Fagus

Release 1.0.1

Lukas Neuenschwander

May 17, 2022

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 Basic principles

2.2.1 Introduction – What it solves

Imagine you want to fetch values from a nested dict like 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.

2.2.2 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.

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

2.2.3 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"]
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 `dicts` 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 *`value_split`*

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

2.2.4 Fagus options

There are several parameters used across many functions in `Fagus` which steer 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.

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.

The four levels of `Fagus`-options:

1. **Argument**: The highest level - if an option is specified directly as an argument to a function, that value takes precedence over all other levels.
2. **Instance**: If an option is set for an instance, it will apply to all function calls at that instance where level one has not been specified.
3. **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.
4. **Default**: If no other level is specified, the hardcoded default for that option is used.

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
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
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 two 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 two from their parent.

The remaining part of this section explains the options one by one.

default

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

This value is returned if the requested *`path`* does not exist. Example in *Introduction*, code block two.

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:

```
1  >>> Fagus.default_node_type = "l"
2  >>> a = Fagus()
3  >>> a()  # the root node of a is an empty list as this was set in line 2
4  []
5  >>> del Fagus.default_node_type
6  >>> b = Fagus()
7  >>> b()  # the root node of b is a dict (default for default_node_type)
8  {}
```

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.

```
1  >>> a = Fagus(if_=True)  # the only allowed value for set is now True
2  >>> a.v1 = True
3  >>> a()  # v1 was set, because it was True (as requested in line 1)
4  {'v1': True}
5  >>> a.v2 = None
6  >>> a()  # note that v2 has not been set as it was not True
7  {'v1': True}
8  >>> a.set(6, "v2", if_=(4, 5, 6))  # 6 was set as it was in (4, 5, 6)
9  {'v1': True, 'v2': 6}
10 >>> a.set("", "v3", if_=bool)  # v3 is not set because bool("") is False
11 {'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()`*.

value_split

2.2.5 Iterating over nested objects

Skipping nodes in iteration.

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*

3.1 Submodules

3.1.1 fagus.fagus module

Base-module that contains the Fagus-class

class fagus.fagus.**Fagus**(*root: Optional[collections.abc.Collection] = None*, *node_types: str = Ellipsis*, *list_insert: int = Ellipsis*, *value_split: str = Ellipsis*, *fagus: bool = Ellipsis*, *default_node_type: str = Ellipsis*, *default=Ellipsis*, *if_=Ellipsis*, *iter_fill=Ellipsis*, *mod_functions: collections.abc.Mapping = Ellipsis*, *copy: bool = False*)

Bases: collections.abc.MutableMapping, collections.abc.MutableSequence, collections.abc.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.

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[collections.abc.Collection] = None*, *node_types: str = Ellipsis*, *list_insert: int = Ellipsis*, *value_split: str = Ellipsis*, *fagus: bool = Ellipsis*, *default_node_type: str = Ellipsis*, *default=Ellipsis*, *if_=Ellipsis*, *iter_fill=Ellipsis*, *mod_functions: collections.abc.Mapping = 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 Fagus-Setting, 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.
- **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
- **value_split** – * used to split path into a list if path is a string, default ” “
- **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”
- **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()
- **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

get(*path: Any = ''*, *default=Ellipsis*, *fagus: bool = Ellipsis*, *copy: bool = False*, *value_split: str = 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 value_split (default ” “), so a[“a 1 c”] also returns “d”.

* means that the parameter is a Fagus-Setting, 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 value_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
- **value_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[*fagus.filters.Fil*] = None*, *fagus: bool = Ellipsis*, *iter_fill=Ellipsis*, *select: Optional[Union[int, collections.abc.Iterable]] = None*, *copy: bool = False*, *iter_nodes: bool = Ellipsis*, *filter_ends: bool = False*, *value_split: str = Ellipsis*) → *fagus.iterators.FagusIterator*

Recursively iterate through Fagus-object, starting at path

* means that the parameter is a Fagus-Setting, 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 TFilter (see README.md and TFilter 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.md 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)

- **value_split** – * used to split path into a list if path is a str, default " "

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

filter(*filter_: fagus.filters.Fil, path: Any = '', fagus: bool = Ellipsis, copy: bool = False, default=Ellipsis, value_split: str = Ellipsis*) → collections.abc.Collection

Filters self, only keeping the nodes that pass the filter

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

Parameters

- **filter_** – TFilter-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
- **value_split** – * used to split path into a list if path is a str, default " "

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_: fagus.filters.Fil, path: Any = '', fagus: bool = Ellipsis, copy: bool = False, default=Ellipsis, value_split: str = Ellipsis*) → Union[Tuple[collections.abc.Collection, collections.abc.Collection], 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 Fagus-Setting, see Fagus-class-docstring for more information about options

Parameters

- **filter_** – TFilter-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, or the
- **value_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*, *path: collections.abc.Iterable*, *node_types: str = Ellipsis*, *list_insert: int = Ellipsis*, *value_split: str = Ellipsis*, *fagus: bool = Ellipsis*, *if_: Any = Ellipsis*, *default_node_type: str = Ellipsis*, *copy: bool = False*) → collections.abc.Collection

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 Fagus-Setting, 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 value_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.
- `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
- `value_split` – * used to split path into a list if path is a string, default " "
- `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"
- `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*, *path: Any = ''*, *node_types: str = Ellipsis*, *list_insert: int = Ellipsis*, *value_split: str = Ellipsis*, *fagus: bool = Ellipsis*, *if_: Any = Ellipsis*, *default_node_type: str = Ellipsis*, *copy: bool = False*) → collections.abc.Collection

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 Fagus-Setting, 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 value_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.
- **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
- **value_split** – * used to split path into a list if path is a string, default " "
- **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"
- **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: collections.abc.Iterable*, *path: Any = ''*, *node_types: str = Ellipsis*, *list_insert: int = Ellipsis*, *value_split: str = Ellipsis*, *fagus: bool = Ellipsis*, *if_: Any = Ellipsis*, *default_node_type: str = Ellipsis*, *copy: bool = False*) → collections.abc.Collection

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 Fagus-Setting, 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 value_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.
- **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
- **value_split** – * used to split path into a list if path is a string, default " "
- **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"
- **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*, *path: Any = ''*, *node_types: str = Ellipsis*, *list_insert: int = Ellipsis*, *value_split: str = Ellipsis*, *fagus: bool = Ellipsis*, *if_: Any = Ellipsis*, *default_node_type: str = Ellipsis*, *copy: bool = False*) → collections.abc.Collection

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 Fagus-Setting, 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 value_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.

- **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
- **value_split** – * used to split path into a list if path is a string, default " "
- **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"
- **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*, *path: Any = ''*, *node_types: str = Ellipsis*, *list_insert: int = Ellipsis*, *value_split: str = Ellipsis*, *fagus: bool = Ellipsis*, *if_: Any = Ellipsis*, *default_node_type: str = Ellipsis*, *copy: bool = False*) → collections.abc.Collection

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 Fagus-Setting, 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 value_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.
- **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
- **value_split** – * used to split path into a list if path is a string, default " "
- **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"
- **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: collections.abc.Iterable*, *path: Any = ''*, *node_types: str = Ellipsis*, *list_insert: int = Ellipsis*, *value_split: str = Ellipsis*, *fagus: bool = Ellipsis*, *if_: Any = Ellipsis*, *default_node_type: str = Ellipsis*, *copy: bool = False*) → collections.abc.Collection

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 Fagus-Setting, 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 value_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.
- **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
- **value_split** – * used to split path into a list if path is a string, default " "
- **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"

- **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=Ellipsis*, *fagus: bool = Ellipsis*, *node_types: str = Ellipsis*, *list_insert: int = Ellipsis*, *value_split: str = Ellipsis*, *default_node_type: str = 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 Fagus-Setting, 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.
- **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
- **value_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"

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*, *path*, *default=Ellipsis*, *replace_value=True*, *fagus: bool = Ellipsis*, *node_types: str = Ellipsis*, *list_insert: int = Ellipsis*, *value_split: str = Ellipsis*, *default_node_type: str = 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 Fagus-Setting, 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 value_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.
- **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
- **value_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"

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*, *filter_: Optional[fagus.filters.Fil] = None*, *path: Any = ''*, *replace_value=True*, *default=Ellipsis*, *max_depth: int = 9223372036854775807*, *fagus: bool = Ellipsis*, *copy=False*, *value_split: str = Ellipsis*) → collections.abc.Collection

Modify all the leaf-values that match a certain filter

* means that the parameter is a Fagus-Setting, 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)
- **value_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[collections.abc.Mapping] = None*, *path: Any = ''*, *node_types: str = Ellipsis*, *list_insert: int = Ellipsis*, *value_split: str = Ellipsis*, *copy: bool = False*) → Union[dict, list]

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 Fagus-Setting, 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.

- **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
- **value_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[fagus.iterators.FagusIterator, collections.abc.Collection]*, *path: Any = ''*, *new_value_action: str = 'r'*, *extend_from: int = 9223372036854775807*, *update_from: int = 9223372036854775807*, *fagus: bool = Ellipsis*, *copy: bool = False*, *copy_obj: bool = False*, *value_split: str = Ellipsis*, *node_types: str = Ellipsis*, *list_insert: int = Ellipsis*, *default_node_type: str = Ellipsis*) → collections.abc.Collection

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
- **value_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.

- **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"

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=Ellipsis*, *fagus: bool = Ellipsis*, *value_split: str = Ellipsis*)

Deletes the value at path and returns it

* means that the parameter is a Fagus-Setting, 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)
- **value_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()

This function is not implemented in Fagus

discard(*path: Any = ''*, *value_split: str = Ellipsis*) → None

Deletes the value at path if it exists

* means that the parameter is a Fagus-Setting, 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
- **value_split** – * used to split path into a list if path is a str, default " "

Returns: None

`remove`(*path: Any = ''*, *value_split: str = Ellipsis*) → None

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

* means that the parameter is a Fagus-Setting, 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
- **value_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 = ''*, *value_split: str = Ellipsis*)

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 Fagus-Setting, 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()
- **value_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 = ''*, *value_split: str = Ellipsis*, *fagus: bool = Ellipsis*, *copy: bool = False*)

Returns values for node at path

* means that the parameter is a Fagus-Setting, 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()
- **value_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 = ''*, *value_split: str = Ellipsis*, *fagus: bool = Ellipsis*, *copy: bool = False*)

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

* means that the parameter is a Fagus-Setting, 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()
- **value_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 = ''*, *value_split: str = Ellipsis*, *copy: bool = False*, *fagus: bool = Ellipsis*) → collections.abc.Collection

Removes all elements from node at path.

* means that the parameter is a Fagus-Setting, 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()
- **value_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*, *path: Any = ''*, *value_split: str = Ellipsis*) → bool

Check if value is present in the node at path

* means that the parameter is a Fagus-Setting, 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()
- **value_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 = ''*, *value_split: str = Ellipsis*) → int

Check the number of elements in the node at path

* means that the parameter is a Fagus-Setting, 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()
- **value_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: int = Ellipsis*, *stop: int = Ellipsis*, *path: Any = ''*, *all_: bool = False*, *value_split: str = Ellipsis*) → Optional[Union[int, Any, collections.abc.Sequence]]

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()
- **all_** – returns all matching indices / keys in a generator (instead of only the first)
- **value_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: collections.abc.Iterable*, *path: Any = ''*, *value_split: str = 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 Fagus-Setting, 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
- **value_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 ke

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

child(*obj: Optional[collections.abc.Collection] = None*, ***kwargs*) → *fagus.fagus.Fagus*

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

reversed(*path: Any = ''*, *fagus: bool = Ellipsis*, *value_split: str = Ellipsis*, *copy: bool = False*)

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

* means that the parameter is a Fagus-Setting, 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
- **value_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: bool = Ellipsis*, *value_split: str = Ellipsis*, *copy: bool = False*) → collections.abc.Collection

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

* means that the parameter is a Fagus-Setting, 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
- **value_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*)

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

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

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=False*)

Recursively creates a shallow-copy of self

__call__()

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*)

__getitem__(*item*)

__setattr__(*attr*, *value*)

Implement setattr(self, name, value).

__setitem__(*path*, *value*)

__delattr__(*attr*)

Implement delattr(self, name).

__delitem__(*path*)

__iter__()

__hash__()

Return hash(self).

__eq__(*other*)

Return self==value.

__ne__(*other*)

Return self!=value.

__lt__(*other*)

Return self<value.

__le__(*other*)

Return self<=value.

__gt__(*other*)

Return self>value.

__ge__(*other*)

Return self>=value.

__contains__(*value*)

__len__()

__bool__()

__repr__()

Return repr(self).

__str__()

Return str(self).

__iadd__(*value*)

__add__(*other*)

__radd__(*other*)

__isub__(*other*)

__sub__(*other*)

__rsub__(*other*)

__imul__(*times: int*)

__mul__(*times: int*)

__rmul__(*times: int*)

__abstractmethods__ = frozenset({})

```
__dict__ = mappingproxy({'__module__': 'fagus.fagus', '__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 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 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 generally want\n to change an option, change it at class-level - all objects
in that file will inherit this option. If you want to\n change the option
specifically for one object, change the option at object-level. If you only want
to change the\n option for one single run of a function, put it as a
function-parameter. More thorough examples of options can be\n 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 object>, '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 object>, 'setdefault': <function Fagus.setdefault>,
'mod': <function Fagus.mod>, 'mod_all': <function Fagus.mod_all>, 'serialize':
<function Fagus.serialize>, '_serialize_r': <staticmethod object>,
'_serializable_value': <staticmethod object>, '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 object>, '_get_mutable_node': <function Fagus._get_mutable_node>,
'_node_type': <staticmethod object>, '_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
'Fagus' objects>, '__abstractmethods__': frozenset(), '_abc_impl': <_abc_data
object>, '__annotations__': {}})
```

__module__ = 'fagus.fagus'

__reversed__()

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

__reduce__()
Helper for pickle.

__reduce_ex__(*protocol*)
Helper for pickle.

3.1.2 fagus.filters module

This module contains filter-classes used in Fagus

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

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

__init__(**filter_args*, *inexclude: str = ''*)
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*) → 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 undefined at index

match_node(*node: collections.abc.Collection*, *_=None*) → bool
This method is overridden by CheckFilter and ValueFilter, and otherwise not in use

__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*, *inexclude: str = ''*, *invert: bool = False*)

Bases: *fagus.filters.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*, *inexclude: str = ''*, *invert: bool = False*)

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: collections.abc.Collection*, *_=None*) → bool

Verify that a node matches ValueFilter

Parameters

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

Returns bool whether the filter matched

__module__ = 'fagus.filters'

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

Bases: *fagus.filters.FilBase*

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

__init__(**filter_args*, *inexclude: str = ''*, *str_as_re: bool = False*)

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*, *value*)

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

`match`(*value*, *index: int = 0*, *__=None*) → Tuple[bool, Optional[*fagus.filters.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), a in that (sub)filter

`match_list`(*value: int*, *index: int = 0*, *node_length: int = 0*) → Tuple[bool, Optional[*fagus.filters.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), a in that (sub)filter

`match_extra_filters`(*node: collections.abc.Collection*, *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

`__module__ = 'fagus.filters'`

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

Bases: *fagus.filters.KFil*

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

`__module__ = 'fagus.filters'`

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

Bases: *fagus.filters.KFil*

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

__init__(*filter_args, inexclude: str = '', str_as_re: bool = False, invert: bool = False)

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: collections.abc.Collection, 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

__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: collections.abc.Collection, 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: collections.abc.Collection, filter_value: bool = False, filter_: Fil = None, filter_index: int = 0*)

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: collections.abc.Collection, filter_value: bool, filter_: Fil, filter_index: int = 0*)

__iter__()

__next__()

__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 object>, '__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__': {}})

__module__ = 'fagus.iterators'

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

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

Internal function. Recursively iterates through Fagus-object

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

__iter__()

__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__': {}})

__module__ = 'fagus.iterators'

__next__()

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

skip(*level: int*, *copy: bool = False*) → collections.abc.Collection

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.**FagusMeta**(*name*, *bases*, *namespace*, ***kwargs*)

Bases: abc.ABCMeta

Metaclass for Fagus-objects to facilitate options at class-level

static __verify_option__(*option_name: str*, *option*)

Verify Fagus-option using the functions / types in ___default_options___

Parameters

- **option_name** – name of the option to verify
- **option** – the value to be verified

Returns the option-value if it was valid (otherwise the function is left in an error)

__default_options__ = {'default': (None,), 'default_node_type': ('d', <class 'str'>, <function FagusMeta.<lambda>>, 'default_node_type must be either "d" for dict or "l" for list.'), 'fagus': (False, <class 'bool'>), 'if_': (<class 'fagus.utils._None'>,), 'iter_fill': (<class 'fagus.utils._None'>,), 'iter_nodes': (False, <class 'bool'>), 'list_insert': (9223372036854775807, <class 'int'>, <function FagusMeta.<lambda>>, 'list-insert must be a positive int. By default (list_insert == INF), all existing list-indices are traversed. If list-insert < maxsize, earliest at level n a new node is inserted if that node is a list'), 'node_types': ('', <class 'str'>, <function FagusMeta.<lambda>>, 'The only allowed characters in node_types are d (for dict) and l (for list). " " can also be used. In that case, existing nodes are used if possible, and default_node_type is used to create new nodes.'), 'value_split': (' ', <class 'str'>, <function FagusMeta.<lambda>>, 'value_split can\'t be "", as a string can\'t be split by "".')}

Default values for all options used in Fagus

no_node = (<class 'str'>, <class 'bytes'>, <class 'bytearray'>)

Every type of Collection in no_node will not be treated as a node, but as a single value

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

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

__setattr__(*attr*, *value*)

Implement setattr(self, name, value).

__getattr__(*attr*)

__delattr__(*attr*)

Implement delattr(self, name).

__module__ = 'fagus.utils'

CHAPTER

FOUR

CHANGELOG

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.

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 `poetry version <major, minor or patch>`to increment the version number in poetry.
   - **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
2. Run `sed -i "s/__version__ = .*\$/__version__ = \"$(poetry version -s)\"/" fagus/__init__.py` (only works on Linux / MacOS) to update the version number in the fagus-package. If you know the command to do this replacement in a windows shell, feel free to add it here.

PYTHON MODULE INDEX

f

INDEX

Symbols

A

C

D

E

F

G

I

K

M

N

O

P

R

S

U

V