dicthandling

Release 0.2

Oct 09, 2020

Contents:

Dicthandling contains convenient methods to work with nested dictionaries.

Using *print_tree()* returns a pretty print of a nested dictionary, which you may find more readable than the standard output.

Further functions of this module are

- for accessing data
  - *get_leaf()* gets an item at an address
  - *set_leaf()* sets an item at an address
  - *update_only_values()*
  - *add_missing_branches()*
  - *overlap_branches()*
  - *join_address()*
- for creating nested dictionaries
  - *get_unused_key()* gets a key which is not in the existing dictionary
  - *unfold_branch()* creates a nested dictionary on base of an address
- Configparser related functions
  - *flatten()*
  - *deep_flatten()*
  - *unflatten()*
  - *deep_unflatten()*
- for storing and reading with the json module
  - *put_into_json_file()*
  - *read_from_json_file()*

CHAPTER 1

General

dicthandling.**print_tree**(*data: dict*, *max_itemlength: int = 40*, *hide_leading_underscores: bool = False*, *indent: str = '..'*, *starting_indent: str = ''*, *hide_empty: bool = False*)

Prints a pretty representation of a nested dictionary.

Examples

```
>>> data = {'a': {'b': 'c', 'd': 'e', 'f': ['g', 'h']}}
>>> print_tree(data)
[a]:
..b: c
..d: e
..f: ['g', 'h']
>>> data = [{'a': {'b': 'c'}}, {'d': ['e', 'f']}]
>>> print_tree(data)
[0]:
..[a]:
....b: c
[1]:
..d: ['e', 'f']
>>> data = {'a': 'b', 'c': None, 'd': 'e', 'f': ''}
>>> print_tree(data, hide_empty=True)
a: b
d: e
>>> data = [{'a': 'b', 'c': None, 'd': 'e'}, {'f': '', 'g': {}, 'h': []}]
>>> print_tree(data, hide_empty=True)
[0]:
..a: b
..d: e
[1]:
..[g]: {}
>>> print_tree(
...     {"This position": "->Nothing<- should be cut from this extra long string."}
... )
```

(continues on next page)

(continued from previous page)

```
This position: ->..<- should be cut from this extra long string.
>>> print_tree({"remove": "trailing whitespaces  "})
remove: trailing whitespaces
```

Parameters

- **data** (*dict*) – (Nested) dictionary which should be printed.
- **max_itemlength** (*int*) – maximum length of value item; if exceeded the value string will be segmented. Default = 40
- **hide_leading_underscores** (*bool*) – if true all branches with leading underscores will be hidden. default = False
- **indent** (*str*) – string which will be used for indentation. default = '..'
- **starting_indent** (*str*) – The intendation the tree starts with. default = ''
- **hide_empty** (*bool*) – Don't prints empty fields of dictionaries, which are None, '' (empty sequences). Empty dicitonaries will still be shown.

dicthandling.**keep_keys**(*root: dict, keysToKeep: List[str]*) → dict

Returns a dictionary with the given set of `keys` from `root` as a deep copy.

Parameters

- **root** (*dict*) – root-dictionary from which key-value pairs should be kept.
- **[list of str]** (*keepingKeys*) – Keys to be kept within the result.

Returns Deep copied dictionary.

Return type dict

CHAPTER 2

Accessing

dicthandling.**get_leaf**(*tree: dict*, *address: str*) → Optional[dict]
Returns the leaf of a address split by the FOLDING_KEY_DELIMITER. The address needs to be absolute.

A address is like: rootkey/leaf1key/leaf2key

Parameters

- **tree** (*dict*) – (Nested) dictionary from which a leaf shall be pulled.
- **address** (*str*) – The address of the leaf of the dict to pull.

Returns The dictionary item at the given ´´address´´. None if address not found.

Return type dict

dicthandling.**set_leaf**(*tree: dict*, *address: str*, *leaf*, *forced: bool = False*) → dict
Updates the leaf of a address split by the FOLDING_KEY_DELIMITER. The address needs to be absolute. If the address is not within the tree no changes will be performed by default.

A address is like: rootkey/leaf1key/leaf2key

Parameters

- **tree** (*dict*) – (Nested) dictionary in which a leaf shall be updated.
- **address** (*str*) – The address of the leaf of the dict to update.
- **leaf** (*any type*) – Item which shall be set to the address.
- **forced** (*bool*) – If ´´True´´ forces the leaf into the tree, by creating all necessary branches and overriding existing within this branch. Default = True.

Returns Returns updated tree. If setting failed internally the original tree is returned.

Return type dict

dicthandling.**update_only_values**(*destination: dict*, *items: dict*) → dict
Updates a destination-dictionary with key-pairs of `items`, if these are not *None* or empty.

Parameters

- **destination** (*dict*) – Destination were the items should be put.
- **items** (*dict*) – items to be put into destination, if not *None* or empty.

Returns The destination dictionary.

Return type destination(dict)

dicthandling.**add_missing_branches**(*targetbranch: dict*, *sourcebranch: dict*) → dict
Overlaps to dictionaries with each other. Only missing branches are taken from *sourcebranch*.

Parameters

- **targetbranch** (*dict*) – Root where the new branch should be put.
- **sourcebranch** (*dict*) – New data to be put into the rootBranch.

dicthandling.**overlap_branches**(*targetbranch: dict*, *sourcebranch: dict*) → dict
Overlaps to dictionaries with each other. This method does apply changes to the given dictionary instances.

Examples

```
>>> overlap_branches(
...     {"a": 1, "b": {"de": "ep"}},
...     {"b": {"de": {"eper": 2}}}
... )
{'a': 1, 'b': {'de': {'eper': 2}}}
>>> overlap_branches(
...     {},
...     {"ne": {"st": "ed"}}
... )
{'ne': {'st': 'ed'}}
>>> overlap_branches(
...     {"ne": {"st": "ed"}},
...     {}
... )
{'ne': {'st': 'ed'}}
>>> overlap_branches(
...     {"ne": {"st": "ed"}},
...     {"ne": {"st": "ed"}}
... )
{'ne': {'st': 'ed'}}
```

Parameters

- **targetbranch** (*dict*) – Root where the new branch should be put.
- **sourcebranch** (*dict*) – New data to be put into the sourcebranch.

dicthandling.**ADDRESS_DELIMITER**
Delimiter by which parts of a path within a dictionary are separated.

dicthandling.**join_address**(*address: Union[str, int]*, **addresses*) → str
Join one or more address components. The return value is the concatenation of *address* and any members of **addresses*

Parameters

- **address** (*Union[str, int]*) – root address of the path
- ***addresses** – address parts to be concatenated.

Returns Address within the dictionary.

Return type str

CHAPTER 3

Creation

dicthandling.**get_unused_key**(*root: dict*, *key: str*) → str

Returns a not used key with addition of ' (i)' within root based on the given key, if key already exists within this database.

Parameters **root** (*dict*) – Root dictionary to put the key in.

Returns First occurrence of non existing key with a postfix of ' (i)' if necessary, else given *key*.

Return type str

dicthandling.**unfold_branch**(*address: str*, *leaf: Optional[dict] = None*) → dict

Creates a branch by given address separated by FOLDING_KEY_DELIMITER.

Parameters

- **address** (*str*) – Address of keys of the branch
- **(Optional[dict]** (*leaf*) – leaf of the branch to put in. Default = None

Returns Nested dictionary based on given address and leaf.

Return type dict

CHAPTER 4

Configparser related functions

dicthandling.**flatten**(*data: dict*, *parentkey=None*) → collections.OrderedDict

Flattens a nested dictionary containing dictionaries to a single dictionary, where all items are flat dictionaries. Top level items will of the root dictionary be put into a dictionary at the key 'root'.

Parameters

- **data** (*dict*) – dictionary to be flatten
- **parentkey** (*string*) – Key of the parent dictionary. Default is *None*

Returns Ordered dictionary with a depth of 2.

Return type collections.OrderedDict

Raises ValueError – If data is not a dictionary.

Example: A nested dicionary like ..

```
root
|- comment : "toplevel item"
|- project : { }
              |- editor : Fry
              |- year : 4029
              |- custumor : Planet Express
              |- paths : { }
                          |- rawdatapath : "/tmp/here"
                          |- processedpath : "/tmp/there"
```

will be converted into ..

```
root
|- DEFAULT : { }
|              |- comment : "toplevel item"
|
|- project : { }
|              |- editor : Fry
```

(continues on next page)

(continued from previous page)

```
|              |- year : 4029
|              |- custumor : Planet Express
|
|- project/paths : { }
                    |- rawdatapath : "/tmp/here"
                    |- processedpath : "/tmp/there"
```

dicthandling.**deep_flatten**(*data: dict*) → collections.OrderedDict
Flattens a nested dictionary containing dictionaries to a single dictionary. Top level items will remain at root.

Parameters

- **data** (*dict*) – dictionary to be flatten
- **parentkey** (*string*) – Key of the parent dictionary. Default is *None*

Returns Ordered dictionary with a depth of 1.

Return type collections.OrderedDict

Raises ValueError – If data is not a dictionary.

dicthandling.**unflatten**(*data: dict*) → collections.OrderedDict
Creates a branch by given folded root dictionary with folded keys seperated by FOLDING_KEY_DELIMITER.

Parameters **data** (*dict*) – root dictionary folded by flatten

dicthandling.**deep_unflatten**(*root: dict*) → collections.OrderedDict
Reverses the result of deep_flatten returning a nested dictionary.

Parameters **root** (*dict*) – dictionary to be flatten

Returns Ordered dictionary with a depth of 1.

Return type collections.OrderedDict

CHAPTER 5

Storing and reading with the json module

dicthandling.**try_decoding_potential_json_content**(*bytelike_content*, *encoding_format_tryouts: List[str] = None*) → str

Tries to decode the given byte-like content as a text using the given encoding format types.

Notes

The first choice is 'utf-8', but in case of different OS are involved, some json files might been created using a different encoding, leading to errors. Therefore this methods tries the encondings listed in *dicthandling.ENCODING_FORMAT_TRYOUTS* by default.

Examples

```
>>> from dicthandling import try_decoding_potential_json_content
>>> sample = '{"a": "test", "json": "string with german literals äöüß"}'
>>> sample_latin_1 = sample.encode(encoding="latin-1")
>>> sample_latin_1
b'{"a": "test", "json": "string with german literals äöüß"}'
>>> try_decoding_potential_json_content(sample_latin_1)
'{"a": "test", "json": "string with german literals äöüß"}'
>>> sample_windows = sample.encode(encoding="windows-1252")
>>> sample_windows
b'{"a": "test", "json": "string with german literals äöüß"}'
>>> try_decoding_potential_json_content(sample_windows)
'{"a": "test", "json": "string with german literals äöüß"}'
```

Parameters

- **bytelike_content** – The text as byte-like object, which should be decoded.
- **encoding_format_tryouts** – List[str]: Formats in which the text might be encoded.

Raises UnicodeDecodeError

Returns Hopefully a proper decoded text.

Return type str

dicthandling.**put_into_json_file**(*filepath: str*, *data: dict*, *address: str = None*, ***json_settings*) → bool

Puts a dictionary into a existing json-file; if file does not exist it will be created then. Existing data within the json file not intersecting with the given `data` will be preserved.

This method will use indent to pretty print the output and disables the ensure_ascii option of the `json.dump` method to enable utf-8 characters.

Parameters

- **filepath** (*str*) – Filepath for the json data.
- **data** (*dict*) – data of type dict to be written into the json file.
- **address** (*str, optional*) – Additional address for the position within the preexisting data.
- ****json_settings** (*optional*) – Additional settings for the json.dump command. Default setting of this method are `ensure_ascii = False` and `indent = ' '`

Raises

- `OSError` – If the file cannot be opened.
- `FileNotFoundError` – If the file don't exists or cannot be created.

dicthandling.**read_from_json_file**(*filepath: str*, *address: str = None*, ***json_settings*) → dict

Reads an utf-8 encoded json-file returning its whole decoded content as a dictionary, if no *address* is supplied. If the *address* does not exist within the content *None* will be returned, otherwise returning the item at the supplied *address*.

Parameters

- **filepath** (*str*) – Filepath for the json data.
- **address** (*str, optional*) – Additional address for the position within the preexisting data.
- ****json_settings** (*dict, optional*) – Additional settings for the json.dump command. Default setting of this method are ensure_ascii=False and indent=' '

Returns content of requested *address*; *None* if address not found.

Return type Any

Raises

- `OSError` – If the file cannot be opened.
- `FileNotFoundError` – If the file don't exists or cannot be created.

Index