Fagus

Release 1.1.2

Lukas Neuenschwander

Aug 21, 2023

CONTENTS:

CHAPTER

ONE

ISC LICENSE

CHAPTER

TWO

README

These days most data is converted to and from `json` and `yaml` while it is sent back and forth to and from API's. Often this data is deeply nested. `Fagus` is a Python-library that makes it easier to work with nested dicts and lists. It allows you to traverse and edit these tree-objects with simple function calls that handle the most common errors and exceptions internally. The name fagus is actually the latin name for the genus of beech-trees.

2.1 Code and tests ready, documentation still WORK IN PROGRESS

This documentation is still Work in Progress. I have some more ideas for features, but most of the coding is done. The code is tested as good as possible, but of course there still might be bugs as this library has just been released. Just report them so we get them away ;). Even though this README is not done yet, you should be able to use most of the functions based on the docstrings and some trial and error. Just ask questions here if sth is unclear. The documentation will be filled in and completed as soon as possible.

HAVE FUN!

2.2 Table of contents

2.3 Basic principles

2.3.1 Introduction – What it solves

Imagine you want to fetch values from a nested dict as shown below:

```
>>> a = {"a1": {"b1": {"c1": 2}, "b2": 4}, "a2": {"d1": 6}}
>>> a["a1"]["b1"]["c1"]  # prints 2, so far so good
2
>>> a["a1"]["b3"]["c2"]  # fails, because b3 doesn't exist
Traceback (most recent call last):
  ...
KeyError: 'b3'
```

The problem is that the consecutive square brackets fail if one of the nodes doesn't exist. There are ways around, like writing `a.get("a1", {}).get("b3", {}).get("c2")` or surrounding each of these statements with `try-except`, but both are hard to maintain and verbose. Below you can see how `Fagus` can help to resolve this:

```
>>> from fagus import Fagus
>>> print(Fagus.get(a, ("a1", "b3", "c2")))  # None, as this key doesn't exist in a
None
```

As you can see, now only one function call is needed to fetch the value from `a`. If one of the keys doesn't exist, a default value is returned. In this case no default value was specified, so `None` is returned.

The whole `Fagus` library is built around these principles. It provides:

- **Simple functions**: replacing tedious code that is hard to maintain and error prone
- **Few exceptions**: Rather than raising a lot of exceptions, `Fagus` does what is most likely the programmer's intention.

2.3.2 The path-parameter

`Fagus` is built around the concept of a Mapping or dict, where there are keys that are used to refer to values. For lists, the indices are used as keys. In opposition to a simple dict, in `Fagus` the key can consist of multiple values – one for each layer.

```
>>> a = [5, {6: ["b", 4, {"c": "v1"}]}, ["e", {"fg": "v2"}]]
>>> Fagus.get(a, (1, 6, -1, "c"))
'v1'
>>> Fagus.get(a, "2 -1 fg")
'v2'
```

- **Line 3**: The path-parameter is the tuple in the second argument of the get-function. The first and third element in that tuple are list-indices, whereas the second and fourth element are dict-keys.
- **Line 5**: In many cases, the dict-keys that are traversed are strings. For convenience, it's also possible to provide the whole path-parameter as one string that is split up into the different keys. In the example above, `" "` is used to split the path-string, this can be customized using the *`path_split`* *FagusOption*.

2.3.3 Static and instance usage

All functions in `Fagus` can be used statically, or on a `Fagus`-instance, so the following two calls of *get()* give the same result:

```
1 >>> a = [5, {6: ["b", 4, {"c": "v1"}]}, ["e", {"fg": "v2"}]]
2 >>> Fagus.get(a, "2 0")
3 'e'
4 >>> b = Fagus(a)
5 >>> b.get("2 0")
6 'e'
```

The first call of *get()* in line 3 is static, as we have seen before. No `Fagus` instance is required, the object `a` is just passed as the first parameter. In line 5, `b` is created as a `Fagus`-instance – calling *get()* on `b` also yields `e`.

While it's not necessary to instantiate `Fagus`, there are some neat shortcuts that are only available to `Fagus`-instances:

```
1 >>> a = Fagus()
2 >>> a["x y z"] = 6  # a = {"x": {"y": {"z": 6}}}
3 >>> a.x  # returns the whole subnode at a["x"]
4 {'y': {'z': 6}}
5 >>> del a[("x", "y", "z")]  # Delete the z-subnode in a["x y z"]
6 >>> a()
7 {'x': {'y': {}}}
```

- **Square bracket notation**: On `Fagus`-instances, the square-bracket notation can be used for easier access of data if no further customization is needed. Line 3 is equivalent to `a.set(6, "x y z")`. It can be used for getting, setting and deleting items (line 6).
- **Dot notation**: The dot-notation is activated for setting, getting and deleting items as well (line 4). It can be used to access `str`-keys in `dict`s and `list`-indices, the index must then be preceded with an underscore due to Python naming limitations (`a._4`). This can be further customized using the *path_split* *FagusOption*.

`Fagus` is a wrapper-class around a tree of `dict`- or `list`-nodes. To get back the root-object inside the instance, use `()` to call the object – this is shown in line 7. Alternatively you can get the root-object through `.root`.

2.3.4 Fagus options

There are several parameters used across many functions in `Fagus` steering the behaviour of that function. Often, similar behaviour is intended across a whole application or parts of it, and this is where options come in handy allowing to only specify these parameters once instead of each time a function is called.

One example of a `Fagus`-option is *default*. This option contains the value that is returned e.g. in *get()* if a *path* doesn't exist, see *Introduction*, code block two for an example of *default*.

There are four levels at which an option can be set, where the higher levels take precedence over the lower levels:

The four levels of `Fagus`-options:

1. **Default**: If no other level is specified, the hardcoded default for that option is used.
2. **Class**: If an option is set at class level (i.e. `Fagus.option`), it applies to all function calls and all instances where level one and two of that option aren't defined. Options at this level apply for the whole file `Fagus` has been imported in.
3. **Instance**: If an option is set for an instance, it will apply to all function calls at that instance where the option wasn't overriden by an argument.

4. **Argument**: The highest level - if an option is specified directly as an argument to a function, that value takes precedence over all other levels.

Below is an example of how the different levels take precedence over one another:

```
1  >>> a = Fagus({"a": 1})
2  >>> print(a.get("b"))  # b does not exist in a - default is None by default
3  None
4  >>> Fagus.default = "class"  # Overriding default at class level (level 2)
5  >>> a.get("b")  # now 'class' is returned, as None was overridden
6  'class'
7  >>> a.default = 'instance'  # setting the default option at instance level (level 3)
8  >>> a.get("b")  # for a default is set to 'instance' -- return 'instance'
9  'instance'
10 >>> b = Fagus({"a": 1})
11 >>> b.get("b")  # for b, line 7 doesn't apply -- line 5 still applies
12 'class'
13 >>> del Fagus.default  # deleting an option resets it to its default
14 >>> print(b.get("b"))  # for default, the default is None
15 None
16 >>> a.get("b", default='arg')  # passing an option as a parameter always wins
17 'arg'
```

All `Fagus`-options at level three can be set in the constructor of `Fagus`, so they don't have to be set one by one like in line 8. You can also use `options()` on an instance or on the `Fagus`-class to set several options in one line, or get all the options that apply to an instance.

Some `Fagus`-functions return child-`Fagus`-objects in their result. These child-objects inherit the options at level three from their parent.

The remaining part of this section explains the `FagusOption`s one by one.

default

- **Default**: `None`
- **Type**: `Any`

This value is returned if the requested *`path`* does not exist, for example in *`get()`*.

```
1  >>> from fagus import Fagus
2  >>> a = {"b": 3}
3  >>> Fagus.get(a, "b", default=8)  # return 3, as "b" exists
4  3
5  >>> Fagus.get(a, "q", default=8)  # return default 8, as "q" does not exist
6  8
7  >>> print(Fagus.get(a, "q"))  # "q" does not exist -- return None being default if it␣
   ↪hasn't been specified as arg
8  None
```

default_node_type

- **Default**: `"d"`
- **Type**: `str`
- **Allowed values**: `"d"` and `"l"`

Can be either `"d"` for `dict` or `"l"` for `list`. A new node of this type is created if it's not specified clearly what other type that node shall have. It is used e.g. when Fagus is instanciated with an empty constructor:

```
>>> Fagus.default_node_type = "l"
>>> a = Fagus()
>>> a()  # the root node of a is an empty list as this was set in line 2
[]
>>> del Fagus.default_node_type
>>> b = Fagus()
>>> b()  # the root node of b is a dict (default for default_node_type)
{}
>>> c = Fagus([])
>>> c()  # the root node of c is now also a list
[]
```

More information about how `default_node_type` is used when new nodes need to be generated can be found in *Basic principles for modifying the tree* and the documentation of the `FagusOption` ***node_types***.

if_

- **Default**: `_None`, meaning that the value is not checked
- **Type**: Any

This option can be used to verify values before they're inserted into the `Fagus`-object. Generating configuration-files, default values can often be omitted whereas special settings shall be included, `if_` can be used to do this without an extra if-statement.

```
>>> a = Fagus(if_=True)  # the only allowed value for set is now True
>>> a.v1 = True
>>> a()  # v1 was set, because it was True (as requested in line 1)
{'v1': True}
>>> a.v2 = None
>>> a()  # note that v2 has not been set as it was not True
{'v1': True}
>>> a.set(6, "v2", if_=(4, 5, 6))  # 6 was set as it was in (4, 5, 6)
{'v1': True, 'v2': 6}
>>> a.set("", "v3", if_=bool)  # v3 is not set because bool("") is False
{'v1': True, 'v2': 6}
```

Possible ways to specify `if_`:

- **Single value**: This is shown in line 1 – the only values that can now be set is `True`, anything else is not accepted.
- **List of values**: You can also specify any `Iterable` (e.g. a `list`) with multiple values – the values that can be set must be one of the values in the `list` (line 8).
- **Callable**: You can also pass a callable object or a function (lambda) – the result of that call determines whether the value is set (line 10).

iter_fill

- **Default**: `_None`, meaning that `iter_fill` is inactive
- **Type**: Any

This option is used to get a constant number of items in the iterator while iterating over a `Fagus`-object, see *here* for more about iteration in `Fagus`. The example below shows what happens by default when iterating over a `Fagus`-object where the leaf-nodes are at different depths:

```
1  >>> a = list(Fagus.iter({"a": {"b": 2}, "c": 4}, 1))
2  >>> a
3  [('a', 'b', 2), ('c', 4)]
4  >>> for x, y, z in a:
5  ...     print(x, y, z)
6  Traceback (most recent call last):
7  ...
8  ValueError: not enough values to unpack (expected 3, got 2)
9  >>> a = list(Fagus.iter({"a": {"b": 2}, "c": 4}, 1, iter_fill=None))
10 >>> a
11 [('a', 'b', 2), ('c', 4, None)]
12 >>> for x, y, z in a:
13 ...     print(x, y, z)
14 a b 2
15 c 4 None
```

In line 3, we see that the first tuple has three items, and the second only two. When this is run in a loop that always expects three values to unpack, it fails (line 4-8). That problem is solved in line 9 by using `iter_fill`, which fills up the shorter tuples with the value that was specified for `iter_fill`, here `None`. With that in place, the loop in line 12-15 runs through without raising an error. Note that `max_depth` has to be specified for `Fagus` to know how many items to fill up to.

iter_nodes

- **Default**: `False`
- **Type**: `bool`

This option is used to get references to the traversed nodes while iterating on a `Fagus`-object, see *here* for more about iteration in `Fagus`. Below is an example of what this means:

```
1 >>> list(Fagus.iter({"a": {"b": 2}, "c": 4}, 1))
2 [('a', 'b', 2), ('c', 4)]
3 >>> list(Fagus.iter({"a": {"b": 2}, "c": 4}, iter_nodes=True))
4 [({'a': {'b': 2}, 'c': 4}, 'a', {'b': 2}, 'b', 2), ({'a': {'b': 2}, 'c': 4}, 'c', 4)]
```

As you can see, the node itself is included as the first element in both tuples. In the first tuple, we also find the subnode `{"b": 2}` as the third element. In line 2, the tuples are filled after this scheme: `key1, key2, key3, ..., value`. In line 4, we additionally get the nodes, so it is `root-node, key1, node, key2, node2, key3, ..., value`.

Sometimes in loops it can be helpful to actually have access to the whole node containing other relevant information. This can be especially useful combined with *`skip()`*.

list_insert

- **Default**: `INF` (infinity, defined as `sys.maxsize`, the max value of an `int` in Python)
- **Type**: `int`

By default, `list`-nodes are traversed in Fagus when new items are inserted. New `list`-nodes are only created if necessary. Consider the following example:

```
>>> a = Fagus([0, [3, 4, [5, 6], 2]])
>>> a.set("insert_1", (1, 2))
[0, [3, 4, 'insert_1', 2]]
```

The list `[5, 6]` is overridden with the new value `"insert_1"`. In some cases it is desirable to insert a new value into one of the lists rather than just overwriting the existing value. This is where `list_insert` comes into the picture. For some background of how `list`-indices work in `Fagus`, you can check out *this section*.

```
1 >>> a = Fagus([0, [3, 4, [5, 6], 2]], default_node_type="l")
2 >>> a.set("insert_2", (1, 2), list_insert=1)  # [5, 6] was not overridden here,␣
  ↪insert_2 is inserted before
3 [0, [3, 4, 'insert_2', [5, 6], 2]]
4 >>> a.set("insert_3", (1, 2), list_insert=0)  # here, insert_3 is inserted at the␣
  ↪base level 0, again without overriding
5 [0, ['insert_3'], [3, 4, 'insert_2', [5, 6], 2]]
```

The parameter `list_insert` defines at which depth a new element should be inserted into the list. In line 2, `list_insert` is set to one, so `"insert_2"` is inserted in position two in the list at index 1 in the `Fagus`-object. In line 4, the new element is inserted in the base-list at depth zero in the `Fagus`-object. As another index is defined in `path` (2), another list is created before `"insert_3` is inserted.

```
1 >>> a = Fagus({2: {1: 4, 3: [4, 6]}, "a": "b"})
2 >>> a.set("insert_4", (2, 3, 1), list_insert=1)
3 {2: {1: 4, 3: [4, 'insert_4', 6]}, 'a': 'b'}
```

In this last example, there is no list to be traversed at depth one. In that case, the insertion of `insert_4` is performed in the first list that is traversed above the indicated `list_insert`-depth (here one), which is at depth two.

node_types

- **Default**: `""`
- **Type**: `str`
- **Allowed values**: Any string only containing the characters `"d"`, `"l"` and `" "`

This parameter is used to precisely specify which types the new nodes to create when inserting a value at *`path`* shall have. There are defined in three possible ways: `"l"` for `list`, `"d"` for `dict` or `" "` for "don't care". Don't care means that if the node exists, its type will be preserved if possible, however if a new node needs to be created because it doesn't exist, *`default_node_type`* will be used if possible. The examples below will make it more clear how this works. For an overview, also check the *basic principles for modifying the tree*.

Example one: creating new nodes inside an empty object:

```
>>> a = Fagus()
>>> a()  # a is a dict, as default_node_type by default generates a dict
{}
>>> a.set(False, ("a", 0, 0), node_types="dl")
{'a': {0: [False]}}
```

The root node, in the case above a `dict`, can't be changed, so `node_types` only affects the nodes that resign within the root node. Therefore, `node_types` is only defined for the second until last key in *path*. For the second key in `path`, here 0, it is defined in `node_types` that it should be a `dict`, therefore a `dict` is created. In that `dict`, a `list` is inserted at key 0 as the second letter in `node_types` is `"l"`, and finally `False` is inserted into that `list.

Example two: clearly defined where to put lists and dicts at each level

```
>>> a = Fagus({3: [[4, {5: "c"}], {"a": "q"}]})
>>> a.set(True, (3, 0, 7, 4), node_types="ldl")
{3: [{7: [True]}, {'a': 'q'}]}
```

In this case, there already are nodes at the base of the position `path` is pointing to. The first key in `path`, 3, is traversed. For the second key in `path`, here 0, it is defined in `node_types` that it should be a `list` (`"l"`), and in this case it actually is a list. The third key in `path` is 7, and in `node_types` it is defined that there should be a `dict` at this level. Therefore, the `list [4, {5: "c"}]` is overwritten with a new `dict` with the key 7. The forth and last element in `path` is 4, and in `node_types` it is defined that this node shall be a `list` again. The value `True` is then placed inside that `list`.

Example three: "don't care" and other special cases:

```
1 >>> a = Fagus(default_node_type="l")
2 >>> a.set(True, (3, "a", "6"))
3 [{'a': [True]}]
4 >>> a.set(None, (1, 5), "ddddddddd")
5 [{'a': [True]}, {5: None}]
6 >>> a.set(False, (1, 1, 1, 1), node_types=" d")
7 [{'a': [True]}, {5: None, 1: {1: [False]}}]
```

The first example in line two shows what happens if `node_types` is has not been defined. In that case, all the new nodes that are to be created are interpreted as "don't care", which means that if possible, new nodes of the type `default_node_type` are created. Here, `default_node_type` is `"l"` (`list`). There is no meaningful easy way to create an `int`-`list`-index from `"a"`, therefore a `dict` is inserted at `"a"`. However, it is possible to create a list index from `"6"` by using `str()`, therefore a `list` is created at key `"a"`, in which `True` finally is inserted.

The second example in line four shows what happens if `node_types` is defined for more than the length of `path`. It's actually no problem to do that, the remaining part of `node_types` is just ignored. The third example in line six shows what happens if `node_types` only is partly defined, in this case it is only defined to be "don't care" for the second key in `path` and `"d"` for the third key in `path`, but not for the last element. For all the keys in `path` where `node_types` is undefined, it is treated as `"don't care"` when new nodes are created.

path_split

- **Default**: `" "`
- **Type**: `str`

The keys needed to traverse a `Fagus`-object for getting or setting a value are passed as a `tuple` or `list` (line 2). `path_split` allows to alternatively specify all the keys in a single string, split by `path_split` (line 4). As shown in line 4, list indices can be specified in the path-string, they are automatically converted back to `int`.

```
1 >>> a = Fagus({"a": {"b": [True, "q"]}})
2 >>> a[("a", "b", 0)]
3 True
4 >>> a["a b 0"]
5 True
```

By default, `path_split` is a single space `" "`, but any other string can be used as a split character. If path string is set to `"_"`, the dot-notation can be used to get or set a node deeply inside a `Fagus`-object.

```
>>> a = Fagus(path_split="_")
>>> a.a_c_1 = 4  # {"a": {"c": {"1": 4}}}
>>> a = Fagus(path_split="_", default_node_type="l")
>>> a._0_2 = 6  # [[6]], note that the str after . is prefixed with a _ for a list index
>>> a = Fagus(path_split="__")
>>> a.example_index__another_index = "q"  # {"example_index": {"another_index": "q"}}
```

2.4 Modifying the tree

`Fagus` does not only allow to easily retrieve elements deeply inside a tree of nested `dict`- and `list`-nodes using *get()*. The tree can also be modified using the different functions shown below. Make sure to read *set()* first as its basic principles apply to all the other modifying functions.

2.4.1 Basic principles for modifying the tree

The following subsections show the logic behind the creation of new nodes in `Fagus`. It is implemented in such a way that the tree is always modified as little as possible to perform the requested change.

Correctly handling list indices

As demonstrated in the examples for the *path*-parameter, `list` indices can be positive and negative `int`-nodes to access specific values in the list:

```
>>> a = Fagus([[[0, 1], 2], [3, 4, [5, [6, 7]], 8]])  # some nested lists to demonstrate indices
>>> a["-1 2 1 1"]  # positive and negative indices can be used to get a value
7
>>> a["0 -1"] = "two"  # the value at index -1 is replaced with the new string
>>> a()
[[[0, 1], 'two'], [3, 4, [5, [6, 7]], 8]]
```

When lists are modified, in many cases it might be desirable to append or prepend a value to the `list` instead of overriding it as shown above. This can be done as shown below:

```
>>> a.set(9, (1, 10000))  # 9 is appended as 10000 is bigger than len([3, 4, [5, [6, 7]], 8])
[[[0, 1], 'two'], [3, 4, [5, [6, 7]], 8, 9]]
>>> a.set(2.5, "1 -6")  # 2.5 is prepended before 3 as -6 is smaller than -len([3, 4, [5, [6, 7]], 8, 9])
[[[0, 1], 'two'], [2.5, 3, 4, [5, [6, 7]], 8, 9]]
```

This shows how elements easily can be appended and prepended just by specifying an index which is bigger than the length of the list to append, or smaller than minus the length of the list to prepend. In order to make sure that a value is always appended / prepended without knowing the length of the list, `INF` can be imported from the `fagus`-module, it is just a reference to `sys.maxsize`. The `FagusOption` *`list_insert`* can be used to insert a new value at an index in the middle of the `list`.

Create the correct type of node

`Fagus` is built around the concept of values being assigned to keys to build nested trees of `dict`- and `list`-nodes. The only supported operation in `set`-nodes is checking whether it `contains` a certain value, therefore `set`-nodes cannot be traversed by *get()* and are thus treated as leaf-nodes. Consequently, the only available nodes to create in the tree are `dict "d"` and `list "l"`.

The `FagusOption` ***node_types*** can be used to clearly specify which types the nodes at each level of the tree should have, see ***node_types*** example one. If `node_types` is not specified clearly or set to `" "` (don't care), ***default_node_type*** determines which type of node will be created:

```
1  >>> a = Fagus()
2  >>> a.set(True, "0 0 0", default_node_type="l")  # only lists are created, as default_
   ↪node_type="l"
3  {'0': [[True]]}
4  >>> a.clear()
5  {}
6  >>> a.set(True, "0 a 0", default_node_type="l")  # a dict is created at level 1 -->␣
   ↪can't convert "a" to a list-index
7  {'0': {'a': [True]}}
8  >>> a = Fagus()
9  >>> a.set(True, "0 0 0")  # only create dicts, as default_node_type is "d" by default
10 {'0': {'0': {'0': True}}}
```

From the example above, we can see the following two rules on how new nodes are created:

1. `list` nodes are created when `default_node_type` is `"l"` and the key can be converted to an `int` –> create `list` for keys like `8` or `"-10"`
2. `dict` nodes are always created when `default_node_type` is `d`, even if the key could be converted to an `int` –> create `dict` also for keys like `8` or `"-10"`

But what happens if there already are existing nodes?

```
1  >>> a = Fagus([{"a": [True]}])
2  >>> a.set(False, (0, "a", 1))  # the new value False is appended to the list
3  [{'a': [True, False]}]
4  >>> a.set(7, "0 a b")  # could not convert "b" to list index, so [True, False] was␣
   ↪replaced with {"b": 7}
5  [{'a': {'b': 7}}]
6  >>> a.set(3, "0 a 2", default_node_type="l")  # did not convert {"b": 7} to list -->␣
   ↪if possible always try to keep node
7  [{'a': {'b': 7, '2': 3}}]
```

This shows that as far as possible, `Fagus` will keep the existing node and not change it like in line 6. An existing node is only overridden and changed if it is not possible to convert the provided key to a list-index.

It is possible to manually override this behaviour by clearly specifying if each node should be a `dict "d"` or a `list "l"`, check out the section about ***node_types*** for examples on this.

Ensure that the required node can be modified

In a nested structure of `dict`- and `list`-nodes, there can also be unmodifyable `list`-nodes called `tuple`. As values can't be changed in a `tuple`, it has to be converted into a `list`. The following example shows how this is done in case of nested `tuple`-nodes:

```
>>> a = Fagus((((1, 0), 2), [3, 4, (5, (6, 7)), 8]))
>>> a.set("seven", "1 2 1 1")  # replacing the value 7 with the string "seven"
(((1, 0), 2), [3, 4, [5, [6, 'seven']], 8])
```

In order to replace the 7 with `"seven"` in the `tuple (6, 7)`, it has to be converted into a modifyable `list` first. `(6, 7)` however resides in another `tuple (5, (6, 7))`, so that outer `tuple` also has to be converted into a `list`. As `(5, (6, 7))` already lies in a `list`, it can be replaced with `[5, [6, "seven"]]`. The key point is that `tuple`-nodes are converted to `list`-nodes as deeply as necessary. The outermost `tuple` containing the whole tree `(((1, 0), 2), [3, 4, (5, (6, 7)), 8])` is not touched, and thus remains a `tuple`

2.4.2 set() – adding and overwriting elements

The `set()` function can be used to add or replace a value anywhere in the tree. This function is also used internally in `Fagus` whereever new nodes need to be created. See *Basic principles for modifying the tree* and ***node_types*** for examples of how `set()` can be fine-tuned. In case no further fine-tuning is used, the `set()`-operation can also be done as shown below:

```
>>> a = Fagus([], path_split="_")
>>> a.set("hello", "0_good_morning")
[{'good': {'morning': 'hello'}}]
>>> a._1_ciao = "byebye"  # the dot-notation for set() is available when path_split␣
↪is set to "_" or "__"
>>> a()  # note that the first index 1 above was prefixed with _, as variable names␣
↪can't start with a digit in Python
[{'good': {'morning': 'hello'}}, {'ciao': 'byebye'}]
>>> a["0_good_evening"] = "night"  # the []-notation is always available for set(),␣
↪a[(0, "evening")] would do the same
>>> a()
[{'good': {'morning': 'hello', 'evening': 'night'}}, {'ciao': 'byebye'}]
```

2.4.3 append() – adding a new element to a `list`

There might be cases where it is desirable to collect all elements of a certain type in a `list`. This can be done in only one step using `append()`:

```
>>> plants = Fagus()
>>> plants.append("daffodil", "flowers")  # a new list is created in the node flowers
{'flowers': ['daffodil']}
>>> plants.append("pine", "trees softwood")  # another list is created in the␣
↪category trees softwood
{'flowers': ['daffodil'], 'trees': {'softwood': ['pine']}}
>>> plants.append("rose", "flowers")  # rose is added to the existing flowers list
{'flowers': ['daffodil', 'rose'], 'trees': {'softwood': ['pine']}}
>>> plants.append("oak", "trees hardwood")  # a new list is created for hardwood␣
↪trees
{'flowers': ['daffodil', 'rose'], 'trees': {'softwood': ['pine'], 'hardwood': ['oak']}
↪}
>>> plants.append("beech", "trees hardwood")  # beech is appended to the hardwood␣
↪trees list
```

(continues on next page)

(continued from previous page)

```
{'flowers': ['daffodil', 'rose'], 'trees': {'softwood': ['pine'], 'hardwood': ['oak',
↪'beech']}}
```

As you can see, this function makes it easy to combine elements belonging to the same category in a `list` inside the tree. The pratical thing here is that it isn't necessary to worry about creating the `list` initially – if there already is a `list`, the new element is appended and if there is no `list`, a new one is created.

```
>>> plants.set("pine", ("trees", "softwood"))  # removing pine from list to put it as␣
↪a single element (for next step)
{'flowers': ['daffodil', 'rose'], 'trees': {'softwood': 'pine', 'hardwood': ['oak',
↪'beech']}}
>>> plants.append("fir", ("trees", "softwood"))  # pine is in this position already ->
↪ put pine in list, then append fir
{'flowers': ['daffodil', 'rose'], 'trees': {'softwood': ['pine', 'fir'], 'hardwood': [
↪'oak', 'beech']}}
>>> plants.append("forest", "trees")  # node trees already present at path -> convert␣
↪node to list -> append element
{'flowers': ['daffodil', 'rose'], 'trees': ['softwood', 'hardwood', 'forest']}
>>> plants = Fagus({"flowers": {"rose", "daffodil", "tulip"}})  # preparing the next␣
↪step - flowers are now in a set
>>> # below another type of node is already at path (here a set) -> convert it to a␣
↪list and then append the element
>>> plants.append("sunflower", "flowers")["flowers"].sort()  # sort list of flowers␣
↪for doctest, irrelevant for example
>>> plants()  # as you can see, {"rose", "daffodil", "tulip"} was converted to a list,
↪ then sunflower was added
{'flowers': ['daffodil', 'rose', 'sunflower', 'tulip']}
```

The examples above show that `append()` is agile and makes the best out of any situation in the tree where it is called. If there is a single element already present at the node, that element is put in a `list` before the new element is added. If there already is another type of node or another `Collection` at the requested `path`, convert that node into a `list` and then append the new element.

```
>>> plants.set("lily", "flowers 4")  # set() with an index bigger than the length of␣
↪the list can also be used to append
{'flowers': ['daffodil', 'rose', 'sunflower', 'tulip', 'lily']}
```

The example above shows that `set()` can also be used to append an element to a `list`. However, note that `set()` in this case won't create a new list if the node doesn't exist yet. It won't convert another node already present at `path` into a `list` neither.

2.4.4 extend() – extending a `list` with multiple elements

The `extend()` function works very similar to ***append()***, the main difference here is that instead of appending one additional element, the list is extended with a collection of elements.

```
>>> plants.extend(("lavender", "daisy", "orchid"), "flowers")  # extend() works like␣
↪append(), just adding more elements
{'flowers': ['daffodil', 'rose', 'sunflower', 'tulip', 'lily', 'lavender', 'daisy',
↪'orchid']}
```

For further reading about when and how new `list`-nodes are created, refer to the documentation of ***append()*** as `extend()` works similar except from the fact that several new elements are added instead of one.

2.4.5 insert() – insert an element at a given index in a `list`

The `insert()` function works similar to *append()*, the main difference is just that instead of appending the new element to the end of the `list`, it can be inserted at any position. For an overview of how and when new `list`-nodes are created before insertion, check out *append()*.

```
>>> plants = Fagus({'flowers': ['daffodil', 'rose', 'sunflower']})
>>> plants.insert(1, "tulip", "flowers")  # index parameter comes first, so the order␣
↪if args is like in list().insert()
{'flowers': ['daffodil', 'tulip', 'rose', 'sunflower']}
```

The normal indexation of `list`-nodes in `Fagus` only allows appending or prepending elements if it is necessary to do so anywhere in *path*, this is documented *here*. Check out the *list_insert* `FagusOption` for examples on how to insert new nodes at any index in the list anywhere in `path`.

2.4.6 add() – adding a new element to a `set`

The `add()` function works similar to *append()*, the main difference is just that instead of creating and appending to `list`-nodes, `set`-nodes are used. For detailed examples of the rules when and how new `set`-nodes are created by this function, check out *append()* just replacing occurrences of `list` with `set`.

```
>>> from tests.test_fagus import sorted_set  # function needed for doctests to work␣
↪with sets -> print the set sorted
>>> sorted_set(plants.add("daisy", "flowers"))  # list is converted into a set, and␣
↪then "daisy" is added to that set
{'flowers': {'daffodil', 'daisy', 'rose', 'sunflower', 'tulip'}}
>>> sorted_set(plants.add("oak", "trees"))  # node does not exist yet - create new␣
↪empty set and add the new value to it
{'flowers': {'daffodil', 'daisy', 'rose', 'sunflower', 'tulip'}, 'trees': {'oak'}}
```

2.4.7 update() – update multiple elements in a `set` or `dict`

This function works similar to *extend()* explained above, however the difference here is that the new elements now are added to a `set` or `dict`. As the function has the same name for `set` and `dict`-nodes, it has to determine what kind of node to create. Consider the following examples:

```
>>> plants = Fagus()  # sorted_set() is used to always print sets deterministic, this␣
↪is needed internally for doctests
>>> sorted_set(plants.update(dict(softwood="pine", hardwood="oak"), "trees"))  #␣
↪creating and updating dict
{'trees': {'softwood': 'pine', 'hardwood': 'oak'}}
>>> sorted_set(plants.update(("tulip", "daisy", "daffodil"), "flowers"))  # create␣
↪set from tuple
{'trees': {'softwood': 'pine', 'hardwood': 'oak'}, 'flowers': {'daffodil', 'daisy',
↪'tulip'}}
>>> plants.clear("flowers")  # emptying this set to keep the example easily readable
{'trees': {'softwood': 'pine', 'hardwood': 'oak'}, 'flowers': set()}
>>> sorted_set(plants.update({"garden flowers": "sunflower", "flower trees": "apple␣
↪tree"}, "flowers"))  # comment below
{'trees': {'softwood': 'pine', 'hardwood': 'oak'}, 'flowers': {'flower trees',
↪'garden flowers'}}
>>> # as you can see, even though a dict was sent in as a parameter, the flowers node␣
↪stayed a set, so only "flower
>>> # trees" and "garden flowers" were added, but not "apple tree" and "sunflower"
```

The examples above illustrate first two of the principles `update()` operates after:

1. If there already is a `dict`- or `set` object at *path*, keep that node if possible.
2. If there already exists a `set`, and a `dict` is passed to `update()`, the `set` is updated with the keys from the dict only (line 8).

```
1  >>> sorted_set(plants.set({"fruit trees": ["apple tree", "lemon tree"]}, "trees"))  #␣
   ↪prepare the next example
2  {'trees': {'fruit trees': ['apple tree', 'lemon tree']}, 'flowers': {'flower trees',
   ↪'garden flowers'}}
3  >>> sorted_set(plants.update((("hardwood", "oak"), ("softwood", "fir")), "trees"))  #␣
   ↪comment below
4  {'trees': {('hardwood', 'oak'), ('softwood', 'fir')}, 'flowers': {'flower trees',
   ↪'garden flowers'}}
5  >>> # it is not possible to update a dict from these tuples -> replace the previous␣
   ↪dict with a new set with the tuples
6  >>> plants = Fagus({"trees": {"hardwood": "beech", "softwood": "fir"}})  # making
   ↪"trees" a dict again for next example
7  >>> sorted_set(plants.update(dict((("hardwood", "oak"), ("softwood", "pine"))), "trees
   ↪"))  # comment below
8  {'trees': {'hardwood': 'oak', 'softwood': 'pine'}}
9  >>> # Here it is shown how a dict can be updated based on a list of tuples with two␣
   ↪elements, or e.g. the iterator
10 >>> # dict.items() returns. By passing the list of tuples to the dict() function␣
   ↪first, Fagus detects your intention
11 >>> # to update a dict instead of overwriting it with a set
```

The third principle `update()` operates after is the following: 3. If you would like to update a `dict`, you must pass a `Mapping` (the type of key-value containers like `dict`). If you just pass e.g. a `tuple` of `tuple`-nodes with two elements or `dict.items()`, the `dict` will be overwritten with a `set`. To update the dict, just pass e.g. the `tuple` of `tuple`-nodes through `dict()` before passing it to `update()`. For any `Iterable` that is not a `Mapping`, the `Mapping` will be removed and a `set` will be created.

Especially this last principle may seem tedious, however it was chosen to implement it that way to prevent ambiguity, and the main reason for that is the `update()` function being used in `set`-nodes as well as `dict`-nodes.

2.4.8 remove(), delete() and pop()

2.4.9 serialize() – ensure that a tree is json- or yaml-serializable

2.4.10 mod() – modifying elements

2.5 Iterating over nested objects

2.5.1 Skipping nodes in iteration.

2.6 Filtering nested objects

CHAPTER

THREE

FAGUS PACKAGE

Library to easily create, edit and traverse nested objects of dicts and lists in Python

The following objects can be imported directly from this module:

- `Fagus`: a wrapper-class for complex, nested objects of dicts and lists
- `Fil`, `CFil` and `VFil` are filter-objects that can be used to filter `Fagus`-objects
- `INF`: alias for `sys.maxsize`, used e.g. to indicate that an element should be appended to a list

Submodules in `fagus`:

- `fagus`: Base-module that contains the `Fagus`-class
- `filters`: filter-classes for filtering `Fagus`-objects
- `iterators`: iterator-classes for iterating on `Fagus`
- `utils`: helper classes and methods for `Fagus`

`class fagus.Fagus(`*root: Optional[Collection[Any]] = None*, *node_types: OptStr = Ellipsis*, *list_insert: OptInt = Ellipsis*, *path_split: OptStr = Ellipsis*, *fagus: OptBool = Ellipsis*, *default_node_type: OptStr = Ellipsis*, *default: OptAny = Ellipsis*, *if_: OptAny = Ellipsis*, *iter_fill: OptAny = Ellipsis*, *mod_functions: Union[Mapping[Union[type, Tuple[type], str], Callable[[Any], Any]], ellipsis] = Ellipsis*, *copy: bool = False*`)`

Bases: `MutableMapping`, `MutableSequence`, `MutableSet`

Fagus is a wrapper-class for complex, nested objects of dicts and lists in Python

Fagus can be used as an object by instantiating it, but it's also possible to use all methods statically without even an object, so that `a = {}; Fagus.set(a, "top med", 1)` and `a = Fagus({}); a.set(1, "top med")` do the same.

The root node is always modified directly. If you don't want to change the root node, all the functions where it makes sense support to rather modify a copy, and return that modified copy using the copy-parameter.

FagusOptions: Several parameters used in functions in Fagus work as options so that you don't have to specify them each time you run a function. In the docstrings, these options are marked with a *, e.g. the fagus parameter is an option. Options can be specified at three levels with increasing precedence: at class-level (`Fagus.fagus = True`), at object-level (`a = Fagus(), a.fagus = True`) and in each function-call (`a.get("b", fagus=True)`). If you generally want to change an option, change it at class-level - all objects in that file will inherit this option. If you want to change the option specifically for one object, change the option at object-level. If you only want to change the option for one single run of a function, put it as a function-parameter. More thorough examples of options can be found in README.md.

`__init__`(*root: Optional[Collection[Any]] = None*, *node_types: OptStr = Ellipsis*, *list_insert: OptInt = Ellipsis*, *path_split: OptStr = Ellipsis*, *fagus: OptBool = Ellipsis*, *default_node_type: OptStr = Ellipsis*, *default: OptAny = Ellipsis*, *if_: OptAny = Ellipsis*, *iter_fill: OptAny = Ellipsis*, *mod_functions: Union[Mapping[Union[type, Tuple[type], str], Callable[[Any], Any]], ellipsis] = Ellipsis*, *copy: bool = False*)

Constructor for Fagus, a wrapper-class for complex, nested objects of dicts and lists in Python

* means that the parameter is a FagusOption, see Fagus-class-docstring for more information about options

Parameters

- **root** – object (like dict / list) to wrap Fagus around. If this is None, an empty node of the type default_node_type will be used. Default None
- **node_types** – * Can be used to manually define if the nodes along path are supposed to be (l)ists or (d)icts. E.g. `"dll"` to create a dict at level 1, and lists at level 2 and 3. `" "` can also be used – space doesn't enforce a node-type like `"d"` or `"l"`. For `" "`, existing nodes are traversed if possible, otherwise default_node_type is used to create new nodes. Default `""`, interpreted as " " at each level. See README
- **list_insert** – * Level at which a new node shall be inserted into the list instead of traversing the existing node in the list at that index. See README
- **path_split** – * used to split path into a list if path is a string, default `" "`, see README
- **fagus** – * this option is used to determine whether nodes in the returned object should be returned as Fagus-objects. This can be useful e.g. if you want to use Fagus in an iteration. Check the particular function you want to use for a more thorough explanation of what this does in each case
- **default_node_type** – * determines if new nodes by default should be created as (d)ict or (l)ist. Must be either `"d"` or `"l"`, default `"d"`, examples in README
- **default** – * ~ is used in get and other functions if a path doesn't exist
- **if_** – * only set value if it meets the condition specified here, otherwise do nothing. The condition can be a lambda, any value or a tuple of accepted values. Default _None (don't check value)
- **iter_fill** – * Fill up tuples with iter_fill (can be any object, e.g. None) to ensure that all the tuples iter() returns are exactly max_items long. See iter()
- **mod_functions** – * ~ is used to define how different types of objects are supposed to be serialized. This is defined in a dict. The keys are either a type (like IPAddress) or a tuple of different types (IPv4Address, IPv6Address). The values are function pointers, or lambdas, which are supposed to convert e.g. an IPv4Address into a string. Check out TFunc if you want to call more complicated functions with several arguments. See README for examples
- **copy** – ~ creates a copy of the root node before Fagus is initialized. Makes sure that changes on this Fagus won't modify the root node that was passed here itself. Default False

`root: Collection[Any]`

Contains the root note the Fagus-object is wrapped around

This can be used to remove the Fagus-wrapper in case the plain object is needed, e.g. if `a = Fagus(["ex"])`, `a.root = ["ex"]`. The root node is also returned when a is called: `a()`, examples in `Fagus.__call__()`.

`get`(*path: Any = ''*, *default: OptAny = Ellipsis*, *fagus: OptBool = Ellipsis*, *copy: bool = False*, *path_split: OptStr = Ellipsis*) → Any

Retrieves value at path. If the value doesn't exist, default is returned.

To get `"hello"` from `x = Fagus({"a": ["b", {"c": "d"}], e: ["f", "g"]})`, you can use `x[("a", 1, "c")]`. The tuple `("a", 1, "c")` is the path-parameter that is used to traverse x. At first, the list at `"a"` is picked in the top-most dict, and then the 2nd element `{"c": "d"}` is picked from that list. Then, "d" is picked from `{"c": "d"}` and returned. The path-parameter can be a tuple or list, the keys must be either integers for lists, or any hashable objects for dicts. For convenience, the keys can also be put in a single string separated by path_split (default `" "`), so `a["a 1 c"]` also returns `"d"`.

* means that the parameter is a FagusOption, see Fagus-class-docstring for more information about options

Parameters

- **path** – List/Tuple of key-values to recursively traverse self. Can also be specified as string, that is split into a tuple using path_split
- **default** – * returned if path doesn't exist in self
- **fagus** – * returns a Fagus-object if the value at path is a list or dict
- **copy** – Option to return a copy of the returned value. The default behaviour is that if there are subnodes (dicts, lists) in the returned values, and you make changes to these nodes, these changes will also be applied in the root node from which values() was called. If you want the returned values to be independent, use copy to get a shallow copy of the returned value
- **path_split** – * used to split path into a list if path is a str, default `" "`

Returns

the value if the path exists, or default if it doesn't exist

iter(*max_depth: int = 9223372036854775807*, *path: Any = ''*, *filter_: Optional[Fil] = None*, *fagus: OptBool = Ellipsis*, *iter_fill: OptAny = Ellipsis*, *select: Optional[Union[int, Iterable[Any]]] = None*, *copy: bool = False*, *iter_nodes: OptBool = Ellipsis*, *filter_ends: bool = False*, *path_split: OptStr = Ellipsis*) → FagusIterator

Recursively iterate through Fagus-object, starting at path

* means that the parameter is a FagusOption, see Fagus-class-docstring for more information about options

Parameters

- **max_depth** – Can be used to limit how deep the iteration goes. Example: `a = {"a": ["b", ["c", "d"]], "e": "f"}` If max_depth is sys.max_size, all the nodes are traversed: `[("a", "b", "c"), ("a", "b", "d"]), ("e", "f")]`. If max_depth is 1, iter returns `[("a", "b", ["c", "d"]), ("e", "f")]`, so `["c", "d"]` is not iterated through but returned as a node. If max_depth is 0, iter returns `[("a", ["b", ["c", "d"]]), ("e", "f")]`, effectively the same as dict.items(). Default sys.maxitems (iterate as deeply as possible). A negative number (e.g. -1) is treated as sys.maxitems.
- **path** – Start iterating at path. Internally calls get(path), and iterates on the node get returns. See get()
- **filter_** – Only iterate over specific nodes defined using Fil (see README.md and Fil for more info)
- **fagus** – * If the leaf in the tuple is a dict or list, return it as a Fagus-object. This option has no effect if max_items is sys.maxitems.
- **iter_fill** – * Fill up tuples with iter_fill (can be any object, e.g. None) to ensure that all the tuples iter() returns are exactly max_items long. This can be useful if you want to unpack the keys / leaves from the tuples in a loop,

which fails if the count of items in the tuples varies. This option has no effect if max_items is -1. The default value is ..., meaning that the tuples are not filled, and the length of the tuples can vary. See README for a more thorough example.

- **select** – Extract only some specified values from the tuples. E.g. if ~ is -1, only the leaf-values are returned. ~ can also be a list of indices. Default None (don't reduce the tuples)
- **copy** – Iterate on a shallow-copy to make sure that you can edit root node without disturbing the iteration
- **iter_nodes** – * includes the traversed nodes into the resulting tuples, order is then: node1, key1, node2, key2, ..., leaf_value
- **filter_ends** – Affects the end dict/list that is returned if max_items is used. Normally, filters are not applied on that end node. If you would like to get the end node filtered too, set this to True. If this is set to True, the last nodes will always be copies (if unfiltered they are references)
- **path_split** – * used to split path into a list if path is a str, default " ", see README

Returns

FagusIterator with one tuple for each leaf-node, containing the keys of the parent-nodes until the leaf

filter(*filter_: Fil*, *path: Any = ''*, *fagus: OptBool = Ellipsis*, *copy: bool = False*, *default: OptAny = Ellipsis*, *path_split: OptStr = Ellipsis*) → Collection[Any]

Filters self, only keeping the nodes that pass the filter

* means that the parameter is a FagusOption, see Fagus-class-docstring for more information about options

Parameters

- **filter_** – Fil-object in which the filtering-criteria are specified
- **path** – at this point in self, the filtering will start (apply filter_ relatively from this point). Default "", meaning that the root node is filtered, see get() and README for examples
- **fagus** – * return the filtered self as Fagus-object (default is just to return the filtered node)
- **copy** – Create a copy and filter on that copy. Default is to modify the self directly
- **default** – * returned if path doesn't exist in self, or the value at path can't be filtered
- **path_split** – * used to split path into a list if path is a string, default " ", see README

Returns

the filtered object, starting at path

Raises

TypeError – if the root node needs to be modified and isn't modifiable (e.g. tuple or frozenset)

split(*filter_: Optional[Fil]*, *path: Any = ''*, *fagus: OptBool = Ellipsis*, *copy: bool = False*, *default: OptAny = Ellipsis*, *path_split: OptStr = Ellipsis*) → Union[Tuple[Collection[Any], Collection[Any]], Tuple[Any, Any]]

Splits self into nodes that pass the filter, and nodes that don't pass the filter

* means that the parameter is a FagusOption, see Fagus-class-docstring for more information about options

Parameters

- **filter_** – Fil-object in which the filtering-criteria are specified
- **path** – at this position in self, the splitting will start (apply filter_ relatively from this point). Default "", meaning that the root node is split, see get() and README for examples
- **fagus** – * return the filtered self as Fagus-object (default is just to return the filtered node)
- **copy** – Create a copy and filter on that copy. Default is to modify the object directly
- **default** – * returned if path doesn't exist in self
- **path_split** – * used to split path into a list if path is a str, default " "

Returns

a tuple, where the first element is the nodes that pass the filter, and the second element is the nodes that don't pass the filter

Raises

TypeError – if the root node needs to be modified and isn't modifiable (e.g. tuple or frozenset)

set(*value: Any*, *path: Iterable[Any]*, *node_types: OptStr = Ellipsis*, *list_insert: OptInt = Ellipsis*, *path_split: OptStr = Ellipsis*, *fagus: OptBool = Ellipsis*, *if_: OptAny = Ellipsis*, *default_node_type: OptStr = Ellipsis*, *copy: bool = False*) → Collection[Any]

Create (if they don't already exist) all sub-nodes in path, and finally set value at leaf-node

* means that the parameter is a FagusOption, see Fagus-class-docstring for more information about options

Parameters

- **value** – ~ is placed at path, after creating new nodes if necessary. An existing value at path is overwritten
- **path** – List/Tuple of key-values that are traversed in self. If no nodes exist at the keys, new nodes are created. Can also be specified as a string, that is split into a tuple using path_split. See get()
- **node_types** – * Can be used to manually define if the nodes along path are supposed to be (l)ists or (d)icts. E.g. "dll" to create a dict at level 1, and lists at level 2 and 3. " " can also be used – space doesn't enforce a node-type like "d" or "l". For " ", existing nodes are traversed if possible, otherwise default_node_type is used to create new nodes. Default "", interpreted as " " at each level. See README
- **list_insert** – * Level at which a new node shall be inserted into the list instead of traversing the existing node in the list at that index. See README
- **path_split** – * used to split path into a list if path is a string, default " ", see README
- **fagus** – * return self as a Fagus-object if it is a node (tuple / list / dict), default False
- **if_** – * only set value if it meets the condition specified here, otherwise do nothing. The condition can be a lambda, any value or a tuple of accepted values. Default _None (don't check value)
- **default_node_type** – * determines if new nodes by default should be created as (d)ict or (l)ist. Must be either "d" or "l", default "d", examples in README

- **copy** – if this is set, a copy of self is modified and then returned (thus self is not modified)

Returns
self as a node if fagus is set, or a modified copy of self if copy is set

Raises

- **ValueError** – if it isn't possible to parse an int-index from the provided key in a position where node-types defines that the node shall be a list (if node-types is not l, the node will be replaced with a dict)
- **TypeError** – if the root node needs to be modified and isn't modifiable (e.g. tuple or frozenset)

append(*value: Any*, *path: Any = ''*, *node_types: OptStr = Ellipsis*, *list_insert: OptInt = Ellipsis*, *path_split: OptStr = Ellipsis*, *fagus: OptBool = Ellipsis*, *if_: OptAny = Ellipsis*, *default_node_type: OptStr = Ellipsis*, *copy: bool = False*) → Collection[Any]

Create (if they don't already exist) all sub-nodes in path, and finally append value to a list at leaf-node

If the leaf-node is a set, tuple or other value it is converted to a list. Then the new value is appended.

* means that the parameter is a FagusOption, see Fagus-class-docstring for more information about options

Parameters

- **value** – ~ is appended to list at path, after creating new nodes along path as necessary
- **path** – List/Tuple of key-values that are traversed in self. If no nodes exist at the keys, new nodes are created. Can also be specified as a string, that is split into a tuple using path_split. See get()
- **node_types** – * Can be used to manually define if the nodes along path are supposed to be (l)ists or (d)icts. E.g. `"dll"` to create a dict at level 1, and lists at level 2 and 3. `" "` can also be used – space doesn't enforce a node-type like `"d"` or `"l"`. For `" "`, existing nodes are traversed if possible, otherwise default_node_type is used to create new nodes. Default `""`, interpreted as " " at each level. See README
- **list_insert** – * Level at which a new node shall be inserted into the list instead of traversing the existing node in the list at that index. See README
- **path_split** – * used to split path into a list if path is a string, default `" "`, see README
- **fagus** – * return self as a Fagus-object if it is a node (tuple / list / dict), default False
- **if_** – * only append value if it meets the condition specified here, otherwise do nothing. The condition can be a lambda, any value or a tuple of accepted values. Default _None (don't check value)
- **default_node_type** – * determines if new nodes by default should be created as (d)ict or (l)ist. Must be either `"d"` or `"l"`, default `"d"`, examples in README
- **copy** – if this is set, a copy of self is modified and then returned (thus self is not modified)

Returns
self as a node if fagus is set, or a modified copy of self if copy is set

Raises

- **ValueError** – if it isn't possible to parse an int-index from the provided key in a position where node-types defines that the node shall be a list (if node-types is not l, the node will be replaced with a dict)
- **TypeError** – if path is empty and the root node is not a list (can't append to a dict, tuple or set) or the root node needs to be modified and isn't modifiable (e.g. tuple or frozenset)

extend(*values: Iterable[Any]*, *path: Any = ''*, *node_types: OptStr = Ellipsis*, *list_insert: OptInt = Ellipsis*, *path_split: OptStr = Ellipsis*, *fagus: OptBool = Ellipsis*, *if_: OptAny = Ellipsis*, *default_node_type: OptStr = Ellipsis*, *copy: bool = False*) → Collection[Any]

Create (if they don't already exist) all sub-nodes in path. Then extend list at leaf-node with the new values

If the leaf-node is a set, tuple or other value it is converted to a list, which is extended with the new values

* means that the parameter is a FagusOption, see Fagus-class-docstring for more information about options

Parameters

- **values** – the list at path is extended with ~, after creating new nodes along path as necessary
- **path** – List/Tuple of key-values that are traversed in self. If no nodes exist at the keys, new nodes are created. Can also be specified as a string, that is split into a tuple using path_split. See get()
- **node_types** – * Can be used to manually define if the nodes along path are supposed to be (l)ists or (d)icts. E.g. `"dll"` to create a dict at level 1, and lists at level 2 and 3. `" "` can also be used – space doesn't enforce a node-type like `"d"` or `"l"`. For `" "`, existing nodes are traversed if possible, otherwise default_node_type is used to create new nodes. Default `""`, interpreted as " " at each level. See README
- **list_insert** – * Level at which a new node shall be inserted into the list instead of traversing the existing node in the list at that index. See README
- **path_split** – * used to split path into a list if path is a string, default `" "`, see README
- **fagus** – * return self as a Fagus-object if it is a node (tuple / list / dict), default False
- **if_** – * only extend with values if they meet the condition specified here, otherwise do nothing. The condition can be a lambda, any value or a tuple of accepted values. Default _None (don't check values)
- **default_node_type** – * determines if new nodes by default should be created as (d)ict or (l)ist. Must be either `"d"` or `"l"`, default `"d"`, examples in README
- **copy** – if this is set, a copy of self is modified and then returned (thus self is not modified)

Returns

self as a node if fagus is set, or a modified copy of self if copy is set

Raises

- **ValueError** – if it isn't possible to parse an int-index from the provided key in a position where node-types defines that the node shall be a list (if node-types is not l, the node will be replaced with a dict)
- **TypeError** – if path is empty and the root node is not a list (can't extend a dict, tuple or set) or the root node needs to be modified and isn't modifiable (e.g. tuple or frozenset)

insert(*index: int*, *value: Any*, *path: Any = ''*, *node_types: OptStr = Ellipsis*, *list_insert: OptInt = Ellipsis*, *path_split: OptStr = Ellipsis*, *fagus: OptBool = Ellipsis*, *if_: OptAny = Ellipsis*, *default_node_type: OptStr = Ellipsis*, *copy: bool = False*) → Collection[Any]

Create (if they don't already exist) all sub-nodes in path. Insert new value at index in list at leaf-node

If the leaf-node is a set, tuple or other value it is converted to a list, in which the new value is inserted at index

* means that the parameter is a FagusOption, see Fagus-class-docstring for more information about options

Parameters

- **index** – ~ at which the value shall be inserted in the list at path
- **value** – ~ is inserted at index into list at path, after creating new nodes along path as necessary
- **path** – List/Tuple of key-values that are traversed in self. If no nodes exist at the keys, new nodes are created. Can also be specified as a string, that is split into a tuple using path_split. See get()
- **node_types** – * Can be used to manually define if the nodes along path are supposed to be (l)ists or (d)icts. E.g. `"dll"` to create a dict at level 1, and lists at level 2 and 3. `" "` can also be used – space doesn't enforce a node-type like `"d"` or `"l"`. For `" "`, existing nodes are traversed if possible, otherwise default_node_type is used to create new nodes. Default `""`, interpreted as " " at each level. See README
- **list_insert** – * Level at which a new node shall be inserted into the list instead of traversing the existing node in the list at that index. See README
- **path_split** – * used to split path into a list if path is a string, default `" "`, see README
- **fagus** – * return self as a Fagus-object if it is a node (tuple / list / dict), default False
- **if_** – * only insert value if it meets the condition specified here, otherwise do nothing. The condition can be a lambda, any value or a tuple of accepted values. Default _None (don't check value)
- **default_node_type** – * determines if new nodes by default should be created as (d)ict or (l)ist. Must be either `"d"` or `"l"`, default `"d"`, examples in README
- **copy** – if this is set, a copy of self is modified and then returned (thus self is not modified)

Returns

self as a node if fagus is set, or a modified copy of self if copy is set

Raises

- **ValueError** – if it isn't possible to parse an int-index from the provided key in a position where node-types defines that the node shall be a list (if node-types is not l, the node will be replaced with a dict)
- **TypeError** – if path is empty and the root node is not a list (can't insert into dict, tuple or set) or the root node needs to be modified and isn't modifiable (e.g. tuple or frozenset)

add(*value: Any*, *path: Any = ''*, *node_types: OptStr = Ellipsis*, *list_insert: OptInt = Ellipsis*, *path_split: OptStr = Ellipsis*, *fagus: OptBool = Ellipsis*, *if_: OptAny = Ellipsis*, *default_node_type: OptStr = Ellipsis*, *copy: bool = False*) → Collection[Any]

Create (if they don't already exist) all sub-nodes in path, and finally add new value to set at leaf-node

If the leaf-node is a list, tuple or other value it is converted to a set, to which the new value is added

* means that the parameter is a FagusOption, see Fagus-class-docstring for more information about options

Parameters

- **value** – ~ is added to set at path, after creating new nodes along path as necessary
- **path** – List/Tuple of key-values that are traversed in self. If no nodes exist at the keys, new nodes are created. Can also be specified as a string, that is split into a tuple using path_split. See get()
- **node_types** – * Can be used to manually define if the nodes along path are supposed to be (l)ists or (d)icts. E.g. `"dll"` to create a dict at level 1, and lists at level 2 and 3. `" "` can also be used – space doesn't enforce a node-type like `"d"` or `"l"`. For `" "`, existing nodes are traversed if possible, otherwise default_node_type is used to create new nodes. Default `""`, interpreted as " " at each level. See README
- **list_insert** – * Level at which a new node shall be inserted into the list instead of traversing the existing node in the list at that index. See README
- **path_split** – * used to split path into a list if path is a string, default `" "`, see README
- **fagus** – * return self as a Fagus-object if it is a node (tuple / list / dict), default False
- **if_** – * only add value if it meets the condition specified here, otherwise do nothing. The condition can be a lambda, any value or a tuple of accepted values. Default _None (don't check value)
- **default_node_type** – * determines if new nodes by default should be created as (d)ict or (l)ist. Must be either `"d"` or `"l"`, default `"d"`, examples in README
- **copy** – if this is set, a copy of self is modified and then returned (thus self is not modified)

Returns

self as a node if fagus is set, or a modified copy of self if copy is set

Raises

- **ValueError** – if it isn't possible to parse an int-index from the provided key in a position where node-types defines that the node shall be a list (if node-types is not l, the node will be replaced with a dict)
- **TypeError** – if path is empty and the root node is not a set (can't add to list or dict) or the root node needs to be modified and isn't modifiable (e.g. tuple or frozenset)

update(*values: Iterable[Any]*, *path: Any = ''*, *node_types: OptStr = Ellipsis*, *list_insert: OptInt = Ellipsis*, *path_split: OptStr = Ellipsis*, *fagus: OptBool = Ellipsis*, *if_: OptAny = Ellipsis*, *default_node_type: OptStr = Ellipsis*, *copy: bool = False*) → Collection[Any]

Create (if they don't already exist) all sub-nodes in path, then update set at leaf-node with new values

If the leaf-node is a list, tuple or other value it is converted to a set. That set is then updated with the new values. If the node at path is a dict, and values also is a dict, the node-dict is updated with the new values.

* means that the parameter is a FagusOption, see Fagus-class-docstring for more information about options

Parameters

- **values** – the set/dict at path is updated with ~, after creating new nodes along path as necessary
- **path** – List/Tuple of key-values that are traversed in self. If no nodes exist at the keys, new nodes are created. Can also be specified as a string, that is split into a tuple using path_split. See get()
- **node_types** – * Can be used to manually define if the nodes along path are supposed to be (l)ists or (d)icts. E.g. `"dll"` to create a dict at level 1, and lists at level 2 and 3. `" "` can also be used – space doesn't enforce a node-type like `"d"` or `"l"`. For `" "`, existing nodes are traversed if possible, otherwise default_node_type is used to create new nodes. Default `""`, interpreted as " " at each level. See README
- **list_insert** – * Level at which a new node shall be inserted into the list instead of traversing the existing node in the list at that index. See README
- **path_split** – * used to split path into a list if path is a string, default `" "`, see README
- **fagus** – * return self as a Fagus-object if it is a node (tuple / list / dict), default False
- **if_** – * only update with values if they meet the condition specified here, otherwise do nothing. The condition can be a lambda, any value or a tuple of accepted values. Default _None (don't check values)
- **default_node_type** – * determines if new nodes by default should be created as (d)ict or (l)ist. Must be either `"d"` or `"l"`, default `"d"`, examples in README
- **copy** – if this is set, a copy of self is modified and then returned (thus self is not modified)

Returns

self as a node if fagus is set, or a modified copy of self if copy is set

Raises

- **ValueError** – if it isn't possible to parse an int-index from the provided key in a position where node-types defines that the node shall be a list (if node-types is not l, the node will be replaced with a dict)
- **TypeError** – if path is empty and the root node is not a set or dict (can't update list) or the root node needs to be modified and isn't modifiable (e.g. tuple or frozenset)

setdefault(*path: Any = ''*, *default: OptAny = Ellipsis*, *fagus: OptBool = Ellipsis*, *node_types: OptStr = Ellipsis*, *list_insert: OptInt = Ellipsis*, *path_split: OptStr = Ellipsis*, *default_node_type: OptStr = Ellipsis*) → Any

Get value at path and return it. If there is no value at path, set default at path, and return default

* means that the parameter is a FagusOption, see Fagus-class-docstring for more information about options

Parameters

- **path** – position in self where default shall be set / from where value shall be fetched. See get() and README
- **default** – * returned if path doesn't exist in self

- **fagus** – * return self as a Fagus-object if it is a node (tuple / list / dict), default False
- **node_types** – * Can be used to manually define if the nodes along path are supposed to be (l)ists or (d)icts. E.g. `"dll"` to create a dict at level 1, and lists at level 2 and 3. `" "` can also be used – space doesn't enforce a node-type like `"d"` or `"l"`. For `" "`, existing nodes are traversed if possible, otherwise default_node_type is used to create new nodes. Default `""`, interpreted as `" "` at each level. See README
- **list_insert** – * Level at which a new node shall be inserted into the list instead of traversing the existing node in the list at that index. See README
- **path_split** – * used to split path into a list if path is a str, default `" "`
- **default_node_type** – * determines if new nodes by default should be created as (d)ict or (l)ist. Must be either `"d"` or `"l"`, default `"d"`, examples in README

Returns
value at path if it exists, otherwise default is set at path and returned

Raises

- **ValueError** – if it isn't possible to parse an int-index from the provided key in a position where node-types defines that the node shall be a list (if node-types is not l, the node will be replaced with a dict)
- **TypeError** – if the root node needs to be modified and isn't modifiable (e.g. tuple or frozenset)

mod(*mod_function: Callable[[Any], Any]*, *path: Iterable[Any]*, *default: OptAny = Ellipsis*, *replace_value: bool = True*, *fagus: OptBool = Ellipsis*, *node_types: OptStr = Ellipsis*, *list_insert: OptInt = Ellipsis*, *path_split: OptStr = Ellipsis*, *default_node_type: OptStr = Ellipsis*) → Any

Modifies the value at path using the function-pointer mod_function

mod can be used like this Fagus.mod(obj, "kitchen spoon", lambda x: x + 1, 1) to count the number of spoons in the kitchen. If there is no value to modify, the default value (here 1) will be set at the node.

* means that the parameter is a FagusOption, see Fagus-class-docstring for more information about options

Parameters

- **mod_function** – A function pointer or lambda that modifies the existing value at path. TFunc can be used to call more complex functions requiring several arguments.
- **path** – position in self at which the value shall be modified. Defined as a list/Tuple of key-values to recursively traverse self. Can also be specified as string which is split into a tuple using path_split
- **default** – * this value is set in path if it doesn't exist
- **fagus** – * Return new value as a Fagus-object if it is a node (tuple / list / dict), default False
- **replace_value** – Replace the old value with what mod_function returns. Can be deactivated e.g. if mod_function changes the object, but returns None (if ~ stays on, the object is replaced with None). Default True. If no value exists at path, the default value is always set at path (independent of ~)
- **node_types** – * Can be used to manually define if the nodes along path are supposed to be (l)ists or (d)icts. E.g. `"dll"` to create a dict at level 1, and lists at level 2 and 3. `" "` can also be used – space doesn't enforce a node-type like `"d"` or `"l"`. For `" "`, existing nodes are traversed if possible, otherwise

default_node_type is used to create new nodes. Default `""`, interpreted as " " at each level. See README

- **list_insert** – * Level at which a new node shall be inserted into the list instead of traversing the existing node in the list at that index. See README
- **path_split** – * used to split path into a list if path is a str, default `" "`
- **default_node_type** – * determines if new nodes by default should be created as (d)ict or (l)ist. Must be either `"d"` or `"l"`, default `"d"`, examples in README

Returns
the new value that was returned by the mod_function, or default if there was no value at path

Raises

- **ValueError** – if it isn't possible to parse an int-index from the provided key in a position where node-types defines that the node shall be a list (if node-types is not l, the node will be replaced with a dict)
- **TypeError** – if the root node needs to be modified and isn't modifiable (e.g. tuple or frozenset)

mod_all(*mod_function: Callable[[Any], Any]*, *filter_: Optional[Fil] = None*, *path: Any = ''*, *replace_value: bool = True*, *default: OptAny = Ellipsis*, *max_depth: int = 9223372036854775807*, *fagus: OptBool = Ellipsis*, *copy: bool = False*, *path_split: OptStr = Ellipsis*) → Any

Modify all the leaf-values that match a certain filter

* means that the parameter is a FagusOption, see Fagus-class-docstring for more information about options

Parameters

- **mod_function** – A function pointer or lambda that modifies the existing value at path. TFunc can be used to call more complex functions requiring several arguments.
- **filter_** – used to select which leaves shall be modified. Default None (all leaves are modified)
- **path** – position in self at which the value shall be modified. See get() / README
- **default** – * this value is returned if path doesn't exist, or if no leaves match the filter
- **fagus** – * Return new value as a Fagus-object if it is a node (tuple / list / dict), default False
- **replace_value** – Replace the old value with what mod_function returns. Can be deactivated e.g. if mod_function changes the object, but returns None (if ~ stays on, the object is replaced with None). Default True. If no value exists at path, the default value is always set at path (independent of ~)
- **max_depth** – Defines the maximum depth for the iteration. See Fagus.iter max_depth for more information
- **copy** – Can be ued to make sure that the node at path is not modified (instead a modified copy is returned)
- **path_split** – * used to split path into a list if path is a str, default `" "`

Returns
the node at path where all the leaves matching filter_ are modified, or default if it didn't exist

Raises

TypeError – if the root node needs to be modified and isn't modifiable (e.g. tuple or frozenset)

serialize(*mod_functions: Optional[Mapping[Union[type, Tuple[type], str], Callable[[Any], Any]]] = None*, *path: Any = ''*, *node_types: OptStr = Ellipsis*, *list_insert: OptInt = Ellipsis*, *path_split: OptStr = Ellipsis*, *copy: bool = False*) → Union[Dict[Any, Any], List[Any]]

Makes sure the object can be serialized so that it can be converted to JSON, YAML etc.

The only allowed data-types for serialization are: dict, list, bool, float, int, str, None

Sets and tuples are converted into lists. Other objects whose types are not allowed in serialized objects are modified to a type that is allowed using the mod_functions-parameter. mod_functions is a dict, with the type of object like IPv4Address or a tuple of types like (IPv4Address, IPv6Address). The values are function pointers or lambdas, that are executed to convert e.g. an IPv4Address to one of the allowed data types mentioned above.

The default mod_functions are: {datetime: lambda x: x.isoformat(), date: lambda x: x.isoformat(), time: lambda x: x.isoformat(), "default": lambda x: str(x)}

By default, date, datetime and time-objects are replaced by their isoformat-string. All other objects whose types don't appear in mod_functions are modified by the function behind the key "default". By default, this function is lambda x: str(x) that replaces the object with its string-representation.

* means that the parameter is a FagusOption, see Fagus-class-docstring for more information about options

Parameters

- **mod_functions** – * ~ is used to define how different types of objects are supposed to be serialized. This is defined in a dict. The keys are either a type (like IPAddress) or a tuple of different types (IPv4Address, IPv6Address). The values are function pointers, or lambdas, which are supposed to convert e.g. an IPv4Address into a string. Check out TFunc if you want to call more complicated functions with several arguments. See README for examples
- **path** – position in self at which the value shall be modified. See get() / README
- **node_types** – * Can be used to manually define if the nodes along path are supposed to be (l)ists or (d)icts. E.g. `"dll"` to create a dict at level 1, and lists at level 2 and 3. `" "` can also be used – space doesn't enforce a node-type like `"d"` or `"l"`. For `" "`, existing nodes are traversed if possible, otherwise default_node_type is used to create new nodes. Default `""`, interpreted as " " at each level. See README
- **list_insert** – * Level at which a new node shall be inserted into the list instead of traversing the existing node in the list at that index. See README
- **path_split** – * used to split path into a list if path is a str, default `" "`
- **copy** – Create a copy and make that copy serializable. Default is to modify self directly

Returns

a serializable object that only contains types allowed in json or yaml

Raises

- **TypeError** – if root node is not a dict or list (serialize can't fix that for the root node)
- **ValueError** – if tuple_keys is not defined in mod_functions and a dict has tuples as keys

- **Exception** – Can raise any exception if it occurs in one of the mod_functions

merge(*obj: Union[FagusIterator, Collection[Any]]*, *path: Any = ''*, *new_value_action: str = 'r'*, *extend_from: int = 9223372036854775807*, *update_from: int = 9223372036854775807*, *fagus: OptBool = Ellipsis*, *copy: bool = False*, *copy_obj: bool = False*, *path_split: OptStr = Ellipsis*, *node_types: OptStr = Ellipsis*, *list_insert: OptInt = Ellipsis*, *default_node_type: OptStr = Ellipsis*) → Collection[Any]

Merges two or more tree-objects to update and extend the root node

Parameters

- **obj** – tree-object that shall be merged. Can also be a FagusIterator returned from iter() to only merge values matching a filter defined in iter()
- **path** – position in root where the new objects shall be merged, default “”
- **new_value_action** – This parameter defines what merge is supposed to do if a value at a path is present in the root and in one of the objects to merge. The possible values are: (r)eplace - the value in the root is replaced with the new value, this is the default behaviour; (i)gnore - the value in the root is not updated; (a)ppend - the old and new value are both put into a list, and thus aggregated
- **extend_from** – By default, lists are traversed, so the value at index i will be compared in both lists. If at some point you rather want to just append the contents from the objects to be merged, use this parameter to define the level (count of keys) from which lists should be extended isf traversed. Default infinite (never extend lists)
- **update_from** – Like extend_from, but for dicts. Allows you to define at which level the contents of the root should just be updated with the contents of the objects instead of traversing and comparing each value
- **fagus** – whether the returned tree-object should be returned as Fagus
- **copy** – Don’t modify the root node, modify and return a copy instead
- **copy_obj** – The objects to be merged are not modified, but references to subnodes of the objects can be put into the root node. Set this to True to prevent that and keep root and objects independent
- **path_split** – * used to split path into a list if path is a str, default `" "`
- **node_types** – * Can be used to manually define if the nodes along path are supposed to be (l)ists or (d)icts. E.g. `"dll"` to create a dict at level 1, and lists at level 2 and 3. `" "` can also be used – space doesn’t enforce a node-type like `"d"` or `"l"`. For `" "`, existing nodes are traversed if possible, otherwise default_node_type is used to create new nodes. Default `""`, interpreted as ” ” at each level. See README
- **list_insert** – * Level at which a new node shall be inserted into the list instead of traversing the existing node in the list at that index. See README
- **default_node_type** – * determines if new nodes by default should be created as (d)ict or (l)ist. Must be either `"d"` or `"l"`, default `"d"`, examples in README

Returns

a reference to the modified root node, or a modified copy of the root node (see copy-parameter)

Raises

- **ValueError** – if it isn’t possible to parse an int-index from the provided key in a position where node-types defines that the node shall be a list (if node-types is not l, the node will be replaced with a dict)

- **TypeError** – if obj is not either a FagusIterator or a Collection. Also raised if you try to merge different types of nodes at root level, e.g. a dict can only be merged with another Mapping, and a list can only be merged with another Iterable. ~ is also raised if a not modifiable root node needs to be modified

pop(*path: Any = ''*, *default: OptAny = Ellipsis*, *fagus: OptBool = Ellipsis*, *path_split: OptStr = Ellipsis*) → Any

Deletes the value at path and returns it

* means that the parameter is a FagusOption, see Fagus-class-docstring for more information about options

Parameters

- **path** – pop value at this position in self, or don't do anything if path doesn't exist in self
- **default** – * returned if path doesn't exist in self
- **fagus** – * return the result as Fagus-object if possible (default is just to return the result)
- **path_split** – * used to split path into a list if path is a str, default `" "`

Returns
value at path if it exists, or default if it doesn't

Raises
TypeError – if the root node needs to be modified and isn't modifiable (e.g. tuple or frozenset)

popitem() → None

This function is not implemented in Fagus

Implementing this would require to cache the value, which was not prioritized to keep memory usage low.

discard(*path: Any = ''*, *path_split: OptStr = Ellipsis*) → None

Deletes the value at path if it exists

* means that the parameter is a FagusOption, see Fagus-class-docstring for more information about options

Parameters

- **path** – pop value at this position in self, or don't do anything if path doesn't exist in self
- **path_split** – * used to split path into a list if path is a str, default `" "`

Returns: None

remove(*path: Any = ''*, *path_split: OptStr = Ellipsis*) → None

Deletes the value at path if it exists, raises KeyError if it doesn't

* means that the parameter is a FagusOption, see Fagus-class-docstring for more information about options

Parameters

- **path** – pop value at this position in self, or don't do anything if path doesn't exist in self
- **path_split** – * used to split path into a list if path is a str, default `" "`

Returns: None

Raises
KeyError – if the value at path doesn't exist

keys(*path: Any = ''*, *path_split: OptStr = Ellipsis*) → Iterable[Any]

Returns keys for the node at path, or None if that node is a set or doesn't exist / doesn't have keys

* means that the parameter is a FagusOption, see Fagus-class-docstring for more information about options

Parameters

- **path** – get keys for node at this position in self. Default "" (gets values from the root node), See get()
- **path_split** – * used to split path into a list if path is a str, default " "

Returns

keys for the node at path, or an empty tuple if that node is a set or doesn't exist / doesn't have keys

values(*path: Any = ''*, *path_split: OptStr = Ellipsis*, *fagus: OptBool = Ellipsis*, *copy: bool = False*) → Iterable[Any]

Returns values for node at path

* means that the parameter is a FagusOption, see Fagus-class-docstring for more information about options

Parameters

- **path** – get values at this position in self, default "" (gets values from the root node). See get()
- **path_split** – * used to split path into a list if path is a str, default " "
- **fagus** – * converts sub-nodes into Fagus-objects in the returned list of values, default False
- **copy** – ~ creates a copy of the node before values() are returned. This can be beneficial if you want to make changes to the returned nodes, but you don't want to change self. Default False

Returns

values for the node at path. Returns an empty tuple if the value doesn't exist, or just the value in a tuple if the node isn't iterable.

items(*path: Any = ''*, *path_split: OptStr = Ellipsis*, *fagus: OptBool = Ellipsis*, *copy: bool = False*) → Iterable[Any]

Returns in iterator of (key, value)-tuples in self, like dict.items()

* means that the parameter is a FagusOption, see Fagus-class-docstring for more information about options

Parameters

- **path** – get items at this position in self, Default "" (gets values from the root node). See get()
- **path_split** – * used to split path into a list if path is a str, default " "
- **fagus** – * converts sub-nodes into Fagus-objects in the returned iterator, default False
- **copy** – ~ creates a copy of the node before items() are returned. This can be beneficial if you want to make changes to the returned nodes, but you don't want to change self. Default False

Returns

iterator of (key, value)-tuples in self, like dict.items()

`clear`(*path: Any = ''*, *path_split: OptStr = Ellipsis*, *copy: bool = False*, *fagus: OptBool = Ellipsis*) → Collection[Any]

Removes all elements from node at path.

* means that the parameter is a FagusOption, see Fagus-class-docstring for more information about options

Parameters

- **path** – clear at this position in self, Default `""` (gets values from the root node). See get()
- **path_split** – * used to split path into a list if path is a str, default `" "`
- **copy** – if ~ is set, a copy of self is modified and then returned (thus self is not modified), default False
- **fagus** – * return self as a Fagus-object if it is a node (tuple / list / dict), default False

Returns

self as a node if fagus is set, or a modified copy of self if copy is set

Raises

TypeError – if the root node needs to be modified and isn't modifiable (e.g. tuple or frozenset)

`contains`(*value: Any*, *path: Any = ''*, *path_split: OptStr = Ellipsis*) → bool

Check if value is present in the node at path

* means that the parameter is a FagusOption, see Fagus-class-docstring for more information about options

Parameters

- **value** – value to check
- **path** – check if value is in node at this position in self, Default `""` (checks root node). See get()
- **path_split** – * used to split path into a list if path is a str, default `" "`

Returns

whether value is in node at path in self. returns value == node if the node isn't iterable, and false if path doesn't exit in self

`count`(*path: Any = ''*, *path_split: OptStr = Ellipsis*) → int

Check the number of elements in the node at path

* means that the parameter is a FagusOption, see Fagus-class-docstring for more information about options

Parameters

- **path** – position in self where the number of elements shall be found.Default `""` (checks root node). See get() and README for examples
- **path_split** – * used to split path into a list if path is a str, default `" "`

Returns

the number of elements in the node at path. if there is no node at path, 0 is returned. If the element at path is not a node, 1 is returned

`index`(*value: Any*, *start: OptInt = Ellipsis*, *stop: OptInt = Ellipsis*, *path: Any = ''*, *all_: bool = False*, *path_split: OptStr = Ellipsis*) → Optional[Union[int, Any, Sequence[Any]]]

Returns the index / key of the specified value in the node at path if it exists

Parameters

- **value** – ~ to search index for
- **start** – start searching at this index. Only applicable if the node at path is a list / tuple
- **stop** – stop searching at this index. Only applicable if the node at path is a list / tuple
- **path** – position in self where the node shall be searched for value. Default "" (checks root node). See get() and README for examples
- **all_** – returns all matching indices / keys in a generator (instead of only the first)
- **path_split** – * used to split path into a list if path is a str, default " "

Returns

The first index of value if the node at path is a list, or the first key containing value if the node at path is a dict. True if the node at path is a Set and contains value. If the element can't be found in the node at path, or there is no Collection at path, None is returned (instead of a ValueError).

isdisjoint(*other: Iterable[Any]*, *path: Any = ''*, *path_split: OptStr = Ellipsis*, *dict_: str = 'keys'*) → bool

Returns whether the other iterable is disjoint (has no common items) with the node at path

* means that the parameter is a FagusOption, see Fagus-class-docstring for more information about options

Parameters

- **other** – other object to check
- **path** – check if the node at this position in self, is disjoint from other
- **path_split** – * used to split path into a list if path is a str, default " "
- **dict_** – use (k)eys, (v)alues or (i)tems for if value is a dict. Default keys

Returns: whether the other iterable is disjoint from the value at path. If value is a dict, the keys are used.

Checks if value is present in other if value isn't iterable. Returns True if there is no value at path.

child(*obj: Optional[Collection[Any]] = None*, ***kwargs*) → Fagus

Creates a Fagus-object for obj that has the same options as self

reversed(*path: Any = ''*, *fagus: OptBool = Ellipsis*, *path_split: OptStr = Ellipsis*, *copy: bool = False*) → Iterator[Any]

Get reversed child-node at path if that node is a list

* means that the parameter is a FagusOption, see Fagus-class-docstring for more information about options

Parameters

- **path** – position in self where a list / tuple shall be returned reversed
- **fagus** – * converts sub-nodes into Fagus-objects in the returned iterator, default False
- **path_split** – * used to split path into a list if path is a str, default " "
- **copy** – ~ creates a copy of the node before it is returned reversed(). This can be beneficial if you want to make changes to the returned nodes, but you don't want to change self. Default False

Returns
a reversed iterator on the node at path (empty if path doesn't exist)

`reverse`(*path: Any = ''*, *fagus: OptBool = Ellipsis*, *path_split: OptStr = Ellipsis*, *copy: bool = False*) → Collection[Any]

Reverse child-node at path if that node exists and is reversible

* means that the parameter is a FagusOption, see Fagus-class-docstring for more information about options

Parameters

- `path` – position in self where a list / tuple shall be reversed
- `fagus` – * converts sub-nodes into Fagus-objects in the returned iterator, default False
- `path_split` – * used to split path into a list if path is a str, default `" "`
- `copy` – ~ creates a copy of the node before it is returned reversed(). This can be beneficial if you want to make changes to the returned nodes, but you don't want to change self. Default False

Returns
self as a node if fagus is set, or a modified copy of self if copy is set

Raises
`TypeError` – if the root node needs to be modified and isn't modifiable (e.g. tuple or frozenset)

`copy`(*deep: bool = False*) → Collection[Any]

Creates a copy of self. Creates a recursive shallow copy by default, or a copy.deepcopy() if deep is set.

`options`(*options: Optional[Dict[str, Any]] = None*, *get_default_options: bool = False*, *reset: bool = False*) → Dict[str, Any]

Function to set multiple Fagus-options in one line

Parameters

- `options` – dict with options that shall be set
- `get_default_options` – return all options (include default-values). Default: only return options that are set
- `reset` – if ~ is set, all options are reset before options is set

Returns
a dict of options that are set, or all options if get_default_options is set

`__copy__`(*recursive: bool = False*) → Collection[Any]

Recursively creates a shallow-copy of self

`__call__`() → Collection[Any]

Calling the Fagus-object returns the root node the Fagus-object is wrapped around (equivalent to .root)

Example

```
>>> from fagus import Fagus
>>> a = Fagus({"f": "q"})
>>> a
Fagus({'f': 'q'})
>>> a()
{'f': 'q'}
>>> a.root  # .root returns the root-object in the same way as ()
{'f': 'q'}
```

Returns
the root object Fagus is wrapped around

__getattr__(*attr: str*) → Any

__getitem__(*item: Any*) → Any

__setattr__(*attr: str*, *value: Any*) → None
Implement setattr(self, name, value).

__setitem__(*path: Any*, *value: Any*) → None

__delattr__(*attr: str*) → None
Implement delattr(self, name).

__delitem__(*path: Any*) → None

__iter__() → Iterator[Any]

__hash__() → int
Return hash(self).

__eq__(*other: Any*) → bool
Return self==value.

__ne__(*other: Any*) → bool
Return self!=value.

__lt__(*other: Any*) → bool
Return self<value.

__le__(*other: Any*) → bool
Return self<=value.

__gt__(*other: Any*) → bool
Return self>value.

__ge__(*other: Any*) → bool
Return self>=value.

__contains__(*value: Any*) → bool

__len__() → int

__bool__() → bool

__repr__() → str
Return repr(self).

__str__() → str

Return str(self).

__iadd__(*value: Any*) → Collection[Any]

__add__(*other: Collection[Any]*) → Collection[Any]

__radd__(*other: Collection[Any]*) → Collection[Any]

__isub__(*other: Collection[Any]*) → Collection[Any]

__sub__(*other: Any*) → Collection[Any]

__rsub__(*other: Collection[Any]*) → Collection[Any]

__imul__(*times: int*) → Collection[Any]

```
__abstractmethods__ = frozenset({})
```

```
__annotations__ = {'_options':  'Optional[Dict[str, Any]]', 'root':
'Collection[Any]'}
```

```
__dict__ = mappingproxy({'__module__': 'fagus.fagus', '__annotations__':
{'root': 'Collection[Any]', '_options': 'Optional[Dict[str, Any]]'}, '__doc__':
'Fagus is a wrapper-class for complex, nested objects of dicts and lists in
Python\n\n Fagus can be used as an object by instantiating it, but it\'s also
possible to use all methods statically without\n even an object, so that ``a =
{}; Fagus.set(a, "top med", 1)`` and ``a = Fagus({}); a.set(1, "top med")`` do
the\n same.\n\n The root node is always modified directly. If you don\'t want to
change the root node, all the functions where it\n makes sense support to rather
modify a copy, and return that modified copy using the copy-parameter.\n\n
**FagusOptions**:\n Several parameters used in functions in Fagus work as options
so that you don\'t have to specify them each time you\n run a function. In the
docstrings, these options are marked with a \\*, e.g. the fagus parameter is an
option.\n Options can be specified at three levels with increasing precedence:
at class-level (``Fagus.fagus = True``), at\n object-level (``a = Fagus(),
a.fagus = True``) and in each function-call (``a.get("b", fagus=True)``). If
you\n generally want to change an option, change it at class-level - all objects
in that file will inherit this option.\n If you want to change the option
specifically for one object, change the option at object-level. If you only
want\n to change the option for one single run of a function, put it as a
function-parameter. More thorough examples of\n options can be found in
README.md.\n ', '__init__': <function Fagus.__init__>, 'get': <function
Fagus.get>, 'iter': <function Fagus.iter>, 'filter': <function Fagus.filter>,
'split': <function Fagus.split>, '_split_r': <staticmethod(<function
Fagus._split_r>)>, 'set': <function Fagus.set>, 'append': <function
Fagus.append>, 'extend': <function Fagus.extend>, 'insert': <function
Fagus.insert>, 'add': <function Fagus.add>, 'update': <function Fagus.update>,
'_build_node': <function Fagus._build_node>, '_put_value':
<staticmethod(<function Fagus._put_value>)>, 'setdefault': <function
Fagus.setdefault>, 'mod': <function Fagus.mod>, 'mod_all': <function
Fagus.mod_all>, 'serialize': <function Fagus.serialize>, '_serialize_r':
<staticmethod(<function Fagus._serialize_r>)>, '_serializable_value':
<staticmethod(<function Fagus._serializable_value>)>, 'merge': <function
Fagus.merge>, 'pop': <function Fagus.pop>, 'popitem': <function Fagus.popitem>,
'discard': <function Fagus.discard>, 'remove': <function Fagus.remove>, 'keys':
<function Fagus.keys>, 'values': <function Fagus.values>, 'items': <function
Fagus.items>, 'clear': <function Fagus.clear>, 'contains': <function
Fagus.contains>, 'count': <function Fagus.count>, 'index': <function
Fagus.index>, 'isdisjoint': <function Fagus.isdisjoint>, 'child': <function
Fagus.child>, 'reversed': <function Fagus.reversed>, 'reverse': <function
Fagus.reverse>, 'copy': <function Fagus.copy>, 'options': <function
Fagus.options>, '_opt': <function Fagus._opt>, '_ensure_mutable_node':
<staticmethod(<function Fagus._ensure_mutable_node>)>, '_get_mutable_node':
<function Fagus._get_mutable_node>, '_mutable_node_type':
<staticmethod(<function Fagus._mutable_node_type>)>, '_node_type':
<staticmethod(<function Fagus._node_type>)>, '_hash': <function Fagus._hash>,
'__copy__': <function Fagus.__copy__>, '__call__': <function Fagus.__call__>,
'__getattr__': <function Fagus.__getattr__>, '__getitem__': <function
Fagus.__getitem__>, '__setattr__': <function Fagus.__setattr__>, '__setitem__':
<function Fagus.__setitem__>, '__delattr__': <function Fagus.__delattr__>,
'__delitem__': <function Fagus.__delitem__>, '__iter__': <function
Fagus.__iter__>, '__hash__': <function Fagus.__hash__>, '__eq__': <function
Fagus.__eq__>, '__ne__': <function Fagus.__ne__>, '__lt__': <function
Fagus.__lt__>, '__le__': <function Fagus.__le__>, '__gt__': <function
Fagus.__gt__>, '__ge__': <function Fagus.__ge__>, '__contains__': <function
Fagus.__contains__>, '__len__': <function Fagus.__len__>, '__bool__': <function
Fagus.__bool__>, '__repr__': <function Fagus.__repr__>, '__str__': <function
Fagus.__str__>, '__iadd__': <function Fagus.__iadd__>, '__add__': <function
Fagus.__add__>, '__radd__': <function Fagus.__radd__>, '__isub__': <function
Fagus.__isub__>, '__sub__': <function Fagus.__sub__>, '__rsub__': <function
Fagus.__rsub__>, '__imul__': <function Fagus.__imul__>, '__mul__': <function
Fagus.__mul__>, '__rmul__': <function Fagus.__rmul__>, '__reversed__':
<function Fagus.__reversed__>, '__reduce__': <function Fagus.__reduce__>,
'__reduce_ex__': <function Fagus.__reduce_ex__>, '__dict__': <attribute
'__dict__' of 'Fagus' objects>, '__weakref__': <attribute '__weakref__' of
```

`__module__ = 'fagus.fagus'`

`__mul__`(*times: int*) → Union[Tuple[Any], List[Any]]

`__weakref__`
list of weak references to the object (if defined)

`__rmul__`(*times: int*) → Union[Tuple[Any], List[Any]]

`__reversed__`() → Iterator[Any]

`__reduce__`() → Union[str, Tuple[Any, ...]]
Helper for pickle.

`__reduce_ex__`(*protocol: Any*) → Union[str, Tuple[Any, ...]]
Helper for pickle.

class `fagus.Fil`(**filter_args: Any*, *inexclude: str = ''*, *str_as_re: bool = False*)

Bases: `KFil`

TFilter - what matches this filter will actually be visible in the result. See README

`__module__ = 'fagus.filters'`

class `fagus.CFil`(**filter_args: Any*, *inexclude: str = ''*, *str_as_re: bool = False*, *invert: bool = False*)

Bases: `KFil`

CFil - can be used to select nodes based on values that shall not appear in the result. See README

`__init__`(**filter_args: Any*, *inexclude: str = ''*, *str_as_re: bool = False*, *invert: bool = False*) → None
Initializes KeyFilter and verifies the arguments passed to it

Parameters

- **`*filter_args`** – Each argument filters one key in the tree, the last argument filters the leaf-value. You can put a list of values to match different values in the same filter. In this list, you can also specify subfilters to match different grains differently.
- **`inexclude`** – In some cases it's easier to specify that a filter shall match everything except b, rather than match a. ~ can be used to specify for each argument if the filter shall include it (+) or exclude it (-). Valid example: "++-+". If this parameter isn't specified, all args will be treated as (+).
- **`str_as_re`** – If this is set to True, it will be evaluated for all str's if they'd match differently as a regex, and in the latter case match these strings as regex patterns. E.g. re.match("a.*", b) will match differently than "a.*" == b. In this case, "a.*" will be used as a regex-pattern. However re.match("abc", b) will give the same result as "abc" == b, so here "abc" == b will be used.

Raises
`TypeError` – if the filters are not stacked correctly, or stacked in a way that doesn't make sense

`match_node`(*node: Collection[Any]*, *index: int = 0*) → bool
Recursive function to completely verify a node and its subnodes in CFil

Parameters

- **`node`** – node to check

- **index** – index in filter to check (filter is self)

Returns
bool whether the filter matched

__annotations__ = {}

__module__ = 'fagus.filters'

class fagus.VFil(**filter_args: Any, inexclude: str = '', invert: bool = False*)

Bases: FilBase

ValueFilter - This special type of filter can be used to inspect the entire node

It can be used to e.g. select all the nodes that contain at least 10 elements. See README for an example

__init__(**filter_args: Any, inexclude: str = '', invert: bool = False*) → None

Parameters

- ***filter_args** – Each argument filters one key in the tree, the last argument filters the leaf-value. You can put a list of values to match different values in the same filter. In this list, you can also specify subfilters to match different grains differently.
- **inexclude** – In some cases it's easier to specify that a filter shall match everything except b, rather than match a. ~ can be used to specify for each argument if the filter shall include it (+) or exclude it (-). Valid example: "++-+". If this parameter isn't specified, all args will be treated as (+).
- **invert** – Invert this whole filter to match if it doesn't match. E.g. if you want to select all the nodes that don't have a certain property.

match_node(*node: Collection[Any], _: Optional[Any] = None*) → bool

Verify that a node matches ValueFilter

Parameters

- **node** – node to check
- **_** – this argument is ignored

Returns
bool whether the filter matched

__annotations__ = {}

__module__ = 'fagus.filters'

3.1 Submodules

3.1.1 fagus.fagus module

Base-module that contains the Fagus-class

class fagus.fagus.Fagus(*root: Optional[Collection[Any]] = None, node_types: OptStr = Ellipsis, list_insert: OptInt = Ellipsis, path_split: OptStr = Ellipsis, fagus: OptBool = Ellipsis, default_node_type: OptStr = Ellipsis, default: OptAny = Ellipsis, if_: OptAny = Ellipsis, iter_fill: OptAny = Ellipsis, mod_functions: Union[Mapping[Union[type, Tuple[type], str], Callable[[Any], Any]], ellipsis] = Ellipsis, copy: bool = False*)

Bases: `MutableMapping`, `MutableSequence`, `MutableSet`

Fagus is a wrapper-class for complex, nested objects of dicts and lists in Python

Fagus can be used as an object by instantiating it, but it's also possible to use all methods statically without even an object, so that `a = {}; Fagus.set(a, "top med", 1)` and `a = Fagus({}); a.set(1, "top med")` do the same.

The root node is always modified directly. If you don't want to change the root node, all the functions where it makes sense support to rather modify a copy, and return that modified copy using the copy-parameter.

FagusOptions: Several parameters used in functions in Fagus work as options so that you don't have to specify them each time you run a function. In the docstrings, these options are marked with a *, e.g. the fagus parameter is an option. Options can be specified at three levels with increasing precedence: at class-level (`Fagus.fagus = True`), at object-level (`a = Fagus(), a.fagus = True`) and in each function-call (`a.get("b", fagus=True)`). If you generally want to change an option, change it at class-level - all objects in that file will inherit this option. If you want to change the option specifically for one object, change the option at object-level. If you only want to change the option for one single run of a function, put it as a function-parameter. More thorough examples of options can be found in README.md.

`__init__`(*root: Optional[Collection[Any]] = None*, *node_types: OptStr = Ellipsis*, *list_insert: OptInt = Ellipsis*, *path_split: OptStr = Ellipsis*, *fagus: OptBool = Ellipsis*, *default_node_type: OptStr = Ellipsis*, *default: OptAny = Ellipsis*, *if_: OptAny = Ellipsis*, *iter_fill: OptAny = Ellipsis*, *mod_functions: Union[Mapping[Union[type, Tuple[type], str], Callable[[Any], Any]], ellipsis] = Ellipsis*, *copy: bool = False*)

Constructor for Fagus, a wrapper-class for complex, nested objects of dicts and lists in Python

* means that the parameter is a FagusOption, see Fagus-class-docstring for more information about options

Parameters

- **`root`** – object (like dict / list) to wrap Fagus around. If this is None, an empty node of the type default_node_type will be used. Default None
- **`node_types`** – * Can be used to manually define if the nodes along path are supposed to be (l)ists or (d)icts. E.g. `"dll"` to create a dict at level 1, and lists at level 2 and 3. `" "` can also be used – space doesn't enforce a node-type like `"d"` or `"l"`. For `" "`, existing nodes are traversed if possible, otherwise default_node_type is used to create new nodes. Default `""`, interpreted as " " at each level. See README
- **`list_insert`** – * Level at which a new node shall be inserted into the list instead of traversing the existing node in the list at that index. See README
- **`path_split`** – * used to split path into a list if path is a string, default `" "`, see README
- **`fagus`** – * this option is used to determine whether nodes in the returned object should be returned as Fagus-objects. This can be useful e.g. if you want to use Fagus in an iteration. Check the particular function you want to use for a more thorough explanation of what this does in each case
- **`default_node_type`** – * determines if new nodes by default should be created as (d)ict or (l)ist. Must be either `"d"` or `"l"`, default `"d"`, examples in README
- **`default`** – * ~ is used in get and other functions if a path doesn't exist
- **`if_`** – * only set value if it meets the condition specified here, otherwise do nothing. The condition can be a lambda, any value or a tuple of accepted values. Default _None (don't check value)

- **iter_fill** – * Fill up tuples with iter_fill (can be any object, e.g. None) to ensure that all the tuples iter() returns are exactly max_items long. See iter()
- **mod_functions** – * ~ is used to define how different types of objects are supposed to be serialized. This is defined in a dict. The keys are either a type (like IPAddress) or a tuple of different types (IPv4Address, IPv6Address). The values are function pointers, or lambdas, which are supposed to convert e.g. an IPv4Address into a string. Check out TFunc if you want to call more complicated functions with several arguments. See README for examples
- **copy** – ~ creates a copy of the root node before Fagus is initialized. Makes sure that changes on this Fagus won't modify the root node that was passed here itself. Default False

`root: Collection[Any]`

Contains the root note the Fagus-object is wrapped around

This can be used to remove the Fagus-wrapper in case the plain object is needed, e.g. if `a = Fagus(["ex"])`, `a.root = ["ex"]`. The root node is also returned when a is called: `a()`, examples in `Fagus.__call__()`.

`get`(*path: Any = ''*, *default: OptAny = Ellipsis*, *fagus: OptBool = Ellipsis*, *copy: bool = False*, *path_split: OptStr = Ellipsis*) → Any

Retrieves value at path. If the value doesn't exist, default is returned.

To get `"hello"` from `x = Fagus({"a": ["b", {"c": "d"}], e: ["f", "g"]})`, you can use `x[("a", 1, "c")]`. The tuple `("a", 1, "c")` is the path-parameter that is used to traverse x. At first, the list at `"a"` is picked in the top-most dict, and then the 2nd element `{"c": "d"}` is picked from that list. Then, "d" is picked from `{"c": "d"}` and returned. The path-parameter can be a tuple or list, the keys must be either integers for lists, or any hashable objects for dicts. For convenience, the keys can also be put in a single string separated by path_split (default `" "`), so `a["a 1 c"]` also returns `"d"`.

* means that the parameter is a FagusOption, see Fagus-class-docstring for more information about options

Parameters

- **path** – List/Tuple of key-values to recursively traverse self. Can also be specified as string, that is split into a tuple using path_split
- **default** – * returned if path doesn't exist in self
- **fagus** – * returns a Fagus-object if the value at path is a list or dict
- **copy** – Option to return a copy of the returned value. The default behaviour is that if there are subnodes (dicts, lists) in the returned values, and you make changes to these nodes, these changes will also be applied in the root node from which values() was called. If you want the returned values to be independent, use copy to get a shallow copy of the returned value
- **path_split** – * used to split path into a list if path is a str, default `" "`

Returns

the value if the path exists, or default if it doesn't exist

`iter`(*max_depth: int = 9223372036854775807*, *path: Any = ''*, *filter_: Optional[Fil] = None*, *fagus: OptBool = Ellipsis*, *iter_fill: OptAny = Ellipsis*, *select: Optional[Union[int, Iterable[Any]]] = None*, *copy: bool = False*, *iter_nodes: OptBool = Ellipsis*, *filter_ends: bool = False*, *path_split: OptStr = Ellipsis*) → FagusIterator

Recursively iterate through Fagus-object, starting at path

* means that the parameter is a FagusOption, see Fagus-class-docstring for more information about options

Parameters

- **max_depth** – Can be used to limit how deep the iteration goes. Example: `a = {"a": ["b", ["c", "d"]], "e": "f"}` If max_depth is sys.max_size, all the nodes are traversed: `[("a", "b", "c"), ("a", "b", "d"]), ("e", "f")]`. If max_depth is 1, iter returns `[("a", "b", ["c", "d"]), ("e", "f")]`, so `["c", "d"]` is not iterated through but returned as a node. If max_depth is 0, iter returns `[("a", ["b", ["c", "d"]]), ("e", "f")]`, effectively the same as dict.items(). Default sys.maxitems (iterate as deeply as possible). A negative number (e.g. -1) is treated as sys.maxitems.
- **path** – Start iterating at path. Internally calls get(path), and iterates on the node get returns. See get()
- **filter_** – Only iterate over specific nodes defined using Fil (see README.md and Fil for more info)
- **fagus** – * If the leaf in the tuple is a dict or list, return it as a Fagus-object. This option has no effect if max_items is sys.maxitems.
- **iter_fill** – * Fill up tuples with iter_fill (can be any object, e.g. None) to ensure that all the tuples iter() returns are exactly max_items long. This can be useful if you want to unpack the keys / leaves from the tuples in a loop, which fails if the count of items in the tuples varies. This option has no effect if max_items is -1. The default value is ..., meaning that the tuples are not filled, and the length of the tuples can vary. See README for a more thorough example.
- **select** – Extract only some specified values from the tuples. E.g. if ~ is -1, only the leaf-values are returned. ~ can also be a list of indices. Default None (don't reduce the tuples)
- **copy** – Iterate on a shallow-copy to make sure that you can edit root node without disturbing the iteration
- **iter_nodes** – * includes the traversed nodes into the resulting tuples, order is then: node1, key1, node2, key2, ..., leaf_value
- **filter_ends** – Affects the end dict/list that is returned if max_items is used. Normally, filters are not applied on that end node. If you would like to get the end node filtered too, set this to True. If this is set to True, the last nodes will always be copies (if unfiltered they are references)
- **path_split** – * used to split path into a list if path is a str, default `" "`, see README

Returns

FagusIterator with one tuple for each leaf-node, containing the keys of the parent-nodes until the leaf

filter(*filter_: Fil*, *path: Any = ''*, *fagus: OptBool = Ellipsis*, *copy: bool = False*, *default: OptAny = Ellipsis*, *path_split: OptStr = Ellipsis*) → Collection[Any]

Filters self, only keeping the nodes that pass the filter

* means that the parameter is a FagusOption, see Fagus-class-docstring for more information about options

Parameters

- **filter_** – Fil-object in which the filtering-criteria are specified
- **path** – at this point in self, the filtering will start (apply filter_ relatively from this point). Default `""`, meaning that the root node is filtered, see get() and README for examples
- **fagus** – * return the filtered self as Fagus-object (default is just to return the filtered node)

- **copy** – Create a copy and filter on that copy. Default is to modify the self directly
- **default** – * returned if path doesn't exist in self, or the value at path can't be filtered
- **path_split** – * used to split path into a list if path is a string, default `" "`, see README

Returns
the filtered object, starting at path

Raises
TypeError – if the root node needs to be modified and isn't modifiable (e.g. tuple or frozenset)

split(*filter_: Optional[Fil]*, *path: Any = ''*, *fagus: OptBool = Ellipsis*, *copy: bool = False*, *default: OptAny = Ellipsis*, *path_split: OptStr = Ellipsis*) → Union[Tuple[Collection[Any], Collection[Any]], Tuple[Any, Any]]

Splits self into nodes that pass the filter, and nodes that don't pass the filter

* means that the parameter is a FagusOption, see Fagus-class-docstring for more information about options

Parameters

- **filter_** – Fil-object in which the filtering-criteria are specified
- **path** – at this position in self, the splitting will start (apply filter_ relatively from this point). Default `""`, meaning that the root node is split, see get() and README for examples
- **fagus** – * return the filtered self as Fagus-object (default is just to return the filtered node)
- **copy** – Create a copy and filter on that copy. Default is to modify the object directly
- **default** – * returned if path doesn't exist in self
- **path_split** – * used to split path into a list if path is a str, default `" "`

Returns
a tuple, where the first element is the nodes that pass the filter, and the second element is the nodes that don't pass the filter

Raises
TypeError – if the root node needs to be modified and isn't modifiable (e.g. tuple or frozenset)

set(*value: Any*, *path: Iterable[Any]*, *node_types: OptStr = Ellipsis*, *list_insert: OptInt = Ellipsis*, *path_split: OptStr = Ellipsis*, *fagus: OptBool = Ellipsis*, *if_: OptAny = Ellipsis*, *default_node_type: OptStr = Ellipsis*, *copy: bool = False*) → Collection[Any]

Create (if they don't already exist) all sub-nodes in path, and finally set value at leaf-node

* means that the parameter is a FagusOption, see Fagus-class-docstring for more information about options

Parameters

- **value** – ~ is placed at path, after creating new nodes if necessary. An existing value at path is overwritten
- **path** – List/Tuple of key-values that are traversed in self. If no nodes exist at the keys, new nodes are created. Can also be specified as a string, that is split into a tuple using path_split. See get()

- **node_types** – * Can be used to manually define if the nodes along path are supposed to be (l)ists or (d)icts. E.g. `"dll"` to create a dict at level 1, and lists at level 2 and 3. `" "` can also be used – space doesn't enforce a node-type like `"d"` or `"l"`. For `" "`, existing nodes are traversed if possible, otherwise default_node_type is used to create new nodes. Default `""`, interpreted as `" "` at each level. See README
- **list_insert** – * Level at which a new node shall be inserted into the list instead of traversing the existing node in the list at that index. See README
- **path_split** – * used to split path into a list if path is a string, default `" "`, see README
- **fagus** – * return self as a Fagus-object if it is a node (tuple / list / dict), default False
- **if_** – * only set value if it meets the condition specified here, otherwise do nothing. The condition can be a lambda, any value or a tuple of accepted values. Default _None (don't check value)
- **default_node_type** – * determines if new nodes by default should be created as (d)ict or (l)ist. Must be either `"d"` or `"l"`, default `"d"`, examples in README
- **copy** – if this is set, a copy of self is modified and then returned (thus self is not modified)

Returns

self as a node if fagus is set, or a modified copy of self if copy is set

Raises

- **ValueError** – if it isn't possible to parse an int-index from the provided key in a position where node-types defines that the node shall be a list (if node-types is not l, the node will be replaced with a dict)
- **TypeError** – if the root node needs to be modified and isn't modifiable (e.g. tuple or frozenset)

append(*value: Any*, *path: Any = ''*, *node_types: OptStr = Ellipsis*, *list_insert: OptInt = Ellipsis*, *path_split: OptStr = Ellipsis*, *fagus: OptBool = Ellipsis*, *if_: OptAny = Ellipsis*, *default_node_type: OptStr = Ellipsis*, *copy: bool = False*) → Collection[Any]

Create (if they don't already exist) all sub-nodes in path, and finally append value to a list at leaf-node

If the leaf-node is a set, tuple or other value it is converted to a list. Then the new value is appended.

* means that the parameter is a FagusOption, see Fagus-class-docstring for more information about options

Parameters

- **value** – ~ is appended to list at path, after creating new nodes along path as necessary
- **path** – List/Tuple of key-values that are traversed in self. If no nodes exist at the keys, new nodes are created. Can also be specified as a string, that is split into a tuple using path_split. See get()
- **node_types** – * Can be used to manually define if the nodes along path are supposed to be (l)ists or (d)icts. E.g. `"dll"` to create a dict at level 1, and lists at level 2 and 3. `" "` can also be used – space doesn't enforce a node-type like `"d"` or `"l"`. For `" "`, existing nodes are traversed if possible, otherwise default_node_type is used to create new nodes. Default `""`, interpreted as `" "` at each level. See README

- **list_insert** – * Level at which a new node shall be inserted into the list instead of traversing the existing node in the list at that index. See README
- **path_split** – * used to split path into a list if path is a string, default " ", see README
- **fagus** – * return self as a Fagus-object if it is a node (tuple / list / dict), default False
- **if_** – * only append value if it meets the condition specified here, otherwise do nothing. The condition can be a lambda, any value or a tuple of accepted values. Default _None (don't check value)
- **default_node_type** – * determines if new nodes by default should be created as (d)ict or (l)ist. Must be either "d" or "l", default "d", examples in README
- **copy** – if this is set, a copy of self is modified and then returned (thus self is not modified)

Returns
self as a node if fagus is set, or a modified copy of self if copy is set

Raises

- **ValueError** – if it isn't possible to parse an int-index from the provided key in a position where node-types defines that the node shall be a list (if node-types is not l, the node will be replaced with a dict)
- **TypeError** – if path is empty and the root node is not a list (can't append to a dict, tuple or set) or the root node needs to be modified and isn't modifiable (e.g. tuple or frozenset)

extend(*values: Iterable[Any]*, *path: Any = ''*, *node_types: OptStr = Ellipsis*, *list_insert: OptInt = Ellipsis*, *path_split: OptStr = Ellipsis*, *fagus: OptBool = Ellipsis*, *if_: OptAny = Ellipsis*, *default_node_type: OptStr = Ellipsis*, *copy: bool = False*) → Collection[Any]

Create (if they don't already exist) all sub-nodes in path. Then extend list at leaf-node with the new values

If the leaf-node is a set, tuple or other value it is converted to a list, which is extended with the new values

* means that the parameter is a FagusOption, see Fagus-class-docstring for more information about options

Parameters

- **values** – the list at path is extended with ~, after creating new nodes along path as necessary
- **path** – List/Tuple of key-values that are traversed in self. If no nodes exist at the keys, new nodes are created. Can also be specified as a string, that is split into a tuple using path_split. See get()
- **node_types** – * Can be used to manually define if the nodes along path are supposed to be (l)ists or (d)icts. E.g. "dll" to create a dict at level 1, and lists at level 2 and 3. " " can also be used – space doesn't enforce a node-type like "d" or "l". For " ", existing nodes are traversed if possible, otherwise default_node_type is used to create new nodes. Default "", interpreted as " " at each level. See README
- **list_insert** – * Level at which a new node shall be inserted into the list instead of traversing the existing node in the list at that index. See README
- **path_split** – * used to split path into a list if path is a string, default " ", see README

- **fagus** – * return self as a Fagus-object if it is a node (tuple / list / dict), default False
- **if_** – * only extend with values if they meet the condition specified here, otherwise do nothing. The condition can be a lambda, any value or a tuple of accepted values. Default _None (don't check values)
- **default_node_type** – * determines if new nodes by default should be created as (d)ict or (l)ist. Must be either "d" or "l", default "d", examples in README
- **copy** – if this is set, a copy of self is modified and then returned (thus self is not modified)

Returns

self as a node if fagus is set, or a modified copy of self if copy is set

Raises

- **ValueError** – if it isn't possible to parse an int-index from the provided key in a position where node-types defines that the node shall be a list (if node-types is not l, the node will be replaced with a dict)
- **TypeError** – if path is empty and the root node is not a list (can't extend a dict, tuple or set) or the root node needs to be modified and isn't modifiable (e.g. tuple or frozenset)

insert(*index: int*, *value: Any*, *path: Any = ''*, *node_types: OptStr = Ellipsis*, *list_insert: OptInt = Ellipsis*, *path_split: OptStr = Ellipsis*, *fagus: OptBool = Ellipsis*, *if_: OptAny = Ellipsis*, *default_node_type: OptStr = Ellipsis*, *copy: bool = False*) → Collection[Any]

Create (if they don't already exist) all sub-nodes in path. Insert new value at index in list at leaf-node

If the leaf-node is a set, tuple or other value it is converted to a list, in which the new value is inserted at index

* means that the parameter is a FagusOption, see Fagus-class-docstring for more information about options

Parameters

- **index** – ~ at which the value shall be inserted in the list at path
- **value** – ~ is inserted at index into list at path, after creating new nodes along path as necessary
- **path** – List/Tuple of key-values that are traversed in self. If no nodes exist at the keys, new nodes are created. Can also be specified as a string, that is split into a tuple using path_split. See get()
- **node_types** – * Can be used to manually define if the nodes along path are supposed to be (l)ists or (d)icts. E.g. "dll" to create a dict at level 1, and lists at level 2 and 3. " " can also be used – space doesn't enforce a node-type like "d" or "l". For " ", existing nodes are traversed if possible, otherwise default_node_type is used to create new nodes. Default "", interpreted as " " at each level. See README
- **list_insert** – * Level at which a new node shall be inserted into the list instead of traversing the existing node in the list at that index. See README
- **path_split** – * used to split path into a list if path is a string, default " ", see README
- **fagus** – * return self as a Fagus-object if it is a node (tuple / list / dict), default False

- **if_** – * only insert value if it meets the condition specified here, otherwise do nothing. The condition can be a lambda, any value or a tuple of accepted values. Default _None (don't check value)
- **default_node_type** – * determines if new nodes by default should be created as (d)ict or (l)ist. Must be either "d" or "l", default "d", examples in README
- **copy** – if this is set, a copy of self is modified and then returned (thus self is not modified)

Returns
self as a node if fagus is set, or a modified copy of self if copy is set

Raises

- **ValueError** – if it isn't possible to parse an int-index from the provided key in a position where node-types defines that the node shall be a list (if node-types is not l, the node will be replaced with a dict)
- **TypeError** – if path is empty and the root node is not a list (can't insert into dict, tuple or set) or the root node needs to be modified and isn't modifiable (e.g. tuple or frozenset)

add(*value: Any*, *path: Any = ''*, *node_types: OptStr = Ellipsis*, *list_insert: OptInt = Ellipsis*, *path_split: OptStr = Ellipsis*, *fagus: OptBool = Ellipsis*, *if_: OptAny = Ellipsis*, *default_node_type: OptStr = Ellipsis*, *copy: bool = False*) → Collection[Any]

Create (if they don't already exist) all sub-nodes in path, and finally add new value to set at leaf-node

If the leaf-node is a list, tuple or other value it is converted to a set, to which the new value is added

* means that the parameter is a FagusOption, see Fagus-class-docstring for more information about options

Parameters

- **value** – ~ is added to set at path, after creating new nodes along path as necessary
- **path** – List/Tuple of key-values that are traversed in self. If no nodes exist at the keys, new nodes are created. Can also be specified as a string, that is split into a tuple using path_split. See get()
- **node_types** – * Can be used to manually define if the nodes along path are supposed to be (l)ists or (d)icts. E.g. "dll" to create a dict at level 1, and lists at level 2 and 3. " " can also be used – space doesn't enforce a node-type like "d" or "l". For " ", existing nodes are traversed if possible, otherwise default_node_type is used to create new nodes. Default "", interpreted as " " at each level. See README
- **list_insert** – * Level at which a new node shall be inserted into the list instead of traversing the existing node in the list at that index. See README
- **path_split** – * used to split path into a list if path is a string, default " ", see README
- **fagus** – * return self as a Fagus-object if it is a node (tuple / list / dict), default False
- **if_** – * only add value if it meets the condition specified here, otherwise do nothing. The condition can be a lambda, any value or a tuple of accepted values. Default _None (don't check value)
- **default_node_type** – * determines if new nodes by default should be created as (d)ict or (l)ist. Must be either "d" or "l", default "d", examples in README

- **copy** – if this is set, a copy of self is modified and then returned (thus self is not modified)

Returns
self as a node if fagus is set, or a modified copy of self if copy is set

Raises

- **ValueError** – if it isn't possible to parse an int-index from the provided key in a position where node-types defines that the node shall be a list (if node-types is not l, the node will be replaced with a dict)
- **TypeError** – if path is empty and the root node is not a set (can't add to list or dict) or the root node needs to be modified and isn't modifiable (e.g. tuple or frozenset)

update(*values: Iterable[Any]*, *path: Any = ''*, *node_types: OptStr = Ellipsis*, *list_insert: OptInt = Ellipsis*, *path_split: OptStr = Ellipsis*, *fagus: OptBool = Ellipsis*, *if_: OptAny = Ellipsis*, *default_node_type: OptStr = Ellipsis*, *copy: bool = False*) → Collection[Any]

Create (if they don't already exist) all sub-nodes in path, then update set at leaf-node with new values

If the leaf-node is a list, tuple or other value it is converted to a set. That set is then updated with the new values. If the node at path is a dict, and values also is a dict, the node-dict is updated with the new values.

* means that the parameter is a FagusOption, see Fagus-class-docstring for more information about options

Parameters

- **values** – the set/dict at path is updated with ~, after creating new nodes along path as necessary
- **path** – List/Tuple of key-values that are traversed in self. If no nodes exist at the keys, new nodes are created. Can also be specified as a string, that is split into a tuple using path_split. See get()
- **node_types** – * Can be used to manually define if the nodes along path are supposed to be (l)ists or (d)icts. E.g. `"dll"` to create a dict at level 1, and lists at level 2 and 3. `" "` can also be used – space doesn't enforce a node-type like `"d"` or `"l"`. For `" "`, existing nodes are traversed if possible, otherwise default_node_type is used to create new nodes. Default `""`, interpreted as " " at each level. See README
- **list_insert** – * Level at which a new node shall be inserted into the list instead of traversing the existing node in the list at that index. See README
- **path_split** – * used to split path into a list if path is a string, default `" "`, see README
- **fagus** – * return self as a Fagus-object if it is a node (tuple / list / dict), default False
- **if_** – * only update with values if they meet the condition specified here, otherwise do nothing. The condition can be a lambda, any value or a tuple of accepted values. Default _None (don't check values)
- **default_node_type** – * determines if new nodes by default should be created as (d)ict or (l)ist. Must be either `"d"` or `"l"`, default `"d"`, examples in README
- **copy** – if this is set, a copy of self is modified and then returned (thus self is not modified)

Returns
self as a node if fagus is set, or a modified copy of self if copy is set

Raises

- **ValueError** – if it isn't possible to parse an int-index from the provided key in a position where node-types defines that the node shall be a list (if node-types is not l, the node will be replaced with a dict)
- **TypeError** – if path is empty and the root node is not a set or dict (can't update list) or the root node needs to be modified and isn't modifiable (e.g. tuple or frozenset)

setdefault(*path: Any = ''*, *default: OptAny = Ellipsis*, *fagus: OptBool = Ellipsis*, *node_types: OptStr = Ellipsis*, *list_insert: OptInt = Ellipsis*, *path_split: OptStr = Ellipsis*, *default_node_type: OptStr = Ellipsis*) → Any

Get value at path and return it. If there is no value at path, set default at path, and return default

* means that the parameter is a FagusOption, see Fagus-class-docstring for more information about options

Parameters

- **path** – position in self where default shall be set / from where value shall be fetched. See get() and README
- **default** – * returned if path doesn't exist in self
- **fagus** – * return self as a Fagus-object if it is a node (tuple / list / dict), default False
- **node_types** – * Can be used to manually define if the nodes along path are supposed to be (l)ists or (d)icts. E.g. `"dll"` to create a dict at level 1, and lists at level 2 and 3. `" "` can also be used – space doesn't enforce a node-type like `"d"` or `"l"`. For `" "`, existing nodes are traversed if possible, otherwise default_node_type is used to create new nodes. Default `""`, interpreted as " " at each level. See README
- **list_insert** – * Level at which a new node shall be inserted into the list instead of traversing the existing node in the list at that index. See README
- **path_split** – * used to split path into a list if path is a str, default `" "`
- **default_node_type** – * determines if new nodes by default should be created as (d)ict or (l)ist. Must be either `"d"` or `"l"`, default `"d"`, examples in README

Returns

value at path if it exists, otherwise default is set at path and returned

Raises

- **ValueError** – if it isn't possible to parse an int-index from the provided key in a position where node-types defines that the node shall be a list (if node-types is not l, the node will be replaced with a dict)
- **TypeError** – if the root node needs to be modified and isn't modifiable (e.g. tuple or frozenset)

mod(*mod_function: Callable[[Any], Any]*, *path: Iterable[Any]*, *default: OptAny = Ellipsis*, *replace_value: bool = True*, *fagus: OptBool = Ellipsis*, *node_types: OptStr = Ellipsis*, *list_insert: OptInt = Ellipsis*, *path_split: OptStr = Ellipsis*, *default_node_type: OptStr = Ellipsis*) → Any

Modifies the value at path using the function-pointer mod_function

mod can be used like this Fagus.mod(obj, "kitchen spoon", lambda x: x + 1, 1) to count the number of spoons in the kitchen. If there is no value to modify, the default value (here 1) will be set at the node.

* means that the parameter is a FagusOption, see Fagus-class-docstring for more information about options

Parameters

- **mod_function** – A function pointer or lambda that modifies the existing value at path. TFunc can be used to call more complex functions requiring several arguments.
- **path** – position in self at which the value shall be modified. Defined as a list/Tuple of key-values to recursively traverse self. Can also be specified as string which is split into a tuple using path_split
- **default** – * this value is set in path if it doesn't exist
- **fagus** – * Return new value as a Fagus-object if it is a node (tuple / list / dict), default False
- **replace_value** – Replace the old value with what mod_function returns. Can be deactivated e.g. if mod_function changes the object, but returns None (if ~ stays on, the object is replaced with None). Default True. If no value exists at path, the default value is always set at path (independent of ~)
- **node_types** – * Can be used to manually define if the nodes along path are supposed to be (l)ists or (d)icts. E.g. `"dll"` to create a dict at level 1, and lists at level 2 and 3. `" "` can also be used – space doesn't enforce a node-type like `"d"` or `"l"`. For `" "`, existing nodes are traversed if possible, otherwise default_node_type is used to create new nodes. Default `""`, interpreted as " " at each level. See README
- **list_insert** – * Level at which a new node shall be inserted into the list instead of traversing the existing node in the list at that index. See README
- **path_split** – * used to split path into a list if path is a str, default `" "`
- **default_node_type** – * determines if new nodes by default should be created as (d)ict or (l)ist. Must be either `"d"` or `"l"`, default `"d"`, examples in README

Returns

the new value that was returned by the mod_function, or default if there was no value at path

Raises

- **ValueError** – if it isn't possible to parse an int-index from the provided key in a position where node-types defines that the node shall be a list (if node-types is not l, the node will be replaced with a dict)
- **TypeError** – if the root node needs to be modified and isn't modifiable (e.g. tuple or frozenset)

mod_all(*mod_function: Callable[[Any], Any]*, *filter_: Optional[Fil] = None*, *path: Any = ''*, *replace_value: bool = True*, *default: OptAny = Ellipsis*, *max_depth: int = 922337203685477580*7, *fagus: OptBool = Ellipsis*, *copy: bool = False*, *path_split: OptStr = Ellipsis*) → Any

Modify all the leaf-values that match a certain filter

* means that the parameter is a FagusOption, see Fagus-class-docstring for more information about options

Parameters

- **mod_function** – A function pointer or lambda that modifies the existing value at path. TFunc can be used to call more complex functions requiring several arguments.

- **filter_** – used to select which leaves shall be modified. Default None (all leaves are modified)
- **path** – position in self at which the value shall be modified. See get() / README
- **default** – * this value is returned if path doesn't exist, or if no leaves match the filter
- **fagus** – * Return new value as a Fagus-object if it is a node (tuple / list / dict), default False
- **replace_value** – Replace the old value with what mod_function returns. Can be deactivated e.g. if mod_function changes the object, but returns None (if ~ stays on, the object is replaced with None). Default True. If no value exists at path, the default value is always set at path (independent of ~)
- **max_depth** – Defines the maximum depth for the iteration. See Fagus.iter max_depth for more information
- **copy** – Can be ued to make sure that the node at path is not modified (instead a modified copy is returned)
- **path_split** – * used to split path into a list if path is a str, default " "

Returns

the node at path where all the leaves matching filter_ are modified, or default if it didn't exist

Raises

TypeError – if the root node needs to be modified and isn't modifiable (e.g. tuple or frozenset)

serialize(*mod_functions: Optional[Mapping[Union[type, Tuple[type], str], Callable[[Any], Any]]] = None*, *path: Any = ''*, *node_types: OptStr = Ellipsis*, *list_insert: OptInt = Ellipsis*, *path_split: OptStr = Ellipsis*, *copy: bool = False*) → Union[Dict[Any, Any], List[Any]]

Makes sure the object can be serialized so that it can be converted to JSON, YAML etc.

The only allowed data-types for serialization are: dict, list, bool, float, int, str, None

Sets and tuples are converted into lists. Other objects whose types are not allowed in serialized objects are modified to a type that is allowed using the mod_functions-parameter. mod_functions is a dict, with the type of object like IPv4Address or a tuple of types like (IPv4Address, IPv6Address). The values are function pointers or lambdas, that are executed to convert e.g. an IPv4Address to one of the allowed data types mentioned above.

The default mod_functions are: {datetime: lambda x: x.isoformat(), date: lambda x: x.isoformat(), time: lambda x: x.isoformat(), "default": lambda x: str(x)}

By default, date, datetime and time-objects are replaced by their isoformat-string. All other objects whose types don't appear in mod_functions are modified by the function behind the key "default". By default, this function is lambda x: str(x) that replaces the object with its string-representation.

* means that the parameter is a FagusOption, see Fagus-class-docstring for more information about options

Parameters

- **mod_functions** – * ~ is used to define how different types of objects are supposed to be serialized. This is defined in a dict. The keys are either a type (like IPAddress) or a tuple of different types (IPv4Address, IPv6Address). The values are function pointers, or lambdas, which are supposed to convert e.g. an IPv4Address into a string. Check out TFunc if you want to call more complicated functions with several arguments. See README for examples

- **path** – position in self at which the value shall be modified. See get() / README
- **node_types** – * Can be used to manually define if the nodes along path are supposed to be (l)ists or (d)icts. E.g. `"dll"` to create a dict at level 1, and lists at level 2 and 3. `" "` can also be used – space doesn't enforce a node-type like `"d"` or `"l"`. For `" "`, existing nodes are traversed if possible, otherwise default_node_type is used to create new nodes. Default `""`, interpreted as `" "` at each level. See README
- **list_insert** – * Level at which a new node shall be inserted into the list instead of traversing the existing node in the list at that index. See README
- **path_split** – * used to split path into a list if path is a str, default `" "`
- **copy** – Create a copy and make that copy serializable. Default is to modify self directly

Returns

a serializable object that only contains types allowed in json or yaml

Raises

- **TypeError** – if root node is not a dict or list (serialize can't fix that for the root node)
- **ValueError** – if tuple_keys is not defined in mod_functions and a dict has tuples as keys
- **Exception** – Can raise any exception if it occurs in one of the mod_functions

merge(*obj: Union[FagusIterator, Collection[Any]]*, *path: Any = ''*, *new_value_action: str = 'r'*, *extend_from: int = 9223372036854775807*, *update_from: int = 9223372036854775807*, *fagus: OptBool = Ellipsis*, *copy: bool = False*, *copy_obj: bool = False*, *path_split: OptStr = Ellipsis*, *node_types: OptStr = Ellipsis*, *list_insert: OptInt = Ellipsis*, *default_node_type: OptStr = Ellipsis*) → Collection[Any]

Merges two or more tree-objects to update and extend the root node

Parameters

- **obj** – tree-object that shall be merged. Can also be a FagusIterator returned from iter() to only merge values matching a filter defined in iter()
- **path** – position in root where the new objects shall be merged, default ""
- **new_value_action** – This parameter defines what merge is supposed to do if a value at a path is present in the root and in one of the objects to merge. The possible values are: (r)eplace - the value in the root is replaced with the new value, this is the default behaviour; (i)gnore - the value in the root is not updated; (a)ppend - the old and new value are both put into a list, and thus aggregated
- **extend_from** – By default, lists are traversed, so the value at index i will be compared in both lists. If at some point you rather want to just append the contents from the objects to be merged, use this parameter to define the level (count of keys) from which lists should be extended isf traversed. Default infinite (never extend lists)
- **update_from** – Like extend_from, but for dicts. Allows you to define at which level the contents of the root should just be updated with the contents of the objects instead of traversing and comparing each value
- **fagus** – whether the returned tree-object should be returned as Fagus
- **copy** – Don't modify the root node, modify and return a copy instead

- **copy_obj** – The objects to be merged are not modified, but references to subnodes of the objects can be put into the root node. Set this to True to prevent that and keep root and objects independent
- **path_split** – * used to split path into a list if path is a str, default " "
- **node_types** – * Can be used to manually define if the nodes along path are supposed to be (l)ists or (d)icts. E.g. "dll" to create a dict at level 1, and lists at level 2 and 3. " " can also be used – space doesn't enforce a node-type like "d" or "l". For " ", existing nodes are traversed if possible, otherwise default_node_type is used to create new nodes. Default "", interpreted as " " at each level. See README
- **list_insert** – * Level at which a new node shall be inserted into the list instead of traversing the existing node in the list at that index. See README
- **default_node_type** – * determines if new nodes by default should be created as (d)ict or (l)ist. Must be either "d" or "l", default "d", examples in README

Returns
a reference to the modified root node, or a modified copy of the root node (see copy-parameter)

Raises

- **ValueError** – if it isn't possible to parse an int-index from the provided key in a position where node-types defines that the node shall be a list (if node-types is not l, the node will be replaced with a dict)
- **TypeError** – if obj is not either a FagusIterator or a Collection. Also raised if you try to merge different types of nodes at root level, e.g. a dict can only be merged with another Mapping, and a list can only be merged with another Iterable. ~ is also raised if a not modifiable root node needs to be modified

pop(*path: Any = ''*, *default: OptAny = Ellipsis*, *fagus: OptBool = Ellipsis*, *path_split: OptStr = Ellipsis*) → Any

Deletes the value at path and returns it

* means that the parameter is a FagusOption, see Fagus-class-docstring for more information about options

Parameters

- **path** – pop value at this position in self, or don't do anything if path doesn't exist in self
- **default** – * returned if path doesn't exist in self
- **fagus** – * return the result as Fagus-object if possible (default is just to return the result)
- **path_split** – * used to split path into a list if path is a str, default " "

Returns
value at path if it exists, or default if it doesn't

Raises
TypeError – if the root node needs to be modified and isn't modifiable (e.g. tuple or frozenset)

popitem() → None

This function is not implemented in Fagus

Implementing this would require to cache the value, which was not prioritized to keep memory usage low.

`discard`(*path: Any = ''*, *path_split: OptStr = Ellipsis*) → None

Deletes the value at path if it exists

* means that the parameter is a FagusOption, see Fagus-class-docstring for more information about options

Parameters

- **path** – pop value at this position in self, or don't do anything if path doesn't exist in self
- **path_split** – * used to split path into a list if path is a str, default `" "`

Returns: None

`remove`(*path: Any = ''*, *path_split: OptStr = Ellipsis*) → None

Deletes the value at path if it exists, raises KeyError if it doesn't

* means that the parameter is a FagusOption, see Fagus-class-docstring for more information about options

Parameters

- **path** – pop value at this position in self, or don't do anything if path doesn't exist in self
- **path_split** – * used to split path into a list if path is a str, default `" "`

Returns: None

Raises

KeyError – if the value at path doesn't exist

`keys`(*path: Any = ''*, *path_split: OptStr = Ellipsis*) → Iterable[Any]

Returns keys for the node at path, or None if that node is a set or doesn't exist / doesn't have keys

* means that the parameter is a FagusOption, see Fagus-class-docstring for more information about options

Parameters

- **path** – get keys for node at this position in self. Default `""` (gets values from the root node), See get()
- **path_split** – * used to split path into a list if path is a str, default `" "`

Returns

keys for the node at path, or an empty tuple if that node is a set or doesn't exist / doesn't have keys

`values`(*path: Any = ''*, *path_split: OptStr = Ellipsis*, *fagus: OptBool = Ellipsis*, *copy: bool = False*) → Iterable[Any]

Returns values for node at path

* means that the parameter is a FagusOption, see Fagus-class-docstring for more information about options

Parameters

- **path** – get values at this position in self, default "" (gets values from the root node). See get()
- **path_split** – * used to split path into a list if path is a str, default `" "`
- **fagus** – * converts sub-nodes into Fagus-objects in the returned list of values, default False

- **copy** – ~ creates a copy of the node before values() are returned. This can be beneficial if you want to make changes to the returned nodes, but you don't want to change self. Default False

Returns
values for the node at path. Returns an empty tuple if the value doesn't exist, or just the value in a tuple if the node isn't iterable.

items(*path: Any = ''*, *path_split: OptStr = Ellipsis*, *fagus: OptBool = Ellipsis*, *copy: bool = False*) → Iterable[Any]

Returns in iterator of (key, value)-tuples in self, like dict.items()

* means that the parameter is a FagusOption, see Fagus-class-docstring for more information about options

Parameters

- **path** – get items at this position in self, Default "" (gets values from the root node). See get()
- **path_split** – * used to split path into a list if path is a str, default " "
- **fagus** – * converts sub-nodes into Fagus-objects in the returned iterator, default False
- **copy** – ~ creates a copy of the node before items() are returned. This can be beneficial if you want to make changes to the returned nodes, but you don't want to change self. Default False

Returns
iterator of (key, value)-tuples in self, like dict.items()

clear(*path: Any = ''*, *path_split: OptStr = Ellipsis*, *copy: bool = False*, *fagus: OptBool = Ellipsis*) → Collection[Any]

Removes all elements from node at path.

* means that the parameter is a FagusOption, see Fagus-class-docstring for more information about options

Parameters

- **path** – clear at this position in self, Default "" (gets values from the root node). See get()
- **path_split** – * used to split path into a list if path is a str, default " "
- **copy** – if ~ is set, a copy of self is modified and then returned (thus self is not modified), default False
- **fagus** – * return self as a Fagus-object if it is a node (tuple / list / dict), default False

Returns
self as a node if fagus is set, or a modified copy of self if copy is set

Raises
TypeError – if the root node needs to be modified and isn't modifiable (e.g. tuple or frozenset)

contains(*value: Any*, *path: Any = ''*, *path_split: OptStr = Ellipsis*) → bool

Check if value is present in the node at path

* means that the parameter is a FagusOption, see Fagus-class-docstring for more information about options

Parameters

- **value** – value to check

- **path** – check if value is in node at this position in self, Default "" (checks root node). See get()
- **path_split** – * used to split path into a list if path is a str, default " "

Returns
whether value is in node at path in self. returns value == node if the node isn't iterable, and false if path doesn't exit in self

count(*path: Any = ''*, *path_split: OptStr = Ellipsis*) → int

Check the number of elements in the node at path

* means that the parameter is a FagusOption, see Fagus-class-docstring for more information about options

Parameters

- **path** – position in self where the number of elements shall be found.Default "" (checks root node). See get() and README for examples
- **path_split** – * used to split path into a list if path is a str, default " "

Returns
the number of elements in the node at path. if there is no node at path, 0 is returned. If the element at path is not a node, 1 is returned

index(*value: Any*, *start: OptInt = Ellipsis*, *stop: OptInt = Ellipsis*, *path: Any = ''*, *all_: bool = False*, *path_split: OptStr = Ellipsis*) → Optional[Union[int, Any, Sequence[Any]]]

Returns the index / key of the specified value in the node at path if it exists

Parameters

- **value** – ~ to search index for
- **start** – start searching at this index. Only applicable if the node at path is a list / tuple
- **stop** – stop searching at this index. Only applicable if the node at path is a list / tuple
- **path** – position in self where the node shall be searched for value. Default "" (checks root node). See get() and README for examples
- **all_** – returns all matching indices / keys in a generator (instead of only the first)
- **path_split** – * used to split path into a list if path is a str, default " "

Returns
The first index of value if the node at path is a list, or the first key containing value if the node at path is a dict. True if the node at path is a Set and contains value. If the element can't be found in the node at path, or there is no Collection at path, None is returned (instead of a ValueError).

isdisjoint(*other: Iterable[Any]*, *path: Any = ''*, *path_split: OptStr = Ellipsis*, *dict_: str = 'keys'*) → bool

Returns whether the other iterable is disjoint (has no common items) with the node at path

* means that the parameter is a FagusOption, see Fagus-class-docstring for more information about options

Parameters

- **other** – other object to check
- **path** – check if the node at this position in self, is disjoint from other
- **path_split** – * used to split path into a list if path is a str, default " "

- **dict_** – use (k)eys, (v)alues or (i)tems for if value is a dict. Default keys

Returns: whether the other iterable is disjoint from the value at path. If value is a dict, the keys are used.

Checks if value is present in other if value isn't iterable. Returns True if there is no value at path.

child(*obj: Optional[Collection[Any]] = None*, ***kwargs*) → Fagus

Creates a Fagus-object for obj that has the same options as self

reversed(*path: Any = ''*, *fagus: OptBool = Ellipsis*, *path_split: OptStr = Ellipsis*, *copy: bool = False*) → Iterator[Any]

Get reversed child-node at path if that node is a list

* means that the parameter is a FagusOption, see Fagus-class-docstring for more information about options

Parameters

- **path** – position in self where a list / tuple shall be returned reversed
- **fagus** – * converts sub-nodes into Fagus-objects in the returned iterator, default False
- **path_split** – * used to split path into a list if path is a str, default " "
- **copy** – ~ creates a copy of the node before it is returned reversed(). This can be beneficial if you want to make changes to the returned nodes, but you don't want to change self. Default False

Returns

a reversed iterator on the node at path (empty if path doesn't exist)

reverse(*path: Any = ''*, *fagus: OptBool = Ellipsis*, *path_split: OptStr = Ellipsis*, *copy: bool = False*) → Collection[Any]

Reverse child-node at path if that node exists and is reversible

* means that the parameter is a FagusOption, see Fagus-class-docstring for more information about options

Parameters

- **path** – position in self where a list / tuple shall be reversed
- **fagus** – * converts sub-nodes into Fagus-objects in the returned iterator, default False
- **path_split** – * used to split path into a list if path is a str, default " "
- **copy** – ~ creates a copy of the node before it is returned reversed(). This can be beneficial if you want to make changes to the returned nodes, but you don't want to change self. Default False

Returns

self as a node if fagus is set, or a modified copy of self if copy is set

Raises

TypeError – if the root node needs to be modified and isn't modifiable (e.g. tuple or frozenset)

copy(*deep: bool = False*) → Collection[Any]

Creates a copy of self. Creates a recursive shallow copy by default, or a copy.deepcopy() if deep is set.

options(*options: Optional[Dict[str, Any]] = None*, *get_default_options: bool = False*, *reset: bool = False*) → Dict[str, Any]

Function to set multiple Fagus-options in one line

Parameters

- **options** – dict with options that shall be set
- **get_default_options** – return all options (include default-values). Default: only return options that are set
- **reset** – if ~ is set, all options are reset before options is set

Returns

a dict of options that are set, or all options if get_default_options is set

__copy__(*recursive: bool = False*) → Collection[Any]

Recursively creates a shallow-copy of self

__call__() → Collection[Any]

Calling the Fagus-object returns the root node the Fagus-object is wrapped around (equivalent to .root)

Example

```
>>> from fagus import Fagus
>>> a = Fagus({"f": "q"})
>>> a
Fagus({'f': 'q'})
>>> a()
{'f': 'q'}
>>> a.root  # .root returns the root-object in the same way as ()
{'f': 'q'}
```

Returns

the root object Fagus is wrapped around

__getattr__(*attr: str*) → Any

__getitem__(*item: Any*) → Any

__setattr__(*attr: str*, *value: Any*) → None

Implement setattr(self, name, value).

__setitem__(*path: Any*, *value: Any*) → None

__delattr__(*attr: str*) → None

Implement delattr(self, name).

__delitem__(*path: Any*) → None

__iter__() → Iterator[Any]

__hash__() → int

Return hash(self).

__eq__(*other: Any*) → bool

Return self==value.

__ne__(*other: Any*) → bool

Return self!=value.

__lt__(*other: Any*) → bool
Return self<value.

__le__(*other: Any*) → bool
Return self<=value.

__gt__(*other: Any*) → bool
Return self>value.

__ge__(*other: Any*) → bool
Return self>=value.

__contains__(*value: Any*) → bool

__len__() → int

__bool__() → bool

__repr__() → str
Return repr(self).

__str__() → str
Return str(self).

__iadd__(*value: Any*) → Collection[Any]

__add__(*other: Collection[Any]*) → Collection[Any]

__radd__(*other: Collection[Any]*) → Collection[Any]

__isub__(*other: Collection[Any]*) → Collection[Any]

__sub__(*other: Any*) → Collection[Any]

__rsub__(*other: Collection[Any]*) → Collection[Any]

__imul__(*times: int*) → Collection[Any]

```
__abstractmethods__ = frozenset({})
```

```
__annotations__ = {'_options':  'Optional[Dict[str, Any]]', 'root':
'Collection[Any]'}
```

__dict__ = mappingproxy({'__module__': 'fagus.fagus', '__annotations__': {'root': 'Collection[Any]', '_options': 'Optional[Dict[str, Any]]'}, '__doc__': 'Fagus is a wrapper-class for complex, nested objects of dicts and lists in Python\n\n Fagus can be used as an object by instantiating it, but it\'s also possible to use all methods statically without\n even an object, so that ``a = {}; Fagus.set(a, "top med", 1)`` and ``a = Fagus({}); a.set(1, "top med")`` do the\n same.\n\n The root node is always modified directly. If you don\'t want to change the root node, all the functions where it\n makes sense support to rather modify a copy, and return that modified copy using the copy-parameter.\n\n **FagusOptions**:\n Several parameters used in functions in Fagus work as options so that you don\'t have to specify them each time you\n run a function. In the docstrings, these options are marked with a *, e.g. the fagus parameter is an option.\n Options can be specified at three levels with increasing precedence: at class-level (``Fagus.fagus = True``), at\n object-level (``a = Fagus(), a.fagus = True``) and in each function-call (``a.get("b", fagus=True)``). If you\n generally want to change an option, change it at class-level - all objects in that file will inherit this option.\n If you want to change the option specifically for one object, change the option at object-level. If you only want\n to change the option for one single run of a function, put it as a function-parameter. More thorough examples of\n options can be found in README.md.\n ', '__init__': <function Fagus.__init__>, 'get': <function Fagus.get>, 'iter': <function Fagus.iter>, 'filter': <function Fagus.filter>, 'split': <function Fagus.split>, '_split_r': <staticmethod(<function Fagus._split_r>)>, 'set': <function Fagus.set>, 'append': <function Fagus.append>, 'extend': <function Fagus.extend>, 'insert': <function Fagus.insert>, 'add': <function Fagus.add>, 'update': <function Fagus.update>, '_build_node': <function Fagus._build_node>, '_put_value': <staticmethod(<function Fagus._put_value>)>, 'setdefault': <function Fagus.setdefault>, 'mod': <function Fagus.mod>, 'mod_all': <function Fagus.mod_all>, 'serialize': <function Fagus.serialize>, '_serialize_r': <staticmethod(<function Fagus._serialize_r>)>, '_serializable_value': <staticmethod(<function Fagus._serializable_value>)>, 'merge': <function Fagus.merge>, 'pop': <function Fagus.pop>, 'popitem': <function Fagus.popitem>, 'discard': <function Fagus.discard>, 'remove': <function Fagus.remove>, 'keys': <function Fagus.keys>, 'values': <function Fagus.values>, 'items': <function Fagus.items>, 'clear': <function Fagus.clear>, 'contains': <function Fagus.contains>, 'count': <function Fagus.count>, 'index': <function Fagus.index>, 'isdisjoint': <function Fagus.isdisjoint>, 'child': <function Fagus.child>, 'reversed': <function Fagus.reversed>, 'reverse': <function Fagus.reverse>, 'copy': <function Fagus.copy>, 'options': <function Fagus.options>, '_opt': <function Fagus._opt>, '_ensure_mutable_node': <staticmethod(<function Fagus._ensure_mutable_node>)>, '_get_mutable_node': <function Fagus._get_mutable_node>, '_mutable_node_type': <staticmethod(<function Fagus._mutable_node_type>)>, '_node_type': <staticmethod(<function Fagus._node_type>)>, '_hash': <function Fagus._hash>, '__copy__': <function Fagus.__copy__>, '__call__': <function Fagus.__call__>, '__getattr__': <function Fagus.__getattr__>, '__getitem__': <function Fagus.__getitem__>, '__setattr__': <function Fagus.__setattr__>, '__setitem__': <function Fagus.__setitem__>, '__delattr__': <function Fagus.__delattr__>, '__delitem__': <function Fagus.__delitem__>, '__iter__': <function Fagus.__iter__>, '__hash__': <function Fagus.__hash__>, '__eq__': <function Fagus.__eq__>, '__ne__': <function Fagus.__ne__>, '__lt__': <function Fagus.__lt__>, '__le__': <function Fagus.__le__>, '__gt__': <function Fagus.__gt__>, '__ge__': <function Fagus.__ge__>, '__contains__': <function Fagus.__contains__>, '__len__': <function Fagus.__len__>, '__bool__': <function Fagus.__bool__>, '__repr__': <function Fagus.__repr__>, '__str__': <function Fagus.__str__>, '__iadd__': <function Fagus.__iadd__>, '__add__': <function Fagus.__add__>, '__radd__': <function Fagus.__radd__>, '__isub__': <function Fagus.__isub__>, '__sub__': <function Fagus.__sub__>, '__rsub__': <function Fagus.__rsub__>, '__imul__': <function Fagus.__imul__>, '__mul__': <function Fagus.__mul__>, '__rmul__': <function Fagus.__rmul__>, '__reversed__': <function Fagus.__reversed__>, '__reduce__': <function Fagus.__reduce__>, '__reduce_ex__': <function Fagus.__reduce_ex__>, '__dict__': <attribute '__dict__' of 'Fagus' objects>, '__weakref__': <attribute '__weakref__' of

__module__ = 'fagus.fagus'

__mul__(*times: int*) → Union[Tuple[Any], List[Any]]

__weakref__
list of weak references to the object (if defined)

__rmul__(*times: int*) → Union[Tuple[Any], List[Any]]

__reversed__() → Iterator[Any]

__reduce__() → Union[str, Tuple[Any, ...]]
Helper for pickle.

__reduce_ex__(*protocol: Any*) → Union[str, Tuple[Any, ...]]
Helper for pickle.

3.1.2 fagus.filters module

This module contains filter-classes used in Fagus

class fagus.filters.FilBase(**filter_args: Any*, *inexclude: str = ''*)

Bases: object

FilterBase - base-class for all filters used in Fagus, providing basic functions shared by all filters

__init__(**filter_args: Any*, *inexclude: str = ''*) → None
Basic constructor for all filter-classes used in Fagus

Parameters

- ***filter_args** – Each argument filters one key in the tree, the last argument filters the leaf-value. You can put a list of values to match different values in the same filter. In this list, you can also specify subfilters to match different grains differently.
- **inexclude** – In some cases it's easier to specify that a filter shall match everything except b, rather than match a. ~ can be used to specify for each argument if the filter shall include it (+) or exclude it (-). Valid example: "++-+". If this parameter isn't specified, all args will be treated as (+).

included(*index: int*) → bool
This function returns if the filter should be an include-filter (+) or an exclude-filter (-) at a given index

Parameters
index – index in filter-arguments that shall be interpreted as include- or exclude-filter

Returns

bool that is True if it is an include-filter, and False if it is an Exclude-Filter, defaults to True if
undefined at index

match_node(*node: Collection[Any]*, *_: Optional[Any] = None*) → bool
This method is overridden by CheckFilter and ValueFilter, and otherwise not in use

__annotations__ = {}

```
__dict__ = mappingproxy({'__module__': 'fagus.filters', '__doc__': 'FilterBase
- base-class for all filters used in Fagus, providing basic functions shared by
all filters', '__init__': <function FilBase.__init__>, 'included': <function
FilBase.included>, 'match_node': <function FilBase.match_node>, '__dict__':
<attribute '__dict__' of 'FilBase' objects>, '__weakref__': <attribute
'__weakref__' of 'FilBase' objects>, '__annotations__': {}})

__module__ = 'fagus.filters'

__weakref__
```

list of weak references to the object (if defined)

class fagus.filters.VFil(*filter_args: Any, inexclude: str = '', invert: bool = False)

Bases: FilBase

ValueFilter - This special type of filter can be used to inspect the entire node

It can be used to e.g. select all the nodes that contain at least 10 elements. See README for an example

__init__(*filter_args: Any, inexclude: str = '', invert: bool = False) → None

Parameters

- ***filter_args** – Each argument filters one key in the tree, the last argument filters the leaf-value. You can put a list of values to match different values in the same filter. In this list, you can also specify subfilters to match different grains differently.
- **inexclude** – In some cases it's easier to specify that a filter shall match everything except b, rather than match a. ~ can be used to specify for each argument if the filter shall include it (+) or exclude it (-). Valid example: "++-+". If this parameter isn't specified, all args will be treated as (+).
- **invert** – Invert this whole filter to match if it doesn't match. E.g. if you want to select all the nodes that don't have a certain property.

match_node(node: Collection[Any], _: Optional[Any] = None) → bool

Verify that a node matches ValueFilter

Parameters

- **node** – node to check
- **_** – this argument is ignored

Returns

bool whether the filter matched

```
__annotations__ = {}

__module__ = 'fagus.filters'
```

class fagus.filters.KFil(*filter_args: Any, inexclude: str = '', str_as_re: bool = False)

Bases: FilBase

KeyFilter - Base class for filters in Fagus that inspect key-values to determine whether the filter matched

__init__(*filter_args: Any, inexclude: str = '', str_as_re: bool = False) → None

Initializes KeyFilter and verifies the arguments passed to it

Parameters

- ***filter_args** – Each argument filters one key in the tree, the last argument filters the leaf-value. You can put a list of values to match different values in the same filter. In this list, you can also specify subfilters to match different grains differently.
- **inexclude** – In some cases it's easier to specify that a filter shall match everything except b, rather than match a. ~ can be used to specify for each argument if the filter shall include it (+) or exclude it (-). Valid example: "++-+". If this parameter isn't specified, all args will be treated as (+).
- **str_as_re** – If this is set to True, it will be evaluated for all str's if they'd match differently as a regex, and in the latter case match these strings as regex patterns. E.g. re.match("a.*", b) will match differently than "a.*" == b. In this case, "a.*" will be used as a regex-pattern. However re.match("abc", b) will give the same result as "abc" == b, so here "abc" == b will be used.

Raises

TypeError – if the filters are not stacked correctly / stacked in a way that doesn't make sense

__getitem__(*index: int*) → Any

Get filter-argument at index

Returns

filter-argument at index, _None if index isn't defined

__setitem__(*key: int*, *value: Any*) → None

Set filter-argument at index. Throws IndexError if that index isn't defined

match(*value: Any*, *index: int = 0*, *_: Optional[Any] = None*) → Tuple[bool, Optional[KFil], int]

match filter at index (matches recursively into subfilters if necessary)

Parameters

- **value** – the value to be matched against the filter
- **index** – index of filter-argument to check
- **_** – this argument is ignored

Returns

whether the value matched the filter, the filter that matched (as it can be a subfilter), and the next index
in that (sub)filter

match_list(*value: int*, *index: int = 0*, *node_length: int = 0*) → Tuple[bool, Optional[KFil], int]

match_list: same as match, but optimized to match list-indices (e. g. no regex-matching here)

Parameters

- **value** – the value to be matched against the filter
- **index** – index of filter-argument to check
- **node_length** – length of the list whose indices shall be verified

Returns

whether the value matched the filter, the filter that matched (as it can be a subfilter), and the next index
in that (sub)filter

match_extra_filters(*node: Collection[Any]*, *index: int = 0*) → bool

Match extra filters on node (CFil and VFil).

Parameters

- **node** – node to be verified
- **index** – filter_index to check for extra filters

Returns
bool whether the extra filters matched

__annotations__ = {}

__module__ = 'fagus.filters'

class fagus.filters.Fil(*filter_args: Any, inexclude: str = '', str_as_re: bool = False***)**

Bases: KFil

TFilter - what matches this filter will actually be visible in the result. See README

__annotations__ = {}

__module__ = 'fagus.filters'

class fagus.filters.CFil(*filter_args: Any, inexclude: str = '', str_as_re: bool = False, invert: bool = False***)**

Bases: KFil

CFil - can be used to select nodes based on values that shall not appear in the result. See README

__init__(*filter_args: Any, inexclude: str = '', str_as_re: bool = False, invert: bool = False***)** → None

Initializes KeyFilter and verifies the arguments passed to it

Parameters

- ***filter_args** – Each argument filters one key in the tree, the last argument filters the leaf-value. You can put a list of values to match different values in the same filter. In this list, you can also specify subfilters to match different grains differently.
- **inexclude** – In some cases it's easier to specify that a filter shall match everything except b, rather than match a. ~ can be used to specify for each argument if the filter shall include it (+) or exclude it (-). Valid example: "++-+". If this parameter isn't specified, all args will be treated as (+).
- **str_as_re** – If this is set to True, it will be evaluated for all str's if they'd match differently as a regex, and in the latter case match these strings as regex patterns. E.g. re.match("a.*", b) will match differently than "a.*" == b. In this case, "a.*" will be used as a regex-pattern. However re.match("abc", b) will give the same result as "abc" == b, so here "abc" == b will be used.

Raises
TypeError – if the filters are not stacked correctly, or stacked in a way that doesn't make sense

match_node(*node: Collection[Any], index: int = 0***)** → bool

Recursive function to completely verify a node and its subnodes in CFil

Parameters

- **node** – node to check
- **index** – index in filter to check (filter is self)

Returns
bool whether the filter matched

__annotations__ = {}

__module__ = 'fagus.filters'

3.1.3 fagus.iterators module

This module contains iterator-classes that are used to iterate over Fagus-objects

class fagus.iterators.**FilteredIterator**(*obj: Collection[Any]*, *filter_value: bool*, *filter_: Fil*, *filter_index: int = 0*)

Bases: object

Iterator class that gives keys and values for any Collection (use optimal_iterator() to initialize it)

static **optimal_iterator**(*obj: Collection[Any]*, *filter_value: bool = False*, *filter_: Optional['Fil'] = None*, *filter_index: int = 0*) → Iterator[Any]

This method returns the simplest possible Iterator to loop through a given object.

If no filter is present, either items or enumerate are called to loop through the keys, for sets ... is put as key for each value (as sets have no meaningful keys). If you additionally need filtering, this class is initialized to support iteration on only the keys and values that pass the filter

__init__(*obj: Collection[Any]*, *filter_value: bool*, *filter_: Fil*, *filter_index: int = 0*) → None

__iter__() → FilteredIterator

__next__() → Any

__dict__ = mappingproxy({'__module__': 'fagus.iterators', '__doc__': 'Iterator class that gives keys and values for any Collection (use optimal_iterator() to initialize it)', 'optimal_iterator': <staticmethod(<function FilteredIterator.optimal_iterator>)>, '__init__': <function FilteredIterator.__init__>, '__iter__': <function FilteredIterator.__iter__>, '__next__': <function FilteredIterator.__next__>, '__dict__': <attribute '__dict__' of 'FilteredIterator' objects>, '__weakref__': <attribute '__weakref__' of 'FilteredIterator' objects>, '__annotations__': {'match_key': 'Callable[[Any, int, Any], Tuple[bool, Optional[KFil], int]]'}})

__module__ = 'fagus.iterators'

__weakref__

list of weak references to the object (if defined)

class fagus.iterators.**FagusIterator**(*obj: Fagus*, *max_depth: int = 9223372036854775807*, *filter_: Optional['Fil'] = None*, *fagus: bool = False*, *iter_fill: Any = <class 'fagus.utils._None'>*, *select: Optional[Union[int, Iterable[Any]]] = None*, *iter_nodes: bool = False*, *copy: bool = False*, *filter_ends: bool = False*)

Bases: object

Iterator-class for Fagus to facilitate the complex iteration with filtering etc. in the tree-object

Internal - use Fagus.iter() to use this iterator on your object

__init__(*obj: Fagus*, *max_depth: int = 9223372036854775807*, *filter_: Optional['Fil'] = None*, *fagus: bool = False*, *iter_fill: Any = <class 'fagus.utils._None'>*, *select: Optional[Union[int, Iterable[Any]]] = None*, *iter_nodes: bool = False*, *copy: bool = False*, *filter_ends: bool = False*) → None

Internal function. Recursively iterates through Fagus-object

Initiate this iterator through Fagus.iter(), there the parameters are discussed as well.

```
__dict__ = mappingproxy({'__module__': 'fagus.iterators', '__doc__':
'Iterator-class for Fagus to facilitate the complex iteration with filtering etc.
in the tree-object\n\n Internal - use Fagus.iter() to use this iterator on your
object', '__init__': <function FagusIterator.__init__>, '__iter__': <function
FagusIterator.__iter__>, '__next__': <function FagusIterator.__next__>, 'skip':
<function FagusIterator.skip>, '__dict__': <attribute '__dict__' of
'FagusIterator' objects>, '__weakref__': <attribute '__weakref__' of
'FagusIterator' objects>, '__annotations__': {}})
```

`__iter__()` → FagusIterator

`__module__ = 'fagus.iterators'`

`__weakref__`
list of weak references to the object (if defined)

`__next__()` → Any

`skip`(*level: int*, *copy: bool = False*) → Any
Skip the remaining iterations of a node at a given level if you're done handling it

Parameters

- **level** (*int*) – which node to skip. Level 0 is the root node, the next node is level 1 etc.
- **copy** (*bool*) – Whether to skip a copy of the node. Can be useful when the tree is modified during iteration

Returns
The node that was skipped

3.1.4 fagus.utils module

This module contains classes and functions used across the Fagus-library that didn't fit in another module

`class fagus.utils.FagusOption`(*name: str, default: ~typing.Any, type_: type = <class 'typing._SpecialForm'>, verify_function: ~typing.Callable[[~typing.Any], bool] = <function FagusOption.<lambda>>, verify_error_msg: ~typing.Optional[str] = None*)

Bases: `object`

Helper class to facilitate Fagus options.

`__init__`(*name: str, default: ~typing.Any, type_: type = <class 'typing._SpecialForm'>, verify_function: ~typing.Callable[[~typing.Any], bool] = <function FagusOption.<lambda>>, verify_error_msg: ~typing.Optional[str] = None*) → None
Initializes FagusOption with the given parameters

Parameters

- **name** (*str*) – The name of the option.
- **default** (*Any*) – The default value for the option if it hasn't been set explicitly at class- or instance level or in the function.
- **type_** (*type*) – The expected type for the input to the option. Defaults to Any. In case the provided input to the option doesn't have the type indicated here, an error-message is thrown.

- **verify_function** (*Callable[[Any], bool]*) – A function to verify the input value to the option. Returns a bool whether the input was valid or not. An error is thrown if the input isn't valid with the error message defined in verify_error_message. Defaults to *lambda x: True*, meaning that any input is valid
- **verify_error_msg** (*Optional[str]*) – An error message to display when the verify_function returns False. Defaults to f"{value} is not a valid value for {self.name}"

Returns
None

`verify`(*value: Any*) → Any

Verifies if the input value to the option has the correct type and passes the validation function.

Parameters
value (*Any*) – The option input value to be verified.

Raises
- **TypeError** – If the input value is not of the expected type.
- **ValueError** – If the input value does not pass the custom validation function.

Returns
The input value if it meets the requirements.

Return type
Any

```
__dict__ = mappingproxy({'__module__': 'fagus.utils', '__doc__': 'Helper class
to facilitate Fagus options.', '__init__': <function FagusOption.__init__>,
'verify': <function FagusOption.verify>, '__dict__': <attribute '__dict__' of
'FagusOption' objects>, '__weakref__': <attribute '__weakref__' of 'FagusOption'
objects>, '__annotations__': {}})
```

```
__module__ = 'fagus.utils'
```

```
__weakref__
```

list of weak references to the object (if defined)

`fagus.utils.EllipsisType`

TypeAlias to represent type(...), which cannot be done in a nicer way prior to Python 3.10

`fagus.utils.OptStr`

TypeAlias for FagusOption requiring a str. Specify custom value as str, or keep ... to use FagusOption default.

alias of `Union`[`str`, `ellipsis`]

`fagus.utils.OptBool`

TypeAlias for FagusOption requiring a bool. Specify custom value as bool, or keep ... to use FagusOption default.

alias of `Union`[`bool`, `ellipsis`]

`fagus.utils.OptInt`

TypeAlias for FagusOption requiring an int. Specify custom value as int, or keep ... to use FagusOption default.

alias of `Union`[`int`, `ellipsis`]

`fagus.utils.OptAny: TypeAlias = typing.Any`

TypeAlias for FagusOption taking any object. Specify custom value, or keep ... to use FagusOption default.

CHAPTER

FOUR

CHANGELOG

2023-08-20 1.1.2 Fixed some errors in the documentation introduced in 1.1.0

- Removed caret dependency for Python 3.6, so now it is compatible with any future Python version.
- Ensure that `TypeAlias` also works in readthedocs
- Added automatically updated TOC to README and CONTRIBUTING

2023-08-19 1.1.0 Fixed strong typing and added more documentation

- Added mypy as a build-dependency to ensure correct and strong typing in the whole library. Consequences:
  - `TypeAlias` was added to make Fagus Options more clear.
  - Now, `OptStr`, `OptBool`, `OptInt` and `OptAny` clearly declare what the `...` means, and make it strongly typed.
- Added external dependency `type_extensions >= 3.74` for Python < 3.10
  - This was necessary to support `TypeAlias`. However, with `>= 3.74` which was released in June 2019, this dependency is kept as open and forgiving as possible.
  - For Python >= 3.10, `Fagus` still has no external dependencies.
- Renamed the `FagusOption` `value_split` to `path_split` which is more descriptive of what it is doing.
- More documentation in README: now all the different `FagusOption`s are documented properly, as well as the basic `set()`, `get()`, `update()`, `add()`, `insert()` and `extend()`-functions.

2022-05-13 1.0.1 Release of Fagus on GitHub and ReadTheDocs

Now. Finally. The documentation is still not completely ready but it's time to get some feedback from the community.

2022-04-05 1.0.0 Renaming to Fagus

Checking GitHub I found that there already were several other libraries and programs having TreeO as a name which I had chosen originally. I then found another (much cooler) name which wasn't in use yet.

2022-04 0.9.0 Release getting closer

Development has been ongoing for almost a year. Documentation and testing takes time, but it is absolutely necessary for a library like this. Finally moving away from two Python-files (one for tests and one for the lib) to a proper `poetry`-project, starting to implement sphinx to parse the docstrings that had been written earlier.

2021-06 0.1.0 First idea for TreeO

Development starts, the idea to this was born writing my Bachelor's thesis where I felt that constantly writing `.get("a", {}).get("b", {}).get("c", {})` was too annoying to go on with.

CHAPTER

FIVE

CONTRIBUTING TO FAGUS

First off, welcome and thank you for taking the time to contribute to Fagus! Any contribution, big or small, is welcome to make Fagus more useful such that more people can benefit from it.

The following is a set of guidelines for contribution to Fagus, which is hosted by the treeorg organisation on GitHub. They are mostly guidelines, not rules. All of this can be discussed - use your best judgement, and feel free to propose changes to this document in a pull request.

5.1 Table of contents

5.2 Fagus Principles

1. **No external dependencies**: Fagus runs on native Python without 3rd party dependencies.
2. **Documented**: All functions / modules / arguments / classes have docstrings.
3. **Tested**: All the functions shall have tests for as many edge cases as possible. It's never possible to imagine all edge-cases, but if e.g. a bug is fixed which there is no test for, a new test case should be added to prevent the bug from being reintroduced.
4. **Consistent**: Fagus's function arguments follow a common structure to be as consistent as possible.
5. **Static and Instance**: All functions in Fagus (except from ___internals___) should be able to run static `Fagus.function(obj)` or at a Fagus-instance `obj = Fagus(); obj.function()`.
6. **Simple and efficient**: If you have suggestions on how to make the code more efficient, feel free to submit.

5.3 How Can I Contribute?

5.3.1 Reporting Bugs

This section guides you through submitting a bug report for Fagus. Following these guidelines helps maintainers and the community understand your report, reproduce the behavior, and find related reports.

Before Submitting A Bug Report

- **Check the FAQ and the discussions** for a list of common questions and problems.
- **Check issues to see if your issue has already been reported**
  - If it has been reported **and the issue is still open**, add a comment to the existing issue instead of opening a new one.
  - If you find a **Closed** issue that seems like it is the same thing that you're experiencing, open a new issue and include a link to the original issue in the body of your new one.

How Do I Submit A (Good) Bug Report?

Bugs are tracked as GitHub issues. When you are creating a bug report, please *include as many details as possible (in particular test-data)*. Fill out the required template, the information it asks for helps us resolve issues faster.

5.3.2 Requesting New Features

This section guides you through submitting an enhancement suggestion for Fagus, including completely new features and minor improvements to existing functionality. Following these guidelines helps maintainers and the community understand your suggestion and find related suggestions.

Before Submitting A Feature Request

- **Check the FAQ and the discussions** for a list of common questions and problems. Probably there already is a solution for your feature-request?
- **Check issues to see if your feature request has already been reported**
  - If it has been reported **and the feature request is still open**, add a comment to the existing issue instead of opening a new one. You can also give it a like to get it prioritized.
  - If you find a **Closed** feature request that seems like it is the same thing that you would like to get added, you can create a new one and include a link to the old one. If many people would like to have a new feature it is more likely to get prioritized.

How Do I Submit A (Good) Feature Request?

Feature requests are tracked as GitHub issues. When you are creating a feature request, please *include as many details as possible (in particular test-data)*. Fill out the required template, the information it asks for helps us to better judge and understand your suggestion.

5.4 Developing Fagus

This section shows you how you can set up a local environment to test and develop Fagus, and finally how you can make your contribution.

5.4.1 Software Dependencies For Development

- Python (at least 3.6.2)
- Poetry for dependency management and deployment (creating packages for PyPi), instructions are found in *installation steps*
- Git to checkout this repo
- An IDE, I used Intellij PyCharm Community. Not mandatory, but I found it handy to see how the data is modified in the debugger.
- Fagus itself has no external dependencies, but some packages are used to smoothen the development process. They are installed and set up through poetry, check pyproject.toml or *Code Styling Rules* for a list.

5.4.2 Code Styling Guidelines

- **Code formatting**: The code is formatted using the PEP-8-Standard, but with a line length of 120 characters.
  - The code is automatically formatted correctly by using black. Run `black .` to ensure correct formatting for all py-files in the repo.
  - The PEP-8-rules are verified through flake8. This tool only shows what is wrong - you'll have to fix it yourself.
- **Docstrings**: All public functions in Fagus have docstrings following the Google Python Style Guide
- **Formatting commit-messages**: commitizen is used to make sure that commit-messages follow a common style
- **Pre-commit checks**: pre-commit is used to ensure that the code changes have test-coverage, are formatted correctly etc. It runs black, flake8, unittests and a lot of other checks prior to accepting a commit.

5.4.3 Setting Up A Local Fagus Developing Environment

1. Install Python and Git
2. Checkout the repository: `git checkout https://github.com/treeorg/Fagus.git; cd Fagus`
3. Instructions how to install poetry can be found here
   - you might have to reopen your terminal after installing poetry (or run `source ~/.bashrc` on Linux)
4. Run `poetry shell` to open a terminal that is set up with the development tools for Fagus.
   - check if you can now run this command without getting errors: `poetry shell`
   - if the `poetry`-command is not found, you might have to add `eval "$(pyenv init --path)"` to your `.bashrc` (on Linux)
   - if you have problems setting this up, just ask a question, we can later include the problem and the solution we found into this guide

5. Install the project and developing dependencies: `poetry install`
6. If you use an IDE, you can now open your project there. If it has a poetry mode, use that mode - `poetry shell` will then be executed automatically in the terminal of your IDE.

5.4.4 Submitting Pull Requests for Fagus

If it hasn't run in your console yet, run `poetry shell` to get all the development dependencies and some new commands available in your console.

Tests

You can run `python3 -m unittest discover` to run all the tests in `./tests`. If you add new functionality in your pull-request, make sure that the tests still work, or update them if necessary. As this is a generic library, it's very important that all the functions have test coverage for as many edge cases as possible.

Doctests have also been defined, some in the docstrings in the `fagus`-module, others in `README.md`. Run `python3 -m tests.test_fagus doctest` to run all the doctests, and make sure that they still work.

Committing using pre-commit and commitizen

1. Make sure all your changes are staged for commit: `git add -A` includes all of your changes
2. Dry-run the pre-commit-checks: `pre-commit`
   - Some errors like missing trailing whitespace or wrong formatting are automatically corrected.
   - If there are errors in the tests, or flake8 observes problems, you'll have to go back in the code and fix the problems.
3. Repeat Step 1 and 2 until all the tests are green.
4. Use `git cz c` to commit using commitizen.
   - If the pre-commit-checks fail, your commit is rejected and after fixing the issues you'd have to retype the commit-message. To not have that problem, do step 3 beforehand.

Releasing A New Fagus Package on PyPi

1. Run the commands from *Tests* to ensure that the tests still work. If possible, also test for Python 3.6.
2. Update `Changelog.md` with a description of the changes you have made.
3. Manually run `package.py` from the project's root folder using the following command
   - `python3 package update -bdlp -v <version number or increment>`
   - `b` builds the package for later upload to PyPi
   - `d` updates the documentation files (see if this runs properly, if it does it will work on sphinx as well)
   - `l` builds a pdf-documentation file
   - `p` runs the pre-commit checks to ensure that everything is alright before publishing
   - `v` requires a version number or increment. Either manually put a version number here, or use one of the following increments:
     - **major**: For backwards incompatible changes (e.g. removing support for Python 3.6)
     - **minor**: Adds functionality in a backwards compatible way

  - **patch**: Fixes bugs in a backwards compatible way

4. Make a commit including all the changes made in step 1 and 2, and repeat them if necessary. Check the following before committing:
   - Ensure that the version number mentioned in `CHANGELOG.md` corresponds to the one that is now present in `pyproject.toml`. If it is not equal, update `CHANGELOG.md` accordingly, and rerun step 2 but without the version-parameter `-v`.
   - Go through the errors and warnings which are thrown especially while the documentation is created in step 2.
     - This warning is alright: `WARNING: more than one target found for cross-reference 'Fil': fagus.Fil, fagus.filters.Fil`
     - Fix all other warnings / errors.
5. Create a Pull Request for the changes back to the `main`-branch, this is easiest to do directly on GitHub. Use the title and text from `CHANGELOG.md` for the title and description of the PR.
6. Run `poetry publish` to publish the new version to `PyPi`.
   - If it doesn't work, make sure that you are allowed to publish to Fagus.
   - Set up an access token in your PyPi account here.
   - Then run `poetry config pypi-token.pypi <my-token>` documented here to add the token to your `poetry`-configuration, `poetry publish` should now work.

INDEX

Symbols

A

C

D

E

F

G

I

K

M

O

P

R

S

U

V