class `augmentedtree.`**`LeafType`**

An enumeration.

3.4.2 Selecting values

By default the methods `select()` and `where()` of *AugmentedTree* and *AugmentedItemSelection* interprets any given path component as an UNIX file pattern. To use regular expressions instead, these path components can be wrapped with *RegularExpressionPart*.

class `augmentedtree.`**`RegularExpressionPart`**

class `augmentedtree.`**`AugmentedItemSelection`**(*data: Union[augmentedtree.abstractbaseclasses.AnAugmentedCollecti Mapping[KT, VT_co], Sequence[T_co]], use_schemas: bool = True, pathmap: Optional[augmentedtree.treeitemselection.PathMap] = None*)

Bases: `augmentedtree.tree.AugmentedTree`

3.4.3 Enhancement of Mappings by *schemas*

`augmentedtree.`**`use_mappingtree_schema`**(*schema: Dict[KT, VT], override_existing: bool = False*)

Registers a (JSON-)schema for a *MappingTreeItem* for global use.

Raises `ValueError` – If a schema with the same identifier is already registered.

Parameters

- **schema** (*Dict*) – (JSON-)schema to be registered.
- **override_existing** (*bool*) – If *True* and existing registered schema with the same id will be overridden.

augmentedtree.**use_mappingtree_schemas**(**schemas*, *override_existing: bool = False*)
Registers (JSON-)schemas for a *MappingTreeItem* for global use.

Raises `ValueError` – If a schema with the same identifier is already registered.

Parameters

- **schemas** (*List[Dict]*) – (JSON-)schemas to be registered.
- **override_existing** (*bool*) – If *True* and existing registered schema with the same id will be overridden.

class augmentedtree.**MappingSchema**

PRIMARYKEY = 'atree_primekey'
Defines the field from which value should be used as *primekey* of the *MappingTreeItem*.

PRIMARYNAME = 'atree_primename'
Defines the field from which value should be used as *primename* of the *MappingTreeItem*.

OUTERVALUES = 'atree_outervalues'
Defines the field from which value should be used as *primename* of the *MappingTreeItem*.

METAFIELDKEYS = 'atree_metafieldkeys'
Defines the keys of a Mapping item which should be treated as 'metadata' of this item. 'metadata' is hidden within the augmented default view.

IDENTIFIER = 'atree_mappingschema'
This field defines the unique schema identifier by an tuple of (key, value), which has to be found within the Mapping item.

Optionally if the field only contains a string, it is assumed the Mapping item contains this MappingTree.SCHEMA_IDENTIFIER as a key of an value with a unique name.

META_ATTRIBUTES = 'atree.meta_attributes'

class augmentedtree.**MappingSchemaBuilder**

static construct(*identifier: Union[str, Tuple[str, str]], primarykey: Optional[str] = None, primaryname: Optional[str] = None, outervalues_key: Optional[str] = None, metafieldkeys: Optional[List[str]] = None, additional_metafieldkeys: Optional[List[str]] = None, meta_attributes: Optional[List[str]] = None*) → dict
Construct a schema for MappingTreeItems.

Notes

- If identifier is a single string the mapping to be used by this schema needs a field with the key *MappingSchema.IDENTIFIER*.
- A mapping item can use *outervalues* or *metafieldkeys* therefore *outervalues_key* always suppress *metafieldkeys* and *additional_metafieldkeys*.
- The identifiers resulting key, *primarykey* and *primaryname* are default *metafieldkeys* if supplied. The *metafieldkeys* overrides the default behavior.

- With *additional_metafieldkeys* additional keys can be defined.

Parameters

- **identifier** (*Union[str, Tuple[str, str]]*) – A single string or a tuple/list with 2 strings defines an identifier.
- **primarykey** (*Optional[str]*) – Defines which key of the mapping should be used as the tree items *primekey*.
- **primaryname** (*Optional[str]*) – Defines which key of the mapping should be used as the tree items *primename*.
- **outervalues_key** (*Optional[str]*) – Defines the key, which contains the tree items children/values resulting in a 'nested-mapping' tree item.
- **metafieldkeys** (*Optional[List[str]]*) – Defines the keys, which will be considered as metadata. All other entries within the mapping will be considered as a child/value.
- **additional_metafieldkeys** – Additional keys to *metafieldkeys*.
- **meta_attributes** (*Optional[List[str]]*) – Defines which values will be used as meta attributes for selection via the *where* method.

Returns A schema for MappingTreeItem(s).

Return type dict

3.4.4 Tree path related

class augmentedtree.core.**TreePath**(*real_path: Union[str, List[str]] = ”, augmented_path: Union[str, List[str]] = ”, meta_attributes: List[T] = None*)

join(*real_path: Union[str, List[str]] = ”, augmented_path: Union[str, List[str]] = ”, meta_attributes: List[T] = None*) → augmentedtree.core.TreePath
Joins the path parts and returns a new AugmentedTreePath.

Notes

If parameter *treepath* is given, all parameters are overriden by it.

Parameters

- **real_path** (*Union[str, List[str]]*) – Real path within the nested data structure of the augmented tree items.
- **augmented_path** (*Union[str, List[str]]*) – Path defined by the augmentation; if no schemas are used identical to *real_path*
- **meta_attributes** (*List*) – Associations of this path part.

Returns Path within augmented tree.

Return type *TreePath*

augmentedtree.core.**normalize_path_of_tree**(**treepath*)
Normalized a path (str) or parts of a path (List[str]) to '/a/path/like/this'.

Parameters ***treepath** – A single tree path part or multiple tree path parts.

Returns Normalized path '/like/this/example'.

Return type str

```
>>> normalize_path_of_tree("//to/many/delimiters///everywhere//")
'/to/many/delimiters/everywhere'
>>> normalize_path_of_tree("missing/front/delimiter")
'/missing/front/delimiter'
>>> normalize_path_of_tree("/this/path/is/correct")
'/this/path/is/correct'
>>> normalize_path_of_tree("/unwanted/delimiter/at/the/end/")
'/unwanted/delimiter/at/the/end'
>>> normalize_path_of_tree(None)
''
>>> normalize_path_of_tree([])
''
```

```
>>> normalize_path_of_tree("//to", "/many/delimiters//", "/everywhere//")
'/to/many/delimiters/everywhere'
>>> normalize_path_of_tree("missing", "front/delimiter")
'/missing/front/delimiter'
>>> normalize_path_of_tree("this", "path", "is/correct")
'/this/path/is/correct'
>>> normalize_path_of_tree("unwanted", "delimiter", "at/the/end/")
'/unwanted/delimiter/at/the/end'
>>> normalize_path_of_tree("invalid", "type", [], "within/the/path")
'/invalid/type/within/the/path'
>>> normalize_path_of_tree(None)
''
>>> normalize_path_of_tree([])
''
```

Index