jsonextended Documentation

Release 0.7.11

Chris Sewell

Jun 18, 2019

Contents:

A module to extend the python json package functionality:

- Treat a directory structure like a nested dictionary
- **Lightweight plugin system**: define bespoke classes for parsing different file extensions (in-the-box: .json, .csv, .hdf5) and encoding/decoding objects.
- **Lazy Loading**: read files only when they are indexed into
- **Tab Completion**: index as tabs for quick exploration of data
- **Manipulation of nested structures**, including; filter, merge, diff, flatten, unflatten
- **Enhanced pretty printer**
- **Output to directory structure** (of n folder levels)
- **On-disk indexing** option for large json files (using the ijson package)
- **Units schema** concept to apply and convert physical quantities (using the pint package)
- All functions are thoroughly documented with tested examples.

CHAPTER 1

Installation

From Conda (recommended):

```
conda install -c conda-forge jsonextended
```

From PyPi:

```
pip install jsonextended
```

jsonextended has no import dependancies, on Python 3.x and only `pathlib2` on 2.7 but, for full functionality, it is advised to install the following packages:

```
conda install -c conda-forge ijson numpy pint  h5py pandas
```

CHAPTER 2

Basic Example

```
from jsonextended import edict, plugins, example_mockpaths
```

Take a directory structure, potentially containing multiple file types:

```
datadir = example_mockpaths.directory1
print(datadir.to_string(indentlvl=3,file_content=True))
```

```
Folder("dir1")
   File("file1.json") Contents:
    {"key2": {"key3": 4, "key4": 5}, "key1": [1, 2, 3]}
   Folder("subdir1")
     File("file1.csv") Contents:
       # a csv file
      header1,header2,header3
      val1,val2,val3
      val4,val5,val6
      val7,val8,val9
     File("file1.literal.csv") Contents:
       # a csv file with numbers
      header1,header2,header3
      1,1.1,string1
      2,2.2,string2
      3,3.3,string3
   Folder("subdir2")
     Folder("subsubdir21")
       File("file1.keypair") Contents:
         # a key-pair file
        key1 val1
        key2 val2
        key3 val3
        key4 val4
```

Plugins can be defined for parsing each file type (see *Creating Plugins* section):

```
plugins.load_builtin_plugins('parsers')
plugins.view_plugins('parsers')
```

```
{'csv.basic': 'read *.csv delimited file with headers to {header:[column_values]}',
 'csv.literal': 'read *.literal.csv delimited files with headers to {header:column_
↪values}, with number strings converted to int/float',
 'hdf5.read': 'read *.hdf5 (in read mode) files using h5py',
 'json.basic': 'read *.json files using json.load',
 'keypair': "read *.keypair, where each line should be; '<key> <pair>'"}
```

LazyLoad then takes a path name, path-like object or dict-like object, which will lazily load each file with a compatible plugin.

```
lazy = edict.LazyLoad(datadir)
lazy
```

```
{file1.json:..,subdir1:..,subdir2:..}
```

Lazyload can then be treated like a dictionary, or indexed by tab completion:

```
list(lazy.keys())
```

```
['subdir1', 'subdir2', 'file1.json']
```

```
lazy[['file1.json','key1']]
```

```
[1, 2, 3]
```

```
lazy.subdir1.file1_literal_csv.header2
```

```
[1.1, 2.2, 3.3]
```

For pretty printing of the dictionary:

```
edict.pprint(lazy,depth=2)
```

```
file1.json:
  key1: [1, 2, 3]
  key2: {...}
subdir1:
  file1.csv: {...}
  file1.literal.csv: {...}
subdir2:
  subsubdir21: {...}
```

Numerous functions exist to manipulate the nested dictionary:

```
edict.flatten(lazy.subdir1)
```

```
{('file1.csv', 'header1'): ['val1', 'val4', 'val7'],
 ('file1.csv', 'header2'): ['val2', 'val5', 'val8'],
 ('file1.csv', 'header3'): ['val3', 'val6', 'val9'],
 ('file1.literal.csv', 'header1'): [1, 2, 3],
```

(continues on next page)

(continued from previous page)

```
 ('file1.literal.csv', 'header2'): [1.1, 2.2, 3.3],
 ('file1.literal.csv', 'header3'): ['string1', 'string2', 'string3']}
```

LazyLoad parses the `plugins.decode` function to parser plugin's `read_file` method (keyword 'object_hook'). Therefore, bespoke decoder plugins can be set up for specific dictionary key signatures:

```
print(example_mockpaths.jsonfile2.to_string())
```

```
File("file2.json") Contents:
{"key1":{"_python_set_": [1, 2, 3]},"key2":{"_numpy_ndarray_": {"dtype": "int64",
↪"value": [1, 2, 3]}}}
```

```
edict.LazyLoad(example_mockpaths.jsonfile2).to_dict()
```

```
{u'key1': {u'_python_set_': [1, 2, 3]},
 u'key2': {u'_numpy_ndarray_': {u'dtype': u'int64', u'value': [1, 2, 3]}}}
```

```
plugins.load_builtin_plugins('decoders')
plugins.view_plugins('decoders')
```

```
{'decimal.Decimal': 'encode/decode Decimal type',
 'numpy.ndarray': 'encode/decode numpy.ndarray',
 'pint.Quantity': 'encode/decode pint.Quantity object',
 'python.set': 'decode/encode python set'}
```

```
dct = edict.LazyLoad(example_mockpaths.jsonfile2).to_dict()
dct
```

```
{u'key1': {1, 2, 3}, u'key2': array([1, 2, 3])}
```

This process can be reversed, using encoder plugins:

```
plugins.load_builtin_plugins('encoders')
plugins.view_plugins('encoders')
```

```
{'decimal.Decimal': 'encode/decode Decimal type',
 'numpy.ndarray': 'encode/decode numpy.ndarray',
 'pint.Quantity': 'encode/decode pint.Quantity object',
 'python.set': 'decode/encode python set'}
```

```
import json
json.dumps(dct,default=plugins.encode)
```

```
'{"key2": {"_numpy_ndarray_": {"dtype": "int64", "value": [1, 2, 3]}}, "key1": {"_
↪python_set_": [1, 2, 3]}}'
```

CHAPTER 3

Creating and Loading Plugins

```
from jsonextended import plugins, utils
```

Plugins are recognised as classes with a minimal set of attributes matching the plugin category interface:

```
plugins.view_interfaces()
```

```
{'decoders': ['plugin_name', 'plugin_descript', 'dict_signature'],
 'encoders': ['plugin_name', 'plugin_descript', 'objclass'],
 'parsers': ['plugin_name', 'plugin_descript', 'file_regex', 'read_file']}
```

```
plugins.unload_all_plugins()
plugins.view_plugins()
```

```
{'decoders': {}, 'encoders': {}, 'parsers': {}}
```

For example, a simple parser plugin would be:

```
class ParserPlugin(object):
    plugin_name = 'example'
    plugin_descript = 'a parser for *.example files, that outputs (line_number:line)'
    file_regex = '*.example'
    def read_file(self, file_obj, **kwargs):
        out_dict = {}
        for i, line in enumerate(file_obj):
            out_dict[i] = line.strip()
        return out_dict
```

Plugins can be loaded as a class:

```
plugins.load_plugin_classes([ParserPlugin],'parsers')
plugins.view_plugins()
```

```
{'decoders': {},
 'encoders': {},
 'parsers': {'example': 'a parser for *.example files, that outputs (line_number:line)
↪'}}
```

Or by directory (loading all .py files):

```
fobj = utils.MockPath('example.py',is_file=True,content="""
class ParserPlugin(object):
    plugin_name = 'example.other'
    plugin_descript = 'a parser for *.example.other files, that outputs (line_
↪number:line)'
    file_regex = '*.example.other'
    def read_file(self, file_obj, **kwargs):
        out_dict = {}
        for i, line in enumerate(file_obj):
            out_dict[i] = line.strip()
        return out_dict
""")
dobj = utils.MockPath(structure=[fobj])
plugins.load_plugins_dir(dobj,'parsers')
plugins.view_plugins()
```

```
{'decoders': {},
 'encoders': {},
 'parsers': {'example': 'a parser for *.example files, that outputs (line_number:line)
↪',
  'example.other': 'a parser for *.example.other files, that outputs (line_
↪number:line)'}}
```

For a more complex example of a parser, see `jsonextended.complex_parsers`

3.1 Interface specifications

- Parsers:
  - *file_regex* attribute, a str denoting what files to apply it to. A file will be parsed by the longest regex it matches.
  - *read_file* method, which takes an (open) file object and kwargs as parameters
- Decoders:
  - *dict_signature* attribute, a tuple denoting the keys which the dictionary must have, e.g. dict_signature=('a','b') decodes {'a':1,'b':2}
  - optionally, the attribute *allow_other_keys* = *True* can be set, to allow the dictionary to have more keys than the dict_signature and still be decoded, e.g. dict_signature=('a','b') would also decode {'a':1,'b':2,'c': 3}
  - *from_...* method(s), which takes a dict object as parameter. The `plugins.decode` function will use the method denoted by the intype parameter, e.g. if intype='json', then *from_json* will be called.
- Encoders:
  - *objclass* attribute, the object class to apply the encoding to, e.g. objclass=decimal.Decimal encodes objects of that type

- *to_...* method(s), which takes a dict object as parameter. The `plugins.encode` function will use the method denoted by the outtype parameter, e.g. if outtype='json', then *to_json* will be called.

CHAPTER 4

Extended Examples

For more information, all functions contain docstrings with tested examples.

4.1 Data Folders JSONisation

```
from jsonextended import ejson, edict, utils
```

```
path = utils.get_test_path()
ejson.jkeys(path)
```

```
['dir1', 'dir2', 'dir3']
```

```
jdict1 = ejson.to_dict(path)
edict.pprint(jdict1,depth=2)
```

```
dir1:
  dir1_1: {...}
  file1: {...}
  file2: {...}
dir2:
  file1: {...}
dir3:
```

```
edict.to_html(jdict1,depth=2)
```

To try the rendered JSON tree, output in the Jupyter Notebook, go to : https://chrisjsewell.github.io/

4.2 Nested Dictionary Manipulation

```
jdict2 = ejson.to_dict(path,['dir1','file1'])
edict.pprint(jdict2,depth=1)
```

```
initial: {...}
meta: {...}
optimised: {...}
units: {...}
```

```
filtered = edict.filter_keys(jdict2,['vol*'],use_wildcards=True)
edict.pprint(filtered)
```

```
initial:
  crystallographic:
    volume: 924.62752781
  primitive:
    volume: 462.313764
optimised:
  crystallographic:
    volume: 1063.98960509
  primitive:
    volume: 531.994803
```

```
edict.pprint(edict.flatten(filtered))
```

```
(initial, crystallographic, volume):   924.62752781
(initial, primitive, volume):          462.313764
(optimised, crystallographic, volume): 1063.98960509
(optimised, primitive, volume):        531.994803
```

4.3 Units Schema

```
from jsonextended.units import apply_unitschema, split_quantities
withunits = apply_unitschema(filtered,{'volume':'angstrom^3'})
edict.pprint(withunits)
```

```
initial:
  crystallographic:
    volume: 924.62752781 angstrom ** 3
  primitive:
    volume: 462.313764 angstrom ** 3
optimised:
  crystallographic:
    volume: 1063.98960509 angstrom ** 3
  primitive:
    volume: 531.994803 angstrom ** 3
```

```
newunits = apply_unitschema(withunits,{'volume':'nm^3'})
edict.pprint(newunits)
```

```
initial:
  crystallographic:
    volume: 0.92462752781 nanometer ** 3
  primitive:
    volume: 0.462313764 nanometer ** 3
optimised:
  crystallographic:
    volume: 1.06398960509 nanometer ** 3
  primitive:
    volume: 0.531994803 nanometer ** 3
```

```
edict.pprint(split_quantities(newunits),depth=4)
```

```
initial:
  crystallographic:
    volume:
      magnitude: 0.92462752781
      units:     nanometer ** 3
  primitive:
    volume:
      magnitude: 0.462313764
      units:     nanometer ** 3
optimised:
  crystallographic:
    volume:
      magnitude: 1.06398960509
      units:     nanometer ** 3
  primitive:
    volume:
      magnitude: 0.531994803
      units:     nanometer ** 3
```

CHAPTER 5

Summary of Functions

5.1 JSON Parsing

jkeys	get keys for initial json level, or at level after following key_path
to_dict	input json to dict

5.2 JSON/Dict Manipulation

LazyLoad	lazy load a dict_like object or file structure as a pseudo dictionary (works with all edict functions) supplies tab completion of keys
to_html	Pretty display dictionary in collapsible format with indents
apply	apply a function to all values with a certain leaf (terminal) key
combine_apply	combine values with certain leaf (terminal) keys by a function
combine_lists	combine lists of dicts
convert_type	convert all values of one type to another
diff	return the difference between two dict_like objects
dump	output dict to json
extract	extract section of dictionary
filter_keyfuncs	filters leaf nodes key:func(val) pairs of nested dictionary, where func(val) -> True/False
filter_keys	filter dict by certain keys
filter_keyvals	filters leaf nodes key:value pairs of nested dictionary
filter_paths	filter dict by certain paths containing key sets

Continued on next page

Table 2 – continued from previous page

filter_values	filters leaf nodes of nested dictionary
flatten	get nested dict as flat {key:val,...}, where key is tuple/string of all nested keys
flatten2d	get nested dict as {key:dict,...}, where key is tuple/string of all-1 nested keys
flattennd	get nested dict as {key:dict,...}, where key is tuple/string of all-n levels of nested keys
indexes	index dictionary by multiple keys
is_dict_like	test if object is dict like
is_iter_non_string	test if object is a list or tuple
is_list_of_dict_like	test if object is a list only containing dict like items
is_path_like	test if object is pathlib.Path like
list_to_dict	convert a list of dicts to a dict with root keys
merge	merge dicts, starting with dicts[1] into dicts[0]
pprint	print a nested dict in readable format
remove_keys	remove certain keys from nested dict, retaining preceeding paths
remove_keyvals	remove paths with at least one branch leading to certain (key,value) pairs from dict
remove_paths	remove paths containing certain keys from dict
rename_keys	rename keys in dict
split_key	split an existing key(s) into multiple levels
split_lists	split_lists key:list pairs into dicts for each item in the lists NB: will only split if all split_keys are present
to_json	output dict to json
unflatten	unflatten dictionary with keys as tuples or delimited strings

5.3 Physical Units

apply_unitschema	apply the unit schema to the data
combine_quantities	combine <unit,magnitude> pairs into pint.Quantity objects
get_in_units	get a value in the required units
split_quantities	split pint.Quantity objects into <unit,magnitude> pairs

CHAPTER 6

Package API

6.1 jsonextended package

6.1.1 Subpackages

jsonextended.encoders package

Submodules

jsonextended.encoders.decimals module

https://stackoverflow.com/questions/1960516/python-json-serialize-a-decimal-object

class jsonextended.encoders.decimals.**Encode_Decimal**
Bases: object

Examples

```
>>> from decimal import Decimal
>>> Encode_Decimal().to_str(Decimal('1.2345'))
'1.2345'
>>> Encode_Decimal().to_json(Decimal('1.2345'))
{'_python_Decimal_': '1.2345'}
>>> Encode_Decimal().from_json({'_python_Decimal_': '1.2345'})
Decimal('1.2345')
```

dict_signature = ['_python_Decimal_']

from_json(*obj*)

objclass
alias of decimal.Decimal

plugin_descript = 'encode/decode Decimal type'

plugin_name = 'decimal.Decimal'

to_json(*obj*)

to_str(*obj*)

jsonextended.encoders.fraction module

https://stackoverflow.com/questions/1960516/python-json-serialize-a-decimal-object

class jsonextended.encoders.fraction.**Encode_Fraction**
Bases: object

Examples

```
>>> from decimal import Decimal
>>> Encode_Fraction().to_str(Fraction(1, 3))
'1/3'
>>> Encode_Fraction().to_json(Fraction('1/3'))
{'_python_Fraction_': '1/3'}
>>> Encode_Fraction().from_json({'_python_Fraction_': '1/3'})
Fraction(1, 3)
```

dict_signature = ['_python_Fraction_']

from_json(*obj*)

objclass
alias of fractions.Fraction

plugin_descript = 'encode/decode Fraction type'

plugin_name = 'fractions.Fraction'

to_json(*obj*)

to_str(*obj*)

jsonextended.encoders.ndarray module

https://stackoverflow.com/questions/27909658/json-encoder-and-decoder-for-complex-numpy-arrays

class jsonextended.encoders.ndarray.**Encode_NDArray**
Bases: object

Examples

```
>>> from pprint import pprint
>>> import numpy as np
```

```
>>> Encode_NDArray().to_str(np.asarray([1,2,3]))
'[1 2 3]'
```

```
>>> pprint(Encode_NDArray().to_json(np.asarray([1,2,3])))
{'_numpy_ndarray_': {'dtype': 'int64', 'value': [1, 2, 3]}}
```

```
>>> Encode_NDArray().from_json({'_numpy_ndarray_': {'dtype': 'int64', 'value': [1,
↪ 2, 3]}})
array([1, 2, 3])
```

dict_signature = ['_numpy_ndarray_']

from_json(*obj*)

objclass
alias of numpy.ndarray

plugin_descript = 'encode/decode numpy.ndarray'

plugin_name = 'numpy.ndarray'

to_json(*obj*)

to_str(*obj*)

jsonextended.encoders.pint_quantity module

class jsonextended.encoders.pint_quantity.**Encode_Quantity**
Bases: object

Examples

```
>>> from pprint import pprint
>>> from pint import UnitRegistry
>>> ureg = UnitRegistry()
```

```
>>> print(Encode_Quantity().to_str(ureg.Quantity(1,'nanometre')))
1 nm
```

```
>>> pprint(Encode_Quantity().to_json(ureg.Quantity(1,'nanometre')))
{'_pint_Quantity_': {'Magnitude': 1, 'Units': 'nanometer'}}
```

```
>>> Encode_Quantity().from_json({'_pint_Quantity_': {'Magnitude': 1, 'Units':
↪'nanometer'}})
<Quantity(1, 'nanometer')>
```

dict_signature = ['_pint_Quantity_']

from_json(*obj*)

objclass
alias of pint.quantity._Quantity

plugin_descript = 'encode/decode pint.Quantity object'

plugin_name = 'pint.Quantity'

to_json(*obj*)

to_str(*obj*)

jsonextended.encoders.set module

class jsonextended.encoders.set.**Encode_Set**
Bases: object

Examples

```
>>> Encode_Set().to_str(set([1,2,3]))
'{1, 2, 3}'
```

```
>>> Encode_Set().to_json(set([1,2,3]))
{'_python_set_': [1, 2, 3]}
```

```
>>> list(Encode_Set().from_json({'_python_set_': [1, 2, 3]}))
[1, 2, 3]
```

dict_signature = ['_python_set_']

from_json(*obj*)

objclass
alias of builtins.set

plugin_descript = 'decode/encode python set'

plugin_name = 'python.set'

to_json(*obj*)

to_str(*obj*)

Module contents

jsonextended.parsers package

Submodules

jsonextended.parsers.csvs module

class jsonextended.parsers.csvs.**CSV_Parser**
Bases: object

Examples

```
>>> from pprint import pprint
```

```
>>> from jsonextended.utils import MockPath
>>> fileobj = MockPath(is_file=True,
... content='''# comment line
... head1,head2
... val1,val2
... val3,val4'''
```

(continues on next page)

(continued from previous page)

```
... )
>>> with fileobj.open() as f:
...     data = CSV_Parser().read_file(f)
>>> pprint(data)
{'head1': ['val1', 'val3'], 'head2': ['val2', 'val4']}
```

file_regex = '*.csv'

plugin_descript = 'read *.csv delimited file with headers to {header:[column_values]}'

plugin_name = 'csv.basic'

read_file(*file_obj*, ***kwargs*)

jsonextended.parsers.csvs_literal module

class jsonextended.parsers.csvs_literal.**CSVLiteral_Parser**

Bases: object

Examples

```
>>> from pprint import pprint
```

```
>>> from jsonextended.utils import MockPath
>>> fileobj = MockPath(is_file=True,
... content='''# comment line
... head1,head2
... 1.1,3
... 2.2,"3.3"'''
... )
>>> with fileobj.open() as f:
...     data = CSVLiteral_Parser().read_file(f)
>>> pprint(data)
{'head1': [1.1, 2.2], 'head2': [3, '3.3']}
```

file_regex = '*.literal.csv'

plugin_descript = 'read *.literal.csv delimited files with headers to {header:column_v
, s.t. values are converted to their python type

plugin_name = 'csv.literal'

read_file(*file_obj*, ***kwargs*)

static tryeval(*val*)

jsonextended.parsers.hdf5 module

class jsonextended.parsers.hdf5.**HDF5_Parser**

Bases: object

Examples

```
>>> import h5py
>>> indata = h5py.File('test.hdf5')
>>> dataset = indata.create_dataset("mydataset", (10,), dtype='i')
>>> indata.close()
```

```
>>> with open('test.hdf5') as f:
...     data = HDF5_Parser().read_file(f)
>>> data['mydataset'][:]
array([0, 0, 0, 0, 0, 0, 0, 0, 0, 0], dtype=int32)
```

```
>>> import os
>>> os.remove('test.hdf5')
```

file_regex = '*.hdf5'

plugin_descript = 'read *.hdf5 (in read mode) files using h5py'

plugin_name = 'hdf5.read'

read_file(*file_obj*, ***kwargs*)

jsonextended.parsers.ipynb module

class jsonextended.parsers.ipynb.**NBParser**
Bases: object

Examples

```
>>> from jsonextended.utils import MockPath
>>> from jsonextended.edict import pprint
>>> fileobj = MockPath(is_file=True,
... content='''{
... "cells":[],
... "metadata":{}
... }'''
... )
>>> with fileobj.open() as f:
...     data = NBParser().read_file(f)
>>> pprint(data)
cells: []
metadata:
```

file_regex = '*.ipynb'

plugin_descript = 'read Jupyter Notebooks'

plugin_name = 'ipynb'

read_file(*file_obj*, ***kwargs*)

jsonextended.parsers.jsons module

class jsonextended.parsers.jsons.**JSON_Parser**
Bases: object

Examples

```
>>> from jsonextended.utils import MockPath
>>> fileobj = MockPath(is_file=True,
... content='{"key1":[1,2,3]}'
... )
>>> with fileobj.open() as f:
...     data = JSON_Parser().read_file(f)
>>> list(data.values())
[[1, 2, 3]]
```

file_regex = '*.json'

plugin_descript = 'read *.json files using json.load'

plugin_name = 'json.basic'

read_file(*file_obj*, ***kwargs*)

jsonextended.parsers.keypairs module

class jsonextended.parsers.keypairs.**KeyPair_Parser**
Bases: object

Examples

```
>>> from pprint import pprint
>>> from jsonextended.utils import MockPath
>>> fileobj = MockPath(is_file=True,
... content='''# comment line
... key1 val1
... key2 val2
... key3 val3'''
... )
>>> with fileobj.open() as f:
...     data = KeyPair_Parser().read_file(f)
>>> pprint(data)
{'key1': 'val1', 'key2': 'val2', 'key3': 'val3'}
```

file_regex = '*.keypair'

plugin_descript = "read *.keypair, where each line should be; '<key> <pair>'"

plugin_name = 'keypair'

read_file(*file_obj*, ***kwargs*)

jsonextended.parsers.yaml module

class jsonextended.parsers.yaml.**YAML_Parser**
Bases: object

Examples

```
>>> from pprint import pprint
>>> from jsonextended.utils import MockPath
>>> fileobj = MockPath(is_file=True,
... content='key1:\n  subkey1: a\n  subkey2: [1,2,3]'
... )
>>> with fileobj.open() as f:
...     data = YAML_Parser().read_file(f)
>>> pprint(dict(data["key1"]))
{'subkey1': 'a', 'subkey2': [1, 2, 3]}
```

file_regex = '*.yaml'

plugin_descript = 'read *.yaml files using ruamel.yaml'

plugin_name = 'yaml.ruamel'

read_file(*file_obj*, ***kwargs*)

Module contents

a module to provide data parsers, from the native format to a json representation

jsonextended.units package

Submodules

jsonextended.units.core module

jsonextended.units.core.**apply_unitschema**(*data*, *uschema*, *as_quantity=True*, *raise_outerr=False*, *convert_base=False*, *use_wildcards=False*, *list_of_dicts=False*)

apply the unit schema to the data

Parameters

- **data** (*dict*) –
- **uschema** (*dict*) – units schema to apply
- **as_quantity** (*bool*) – if true, return values as pint.Quantity objects
- **raise_outerr** (*bool*) – raise error if a unit cannot be found in the outschema
- **convert_to_base** (*bool*) – rescale units to base units
- **use_wildcards** (*bool*) – if true, can use * (matches everything) and ? (matches any single character)
- **list_of_dicts** (*bool*) – treat list of dicts as additional branches

Examples

```
>>> from pprint import pprint
```

```
>>> data = {'energy':1,'x':[1,2],'other':{'y':[4,5]},'y':[4,5],'meta':None}
>>> uschema =   {'energy':'eV','x':'nm','other':{'y':'m'},'y':'cm'}
>>> data_units = apply_unitschema(data,uschema)
>>> pprint(data_units)
{'energy': <Quantity(1, 'electron_volt')>,
 'meta': None,
 'other': {'y': <Quantity([4 5], 'meter')>},
 'x': <Quantity([1 2], 'nanometer')>,
 'y': <Quantity([4 5], 'centimeter')>}
```

```
>>> newschema = {'energy':'kJ','other':{'y':'nm'},'y':'m'}
>>> new_data = apply_unitschema(data_units,newschema)
>>> str(new_data["energy"])
'1.60217653e-22 kilojoule'
>>> new_data["other"]["y"].magnitude.round(3).tolist(), str(new_data["other"]["y
↪"].units)
([4000000000.0, 5000000000.0], 'nanometer')
```

```
>>> old_data = apply_unitschema(new_data,uschema,as_quantity=False)
>>> old_data["energy"]
1.0
>>> old_data["other"]["y"].round(3).tolist()
[4.0, 5.0]
```

jsonextended.units.core.**combine_quantities**(*data*, *units='units'*, *magnitude='magnitude'*, *list_of_dicts=False*)

combine <unit,magnitude> pairs into pint.Quantity objects

Parameters

- **data** (*dict*) –
- **units** (*str*) – name of units key
- **magnitude** (*str*) – name of magnitude key
- **list_of_dicts** (*bool*) – treat list of dicts as additional branches

Examples

```
>>> from pprint import pprint
```

```
>>> sdata = {'energy': {'magnitude': 1.602e-22, 'units': 'kilojoule'},
...          'meta': None,
...          'other': {'y': {'magnitude': [4, 5, 6], 'units': 'nanometer'}},
...          'x': {'magnitude': [1, 2, 3], 'units': 'nanometer'},
...          'y': {'magnitude': [8,9,10], 'units': 'meter'}}
...
>>> combined_data = combine_quantities(sdata)
>>> pprint(combined_data)
{'energy': <Quantity(1.602e-22, 'kilojoule')>,
 'meta': None,
```

(continues on next page)

(continued from previous page)

```
 'other': {'y': <Quantity([4 5 6], 'nanometer')>},
 'x': <Quantity([1 2 3], 'nanometer')>,
 'y': <Quantity([ 8  9 10], 'meter')>}
```

jsonextended.units.core.**get_in_units**(*value*, *units*)

get a value in the required units

jsonextended.units.core.**split_quantities**(*data*, *units='units'*, *magnitude='magnitude'*, *list_of_dicts=False*)

split pint.Quantity objects into <unit,magnitude> pairs

Parameters

- **data** (*dict*) –
- **units** (*str*) – name for units key
- **magnitude** (*str*) – name for magnitude key
- **list_of_dicts** (*bool*) – treat list of dicts as additional branches

Examples

```
>>> from pprint import pprint
```

```
>>> from pint import UnitRegistry
>>> ureg = UnitRegistry()
>>> Q = ureg.Quantity
```

```
>>> qdata = {'energy': Q(1.602e-22, 'kilojoule'),
...          'meta': None,
...          'other': {'y': Q([4,5,6], 'nanometer')},
...          'x': Q([1,2,3], 'nanometer'),
...          'y': Q([8,9,10], 'meter')}
...
>>> split_data = split_quantities(qdata)
>>> pprint(split_data)
{'energy': {'magnitude': 1.602e-22, 'units': 'kilojoule'},
 'meta': None,
 'other': {'y': {'magnitude': array([4, 5, 6]), 'units': 'nanometer'}},
 'x': {'magnitude': array([1, 2, 3]), 'units': 'nanometer'},
 'y': {'magnitude': array([ 8,  9, 10]), 'units': 'meter'}}
```

Module contents

a package to manipulate the physical units of JSON data, via unitschema

jsonextended utilises the pint package to manage physical units. This was chosen due to its large user-base and support for numpy: https://socialcompare.com/en/comparison/python-units-quantities-packages-383avix4

6.1.2 Submodules

jsonextended.edict module

a module to manipulate python dictionary like objects

class jsonextended.edict.**LazyLoad**(*obj*, *ignore_regexes=('.*', '_*')*, *recursive=True*, *parent=None*, *key_paths=True*, *list_of_dicts=False*, *parse_errors=True*, ***parser_kwargs*)

Bases: object

lazy load a dict_like object or file structure as a pseudo dictionary (works with all edict functions) supplies tab completion of keys

Parameters

- **obj** (*dict or str or object*) – file like object or path to file
- **ignore_regexes** (*list[str]*) – ignore files and folders matching these regexes (can contain *, ? and [] wildcards)
- **recursive** (*bool*) – if True, load subdirectories
- **parent** (*object*) – the parent object of this instance
- **key_paths** (*bool*) – indicates if the keys of the object can be resolved as file/folder paths (to ensure strings do not get unintentionally treated as paths)
- **list_of_dicts** (*bool*) – treat list of dicts as additional branches
- **parse_errors** (*bool*) – if True, if parsing a file fails then an IOError will be raised if False, if parsing a file fails then only a logging.error will be made and the value will be returned as None
- **parser_kwargs** (*dict*) – additional keywords for parser plugins read_file method, (loaded decoder plugins are parsed by default)

Examples

```
>>> from jsonextended import plugins
>>> plugins.load_builtin_plugins()
[]
```

```
>>> l = LazyLoad({'a':{'b':2},3:4})
>>> print(l)
{3:..,a:..}
>>> l['a']
{b:..}
>>> l[['a','b']]
2
>>> l.a.b
2
>>> l.i3
4
```

```
>>> from jsonextended.utils import get_test_path
>>> from jsonextended.edict import pprint
```

```
>>> lazydict = LazyLoad(get_test_path())
>>> pprint(lazydict,depth=2)
```

(continues on next page)

(continued from previous page)

```
dir1:
  dir1_1: {...}
  file1.json: {...}
  file2.json: {...}
dir2:
  file1.csv: {...}
  file1.json: {...}
dir3:
file1.keypair:
  key1: val1
  key2: val2
  key3: val3
```

```
>>> 'dir1' in lazydict
True
```

```
>>> sorted(lazydict.keys())
['dir1', 'dir2', 'dir3', 'file1.keypair']
```

```
>>> sorted(lazydict.values())
[{}, {key1:..,key2:..,key3:..}, {file1.csv:..,file1.json:..}, {dir1_1:..,file1.
↪json:..,file2.json:..}]
```

```
>>> lazydict.dir1.file1_json
{initial:..,meta:..,optimised:..,units:..}
```

```
>>> ldict = lazydict.dir1.file1_json.to_dict()
>>> isinstance(ldict,dict)
True
>>> pprint(ldict,depth=1)
initial: {...}
meta: {...}
optimised: {...}
units: {...}
```

```
>>> lazydict = LazyLoad(get_test_path(),recursive=False)
>>> lazydict
{file1.keypair:..}
```

```
>>> lazydict = LazyLoad([{'a':{'b':{'c':1}}},{'a':2}],
...                     list_of_dicts=True)
>>> lazydict.i0.a.b.c
1
```

```
>>> LazyLoad([1,2,3])
Traceback (most recent call last):
 ...
ValueError: not an expandable object: [1, 2, 3]
```

```
>>> plugins.unload_all_plugins()
```

items() → list of D's (key, value) pairs, as 2-tuples

keys() → iter of D's keys

to_df(***kwargs*)
return the (fully loaded) structure as a pandas.DataFrame

to_dict()
return the (fully loaded) structure as a nested dictionary

to_obj()
return the internal object

values() → list of D's values

jsonextended.edict.**apply**(*d*, *leaf_key*, *func*, *new_name=None*, *remove_lkey=True*, *list_of_dicts=False*, *unflatten_level=0*, *deepcopy=True*, ***kwargs*)
apply a function to all values with a certain leaf (terminal) key

Parameters

- **d** (*dict*) –
- **leaf_key** (*str*) – name of leaf key
- **func** (*callable*) – function to apply
- **new_name** (*str*) – if not None, rename leaf_key
- **remove_lkey** (*bool*) – whether to remove original leaf_key (if new_name is not None)
- **list_of_dicts** (*bool*) – treat list of dicts as additional branches
- **unflatten_level** (*int or None*) – the number of levels to leave unflattened before combining, for instance if you need dicts as inputs
- **deepcopy** (*bool*) – deepcopy values
- **kwargs** (*dict*) – additional keywords to parse to function

Examples

```
>>> from pprint import pprint
>>> d = {'a':1,'b':1}
>>> func = lambda x: x+1
>>> pprint(apply(d,'a',func))
{'a': 2, 'b': 1}
>>> pprint(apply(d,'a',func,new_name='c'))
{'b': 1, 'c': 2}
>>> pprint(apply(d,'a',func,new_name='c', remove_lkey=False))
{'a': 1, 'b': 1, 'c': 2}
```

```
>>> test_dict = {"a":[{"b":[{"c":1, "d": 2}, {"e":3, "f": 4}]}, {"b":[{"c":5, "d
↪": 6}, {"e":7, "f": 8}]}]}
>>> pprint(apply(test_dict, "b", lambda x: x[-1], list_of_dicts=True, unflatten_
↪level=2))
{'a': [{'b': {'e': 3, 'f': 4}}, {'b': {'e': 7, 'f': 8}}]}
```

jsonextended.edict.**combine_apply**(*d*, *leaf_keys*, *func*, *new_name*, *unflatten_level=1*, *remove_lkeys=True*, *overwrite=False*, *list_of_dicts=False*, *deepcopy=True*, ***kwargs*)
combine values with certain leaf (terminal) keys by a function

Parameters

- **d** (*dict*) –

- **leaf_keys** (*list*) – names of leaf keys
- **func** (*callable*) – function to apply, must take at least len(leaf_keys) arguments
- **new_name** (*str*) – new key name
- **unflatten_level** (*int or None*) – the number of levels to leave unflattened before combining, for instance if you need dicts as inputs (None means all)
- **remove_lkeys** (*bool*) – whether to remove original leaf_keys
- **overwrite** (*bool*) – whether to overwrite any existing new_name key
- **list_of_dicts** (*bool*) – treat list of dicts as additional branches
- **deepcopy** (*bool*) – deepcopy values
- **kwargs** (*dict*) – additional keywords to parse to function

Examples

```
>>> from pprint import pprint
>>> d = {'a':1,'b':2}
>>> func = lambda x,y: x+y
>>> pprint(combine_apply(d,['a','b'],func,'c'))
{'c': 3}
>>> pprint(combine_apply(d,['a','b'],func,'c',remove_lkeys=False))
{'a': 1, 'b': 2, 'c': 3}
```

```
>>> d = {1:{'a':1,'b':2},2:{'a':4,'b':5},3:{'a':1}}
>>> pprint(combine_apply(d,['a','b'],func,'c'))
{1: {'c': 3}, 2: {'c': 9}, 3: {'a': 1}}
```

```
>>> func2 = lambda x: sorted(list(x.keys()))
>>> d2 = {'d':{'a':{'b':1,'c':2}}}
>>> pprint(combine_apply(d2,['a'],func2,'a',unflatten_level=2))
{'d': {'a': ['b', 'c']}}
```

jsonextended.edict.**combine_lists**(*d*, *keys=None*, *deepcopy=True*)

combine lists of dicts

Parameters

- **d** (*dict or list[dict]*) –
- **keys** (*list*) – keys to combine (all if None)
- **deepcopy** (*bool*) – deepcopy values

Example

```
>>> from pprint import pprint
>>> d = {'path_key': {'a': 1, 'split': [{'x': 1, 'y': 3}, {'x': 2, 'y': 4}]}}
>>> pprint(combine_lists(d,['split']))
{'path_key': {'a': 1, 'split': {'x': [1, 2], 'y': [3, 4]}}}
```

```
>>> combine_lists([{"a":2}, {"a":1}])
{'a': [2, 1]}
```

jsonextended.edict.**convert_type**(*d*, *intype*, *outtype*, *convert_list=True*, *in_place=True*)
convert all values of one type to another

Parameters

- **d** (*dict*) –
- **intype** (*type_class*) –
- **outtype** (*type_class*) –
- **convert_list** (*bool*) – whether to convert instances inside lists and tuples
- **in_place** (*bool*) – if True, applies conversions to original dict, else returns copy

Examples

```
>>> from pprint import pprint
```

```
>>> d = {'a':'1','b':'2'}
>>> pprint(convert_type(d,str,float))
{'a': 1.0, 'b': 2.0}
```

```
>>> d = {'a':['1','2']}
>>> pprint(convert_type(d,str,float))
{'a': [1.0, 2.0]}
```

```
>>> d = {'a':[('1','2'),[3,4]]}
>>> pprint(convert_type(d,str,float))
{'a': [(1.0, 2.0), [3, 4]]}
```

jsonextended.edict.**diff**(*new_dict*, *old_dict*, *iter_prefix='__iter__'*, *np_allclose=False*, ***kwargs*)
return the difference between two dict_like objects

Parameters

- **new_dict** (*dict*) –
- **old_dict** (*dict*) –
- **iter_prefix** (*str*) – prefix to use for list and tuple indexes
- **np_allclose** (*bool*) – if True, try using numpy.allclose to assess differences
- ****kwargs** – keyword arguments to parse to numpy.allclose

Returns

outcome – Containing none or more of:

- ”insertions” : list of (path, val)
- ”deletions” : list of (path, val)
- ”changes” : list of (path, (val1, val2))
- ”uncomparable” : list of (path, (val1, val2))

Return type dict

Examples

```
>>> from pprint import pprint
```

```
>>> diff({'a':1},{'a':1})
{}
```

```
>>> pprint(diff({'a': 1, 'b': 2, 'c': 5},{'b': 3, 'c': 4, 'd': 6}))
{'changes': [(('b',), (2, 3)), (('c',), (5, 4))],
 'deletions': [(('d',), 6)],
 'insertions': [(('a',), 1)]}
```

```
>>> pprint(diff({'a': [{"b":1}, {"c":2}, 1]},{'a': [{"b":1}, {"d":2}, 2]}))
{'changes': [(('a', '__iter__2'), (1, 2))],
 'deletions': [(('a', '__iter__1', 'd'), 2)],
 'insertions': [(('a', '__iter__1', 'c'), 2)]}
```

```
>>> diff({'a':1}, {'a':1+1e-10})
{'changes': [(('a',), (1, 1.0000000001))]}
```

```
>>> diff({'a':1}, {'a':1+1e-10}, np_allclose=True)
{}
```

jsonextended.edict.**dump**(*dct*, *jfile*, *overwrite=False*, *dirlevel=0*, *sort_keys=True*, *indent=2*, *default_name='root.json'*, ***kwargs*)

output dict to json

Parameters

- **dct** (*dict*) –
- **jfile** (*str or file_like*) – if file_like, must have write method
- **overwrite** (*bool*) – whether to overwrite existing files
- **dirlevel** (*int*) – if jfile is path to folder, defines how many key levels to set as sub-folders
- **sort_keys** (*bool*) – if true then the output of dictionaries will be sorted by key
- **indent** (*int*) – if non-negative integer, then JSON array elements and object members will be pretty-printed on new lines with that indent level spacing.
- **kwargs** (*dict*) – keywords for json.dump

jsonextended.edict.**extract**(*d*, *path=None*)

extract section of dictionary

Parameters

- **d** (*dict*) –
- **path** (*list[str]*) – keys to section

Returns

- **new_dict** (*dict*) – original, without extracted section
- **extract_dict** (*dict*) – extracted section

Examples

```
>>> from pprint import pprint
>>> d = {1:{"a":"A"},2:{"b":"B",'c':'C'}}
>>> pprint(extract(d,[2,'b']))
({1: {'a': 'A'}, 2: {'c': 'C'}}, {'b': 'B'})
```

jsonextended.edict.**filter_keyfuncs**(*d*, *keyfuncs*, *logic='OR'*, *keep_siblings=False*, *list_of_dicts=False*, *deepcopy=True*)

filters leaf nodes key:func(val) pairs of nested dictionary, where func(val) -> True/False

Parameters

- **d** (*dict*) –
- **keyfuncs** (*dict or list[tuple]*) – (key,funcs) pairs to filter by
- **logic** (*str*) – "OR" or "AND" for matching pairs
- **keep_siblings** (*bool*) – keep all sibling paths
- **list_of_dicts** (*bool*) – treat list of dicts as additional branches
- **deepcopy** (*bool*) – deepcopy values

Examples

```
>>> from pprint import pprint
```

```
>>> d = {'a':{'b':1,'c':2,'d':3}, 'e':4}
>>> func1 = lambda v: v <= 2
```

```
>>> pprint(filter_keyfuncs(d, {'b': func1, 'e': func1}, logic="OR", keep_
↪siblings=False))
{'a': {'b': 1}}
```

```
>>> pprint(filter_keyfuncs(d,[('b',func1), ('d', func1)], logic="OR", keep_
↪siblings=True))
{'a': {'b': 1, 'c': 2, 'd': 3}}
```

```
>>> pprint(filter_keyfuncs(d, {'b': func1, 'e': func1}, logic="AND", keep_
↪siblings=False))
{}
```

```
>>> pprint(filter_keyfuncs(d, {'b': func1, 'd': func1}, logic="AND", keep_
↪siblings=False))
{}
```

```
>>> pprint(filter_keyfuncs(d, {'b': func1, 'c': func1}, logic="AND", keep_
↪siblings=False))
{'a': {'b': 1, 'c': 2}}
```

```
>>> pprint(filter_keyfuncs(d,[('b',func1), ('c',func1)], logic="AND", keep_
↪siblings=True))
{'a': {'b': 1, 'c': 2, 'd': 3}}
```

jsonextended.edict.**filter_keys**(*d*, *keys*, *use_wildcards=False*, *list_of_dicts=False*, *deepcopy=True*)

filter dict by certain keys

Parameters

- **d** (*dict*) –
- **keys** (*list*) –
- **use_wildcards** (*bool*) – if true, can use * (matches everything) and ? (matches any single character)
- **list_of_dicts** (*bool*) – treat list of dicts as additional branches
- **deepcopy** (*bool*) – deepcopy values

Examples

```
>>> from pprint import pprint
```

```
>>> d = {1:{"a":"A"},2:{"b":"B"},4:{5:{6:'a',7:'b'}}}
>>> pprint(filter_keys(d,['a',6]))
{1: {'a': 'A'}, 4: {5: {6: 'a'}}}
```

```
>>> d = {1:{"axxxx":"A"},2:{"b":"B"}}
>>> pprint(filter_keys(d,['a*'],use_wildcards=True))
{1: {'axxxx': 'A'}}
```

jsonextended.edict.**filter_keyvals**(*d*, *keyvals*, *logic='OR'*, *keep_siblings=False*, *list_of_dicts=False*, *deepcopy=True*)

filters leaf nodes key:value pairs of nested dictionary

Parameters

- **d** (*dict*) –
- **keyvals** (*dict or list[tuple]*) – (key,value) pairs to filter by
- **logic** (*str*) – "OR" or "AND" for matching pairs
- **keep_siblings** (*bool*) – keep all sibling paths
- **list_of_dicts** (*bool*) – treat list of dicts as additional branches
- **deepcopy** (*bool*) – deepcopy values

Examples

```
>>> from pprint import pprint
```

```
>>> d = {1:{6:'a'},3:{7:'a'},2:{6:"b"},4:{5:{6:'a'}}}
>>> pprint(filter_keyvals(d,[(6,'a')]))
{1: {6: 'a'}, 4: {5: {6: 'a'}}}
```

```
>>> d2 = {'a':{'b':1,'c':2,'d':3}, 'e':4}
```

```
>>> pprint(filter_keyvals(d2, {'b': 1, 'e': 4}, logic="OR", keep_siblings=False))
{'a': {'b': 1}, 'e': 4}
```

```
>>> pprint(filter_keyvals(d2,[('b',1)], logic="OR", keep_siblings=True))
{'a': {'b': 1, 'c': 2, 'd': 3}}
```

```
>>> pprint(filter_keyvals(d2, {'b': 1, 'e': 4}, logic="AND", keep_siblings=False))
{}
```

```
>>> pprint(filter_keyvals(d2, {'b': 1, 'c': 2}, logic="AND", keep_siblings=False))
{'a': {'b': 1, 'c': 2}}
```

```
>>> pprint(filter_keyvals(d2,[('b',1), ('c',2)], logic="AND", keep_siblings=True))
{'a': {'b': 1, 'c': 2, 'd': 3}}
```

```
>>> d3 = {"a": {"b": 1, "f": {"d": 3}}, "e": {"b": 1, "c": 2, "f": {"d": 3}}, "g
↪": 5}
>>> pprint(filter_keyvals(d3,[('b',1), ('c', 2)], logic="OR", keep_siblings=True))
{'a': {'b': 1, 'f': {'d': 3}}, 'e': {'b': 1, 'c': 2, 'f': {'d': 3}}}
```

```
>>> pprint(filter_keyvals(d3,[('b',1), ('c', 2)], logic="AND", keep_
↪siblings=True))
{'e': {'b': 1, 'c': 2, 'f': {'d': 3}}}
```

jsonextended.edict.**filter_paths**(*d*, *paths*, *list_of_dicts=False*, *deepcopy=True*)

filter dict by certain paths containing key sets

Parameters

- **d** (*dict*) –
- **paths** (*list[str] or list[tuple]*) –
- **list_of_dicts** (*bool*) – treat list of dicts as additional branches
- **deepcopy** (*bool*) – deepcopy values

Examples

```
>>> from pprint import pprint
>>> d = {'a':{'b':1,'c':{'d':2}},'e':{'c':3}}
>>> filter_paths(d,[('c','d')])
{'a': {'c': {'d': 2}}}
```

```
>>> d2 = {'a':[{'b':1,'c':3},{'b':1,'c':2}]}
>>> pprint(filter_paths(d2,["b"],list_of_dicts=False))
{}
```

```
>>> pprint(filter_paths(d2,["c"],list_of_dicts=True))
{'a': [{'c': 3}, {'c': 2}]}
```

jsonextended.edict.**filter_values**(*d*, *vals=None*, *list_of_dicts=False*, *deepcopy=True*)

filters leaf nodes of nested dictionary

Parameters

- **d** (*dict*) –
- **vals** (*list*) – values to filter by
- **list_of_dicts** (*bool*) – treat list of dicts as additional branches
- **deepcopy** (*bool*) – deepcopy values

Examples

```
>>> d = {1:{"a":"A"},2:{"b":"B"},4:{5:{6:'a'}}}
>>> filter_values(d,['a'])
{4: {5: {6: 'a'}}}
```

jsonextended.edict.**flatten**(*d*, *key_as_tuple=True*, *sep='.'*, *list_of_dicts=None*, *all_iters=None*)

get nested dict as flat {key:val,...}, where key is tuple/string of all nested keys

Parameters

- **d** (*object*) –
- **key_as_tuple** (*bool*) – whether keys are list of nested keys or delimited string of nested keys
- **sep** (*str*) – if key_as_tuple=False, delimiter for keys
- **list_of_dicts** (*str or None*) – if not None, flatten lists of dicts using this prefix
- **all_iters** (*str or None*) – if not None, flatten all lists and tuples using this prefix

Examples

```
>>> from pprint import pprint
```

```
>>> d = {1:{"a":"A"}, 2:{"b":"B"}}
>>> pprint(flatten(d))
{(1, 'a'): 'A', (2, 'b'): 'B'}
```

```
>>> d = {1:{"a":"A"},2:{"b":"B"}}
>>> pprint(flatten(d,key_as_tuple=False))
{'1.a': 'A', '2.b': 'B'}
```

```
>>> d = [{'a':1},{'b':[1, 2]}]
>>> pprint(flatten(d,list_of_dicts='__list__'))
{('__list__0', 'a'): 1, ('__list__1', 'b'): [1, 2]}
```

```
>>> d = [{'a':1},{'b':[1, 2]}]
>>> pprint(flatten(d,all_iters='__iter__'))
{('__iter__0', 'a'): 1,
 ('__iter__1', 'b', '__iter__0'): 1,
 ('__iter__1', 'b', '__iter__1'): 2}
```

jsonextended.edict.**flatten2d**(*d*, *key_as_tuple=True*, *delim='.'*, *list_of_dicts=None*)

get nested dict as {key:dict,...}, where key is tuple/string of all-1 nested keys

NB: is same as flattennd(d,1,key_as_tuple,delim)

Parameters

- **d** (*dict*) –
- **key_as_tuple** (*bool*) – whether keys are list of nested keys or delimited string of nested keys
- **delim** (*str*) – if key_as_tuple=False, delimiter for keys
- **list_of_dicts** (*str or None*) – if not None, flatten lists of dicts using this prefix

Examples

```
>>> from pprint import pprint
```

```
>>> d = {1:{2:{3:{'b':'B','c':'C'},4:'D'}}}
>>> pprint(flatten2d(d))
{(1, 2): {4: 'D'}, (1, 2, 3): {'b': 'B', 'c': 'C'}}
```

```
>>> pprint(flatten2d(d,key_as_tuple=False,delim=','))
{'1,2': {4: 'D'}, '1,2,3': {'b': 'B', 'c': 'C'}}
```

jsonextended.edict.**flattennd**(*d*, *levels=0*, *key_as_tuple=True*, *delim='.'*, *list_of_dicts=None*)

get nested dict as {key:dict,...}, where key is tuple/string of all-n levels of nested keys

Parameters

- **d** (*dict*) –
- **levels** (*int*) – the number of levels to leave unflattened
- **key_as_tuple** (*bool*) – whether keys are list of nested keys or delimited string of nested keys
- **delim** (*str*) – if key_as_tuple=False, delimiter for keys
- **list_of_dicts** (*str or None*) – if not None, flatten lists of dicts using this prefix

Examples

```
>>> from pprint import pprint
```

```
>>> d = {1:{2:{3:{'b':'B','c':'C'},4:'D'}}}
>>> pprint(flattennd(d,0))
{(1, 2, 3, 'b'): 'B', (1, 2, 3, 'c'): 'C', (1, 2, 4): 'D'}
```

```
>>> pprint(flattennd(d,1))
{(1, 2): {4: 'D'}, (1, 2, 3): {'b': 'B', 'c': 'C'}}
```

```
>>> pprint(flattennd(d,2))
{(1,): {2: {4: 'D'}}, (1, 2): {3: {'b': 'B', 'c': 'C'}}}
```

```
>>> pprint(flattennd(d,3))
{(): {1: {2: {4: 'D'}}}, (1,): {2: {3: {'b': 'B', 'c': 'C'}}}}
```

```
>>> pprint(flattennd(d,4))
{(): {1: {2: {3: {'b': 'B', 'c': 'C'}, 4: 'D'}}}}
```

```
>>> pprint(flattennd(d,5))
{(): {1: {2: {3: {'b': 'B', 'c': 'C'}, 4: 'D'}}}}
```

```
>>> pprint(flattennd(d,1,key_as_tuple=False,delim='.'))
{'1.2': {4: 'D'}, '1.2.3': {'b': 'B', 'c': 'C'}}
```

```
>>> test_dict = {"a":[{"b":[{"c":1, "d": 2}, {"e":3, "f": 4}]}, {"b":[{"c":5, "d
↪": 6}, {"e":7, "f": 8}]}]}
>>> pprint(flattennd(test_dict, list_of_dicts="__list__", levels=2))
{('a', '__list__0', 'b'): [{'c': 1, 'd': 2}, {'e': 3, 'f': 4}],
 ('a', '__list__1', 'b'): [{'c': 5, 'd': 6}, {'e': 7, 'f': 8}]}
```

```
>>> pprint(flattennd(test_dict, list_of_dicts="__list__", levels=3))
{('a', '__list__0'): {'b': [{'c': 1, 'd': 2}, {'e': 3, 'f': 4}]},
 ('a', '__list__1'): {'b': [{'c': 5, 'd': 6}, {'e': 7, 'f': 8}]}}
```

jsonextended.edict.**indexes**(*dic*, *keys=None*)
index dictionary by multiple keys

Parameters

- **dic** (*dict*) –
- **keys** (*list*) –

Examples

```
>>> d = {1:{"a":"A"},2:{"b":"B"}}
>>> indexes(d,[1,'a'])
'A'
```

jsonextended.edict.**is_dict_like**(*obj*, *attr=('keys', 'items')*)
test if object is dict like

jsonextended.edict.**is_iter_non_string**(*obj*)
test if object is a list or tuple

jsonextended.edict.**is_list_of_dict_like**(*obj*, *attr=('keys', 'items')*)
test if object is a list only containing dict like items

jsonextended.edict.**is_path_like**(*obj*, *attr=('name', 'is_file', 'is_dir', 'iterdir')*)
test if object is pathlib.Path like

jsonextended.edict.**list_to_dict**(*lst*, *key=None*, *remove_key=True*)
convert a list of dicts to a dict with root keys

Parameters

- **lst** (*list[dict]*) –
- **key** (*str or None*) – a key contained by all of the dicts if None use index number string
- **remove_key** (*bool*) – remove key from dicts in list

Examples

```
>>> from pprint import pprint
>>> lst = [{'name':'f','b':1},{'name':'g','c':2}]
>>> pprint(list_to_dict(lst))
{'0': {'b': 1, 'name': 'f'}, '1': {'c': 2, 'name': 'g'}}
```

```
>>> pprint(list_to_dict(lst,'name'))
{'f': {'b': 1}, 'g': {'c': 2}}
```

jsonextended.edict.**merge**(*dicts*, *overwrite=False*, *append=False*, *list_of_dicts=False*)

merge dicts, starting with dicts[1] into dicts[0]

Parameters

- **dicts** (*list[dict]*) – list of dictionaries
- **overwrite** (*bool*) – if true allow overwriting of current data
- **append** (*bool*) – if true and items are both lists, then add them
- **list_of_dicts** (*bool*) – treat list of dicts as additional branches

Examples

```
>>> from pprint import pprint
```

```
>>> d1 = {1:{"a":"A"},2:{"b":"B"}}
>>> d2 = {1:{"a":"A"},2:{"c":"C"}}
>>> pprint(merge([d1,d2]))
{1: {'a': 'A'}, 2: {'b': 'B', 'c': 'C'}}
```

```
>>> d1 = {1:{"a":["A"]}}
>>> d2 = {1:{"a":["D"]}}
>>> pprint(merge([d1,d2],append=True))
{1: {'a': ['A', 'D']}}
```

```
>>> d1 = {1:{"a":"A"},2:{"b":"B"}}
>>> d2 = {1:{"a":"X"},2:{"c":"C"}}
>>> merge([d1,d2],overwrite=False)
Traceback (most recent call last):
...
ValueError: different data already exists at "1.a": old: A, new: X
```

```
>>> merge([{},{}],overwrite=False)
{}
>>> merge([{},{'a':1}],overwrite=False)
{'a': 1}
>>> pprint(merge([{},{'a':1},{'a':1},{'b':2}]))
{'a': 1, 'b': 2}
>>> pprint(merge([{'a':[{"b": 1}, {"c": 2}]}, {'a':[{"d": 3}]}]))
Traceback (most recent call last):
 ...
ValueError: different data already exists at "a": old: [{'b': 1}, {'c': 2}], new:␣
↪[{'d': 3}]
>>> pprint(merge([{'a':[{"b": 1}, {"c": 2}]}, {'a':[{"d": 3}]}], list_of_
↪dicts=True))
```

(continues on next page)

(continued from previous page)

```
Traceback (most recent call last):
  ...
ValueError: list of dicts are of different lengths at "a": old: [{'b': 1}, {'c':␣
↪2}], new: [{'d': 3}]
>>> pprint(merge([{'a':[{"b": 1}, {"c": 2}]}, {'a':[{"d": 3}, {"e": 4}]}], list_
↪of_dicts=True))
{'a': [{'b': 1, 'd': 3}, {'c': 2, 'e': 4}]}
```

jsonextended.edict.**pprint**(*d*, *lvlindent=2*, *initindent=0*, *delim=':'*, *max_width=80*, *depth=3*, *no_values=False*, *align_vals=True*, *print_func=None*, *keycolor=None*, *compress_lists=None*, *round_floats=None*, *_dlist=False*)

print a nested dict in readable format (- denotes an element in a list of dictionaries)

Parameters

- **d** (*object*) –
- **lvlindent** (*int*) – additional indentation spaces for each level
- **initindent** (*int*) – initial indentation spaces
- **delim** (*str*) – delimiter between key and value nodes
- **max_width** (*int*) – max character width of each line
- **depth** (*int or None*) – maximum levels to display
- **no_values** (*bool*) – whether to print values
- **align_vals** (*bool*) – whether to align values for each level
- **print_func** (*callable or None*) – function to print strings (print if None)
- **keycolor** (*None or str*) – if str, color keys by this color, allowed: red, green, yellow, blue, magenta, cyan, white
- **compress_lists** (*int*) –

  compress lists/tuples longer than this, e.g. [1,1,1,1,1,1] -> [1, 1,..., 1]
- **round_floats** (*int*) – significant figures for floats

Examples

```
>>> d = {'a':{'b':{'c':'Å','de':[4,5,[7,'x'],9]}}}
>>> pprint(d,depth=None)
a:
  b:
    c:  Å
    de: [4, 5, [7, x], 9]
>>> pprint(d,max_width=17,depth=None)
a:
  b:
    c:  Å
    de: [4, 5,
         [7, x],
         9]
>>> pprint(d,no_values=True,depth=None)
a:
```

(continues on next page)

(continued from previous page)

```
  b:
    c:
    de:
>>> pprint(d,depth=2)
a:
  b: {...}
>>> pprint({'a':[1,1,1,1,1,1,1,1]},
...        compress_lists=3)
a: [1, 1, 1, ...(x5)]
```

jsonextended.edict.**remove_keys**(*d*, *keys=None*, *use_wildcards=True*, *list_of_dicts=False*, *deepcopy=True*)

remove certain keys from nested dict, retaining preceeding paths

Parameters

- **keys** (*list*) –
- **use_wildcards** (*bool*) – if true, can use * (matches everything) and ? (matches any single character)
- **list_of_dicts** (*bool*) – treat list of dicts as additional branches
- **deepcopy** (*bool*) –

Examples

```
>>> from pprint import pprint
>>> d = {1:{"a":"A"},"a":{"b":"B"}}
>>> pprint(remove_keys(d,['a']))
{1: 'A', 'b': 'B'}
```

```
>>> pprint(remove_keys({'abc':1},['a*'],use_wildcards=False))
{'abc': 1}
>>> pprint(remove_keys({'abc':1},['a*'],use_wildcards=True))
{}
```

jsonextended.edict.**remove_keyvals**(*d*, *keyvals=None*, *list_of_dicts=False*, *deepcopy=True*)

remove paths with at least one branch leading to certain (key,value) pairs from dict

Parameters

- **d** (*dict*) –
- **keyvals** (*dict or list[tuple]*) – (key,value) pairs to remove
- **list_of_dicts** (*bool*) – treat list of dicts as additional branches

Examples

```
>>> from pprint import pprint
>>> d = {1:{"b":"A"},"a":{"b":"B","c":"D"},"b":{"a":"B"}}
>>> pprint(remove_keyvals(d,[("b","B")]))
{1: {'b': 'A'}, 'b': {'a': 'B'}}
```

```
>>> d2 = {'a':[{'b':1,'c':1},{'b':1,'c':2}]}
>>> pprint(remove_keyvals(d2,[("b",1)]))
{'a': [{'b': 1, 'c': 1}, {'b': 1, 'c': 2}]}
```

```
>>> pprint(remove_keyvals(d2,[("b",1)],list_of_dicts=True))
{}
```

jsonextended.edict.**remove_paths**(*d*, *keys*, *list_of_dicts=False*, *deepcopy=True*)
remove paths containing certain keys from dict

Parameters

- **d** (*dict*) –
- **keys** (*list*) – list of keys to find and remove path
- **list_of_dicts** (*bool*) – treat list of dicts as additional branches
- **deepcopy** (*bool*) – deepcopy values

Examples

```
>>> from pprint import pprint
>>> d = {1:{"a":"A"},2:{"b":"B"},4:{5:{6:'a',7:'b'}}}
>>> pprint(remove_paths(d,[6,'a']))
{2: {'b': 'B'}, 4: {5: {7: 'b'}}}
```

```
>>> d = {1:{2: 3}, 1:{4: 5}}
>>> pprint(remove_paths(d,[(1, 2)]))
{1: {4: 5}}
```

```
>>> d2 = {'a':[{'b':1,'c':{'b':3}},{'b':1,'c':2}]}
>>> pprint(remove_paths(d2,["b"],list_of_dicts=False))
{'a': [{'b': 1, 'c': {'b': 3}}, {'b': 1, 'c': 2}]}
```

```
>>> pprint(remove_paths(d2,["b"],list_of_dicts=True))
{'a': [{'c': 2}]}
```

jsonextended.edict.**rename_keys**(*d*, *keymap=None*, *list_of_dicts=False*, *deepcopy=True*)
rename keys in dict

Parameters

- **d** (*dict*) –
- **keymap** (*dict*) – dictionary of key name mappings
- **list_of_dicts** (*bool*) – treat list of dicts as additional branches
- **deepcopy** (*bool*) – deepcopy values

Examples

```
>>> from pprint import pprint
>>> d = {'a':{'old_name':1}}
>>> pprint(rename_keys(d,{'old_name':'new_name'}))
{'a': {'new_name': 1}}
```

jsonextended.edict.**split_key**(*d*, *key*, *new_keys*, *before=True*, *list_of_dicts=False*, *deepcopy=True*)

split an existing key(s) into multiple levels

Parameters

- **d** (*dict*) – or dict like
- **key** (*str*) – existing key value
- **new_keys** (*list[str]*) – new levels to add
- **before** (*bool*) – add level before existing key (else after)
- **list_of_dicts** (*bool*) – treat list of dicts as additional branches

Examples

```
>>> from pprint import pprint
>>> d = {'a':1,'b':2}
>>> pprint(split_key(d,'a',['c','d']))
{'b': 2, 'c': {'d': {'a': 1}}}
```

```
>>> pprint(split_key(d,'a',['c','d'],before=False))
{'a': {'c': {'d': 1}}, 'b': 2}
```

```
>>> d2 = [{'a':1},{'a':2},{'a':3}]
>>> pprint(split_key(d2,'a',['b'],list_of_dicts=True))
[{'b': {'a': 1}}, {'b': {'a': 2}}, {'b': {'a': 3}}]
```

jsonextended.edict.**split_lists**(*d*, *split_keys*, *new_name='split'*, *check_length=True*, *deepcopy=True*)

split_lists key:list pairs into dicts for each item in the lists NB: will only split if all split_keys are present

Parameters

- **d** (*dict*) –
- **split_keys** (*list*) – keys to split
- **new_name** (*str*) – top level key for split items
- **check_length** (*bool*) – if true, raise error if any lists are of a different length
- **deepcopy** (*bool*) – deepcopy values

Examples

```
>>> from pprint import pprint
```

```
>>> d = {'path_key':{'x':[1,2],'y':[3,4],'a':1}}
>>> new_d = split_lists(d,['x','y'])
>>> pprint(new_d)
{'path_key': {'a': 1, 'split': [{'x': 1, 'y': 3}, {'x': 2, 'y': 4}]}}
```

```
>>> split_lists(d,['x','a'])
Traceback (most recent call last):
...
ValueError: "a" data at the following path is not a list ('path_key',)
```

```
>>> d2 = {'path_key':{'x':[1,7],'y':[3,4,5]}}
>>> split_lists(d2,['x','y'])
Traceback (most recent call last):
...
ValueError: lists at the following path do not have the same size ('path_key',)
```

class jsonextended.edict.**to_html**(*obj*, *depth=2*, *max_length=20*, *max_height=600*, *sort=True*, *local=True*, *uniqueid=None*)

Bases: object

Pretty display dictionary in collapsible format with indents

Parameters

- **obj** (*str or dict*) – dict or json
- **depth** (*int*) – Depth of the json tree structure displayed, the rest is collapsed.
- **max_length** (*int*) – Maximum number of characters of a string displayed as preview, longer string appear collapsed.
- **max_height** (*int*) – Maxium height in pixels of containing box.
- **sort** (*bool*) – Whether the json keys are sorted alphabetically.
- **local** (*bool*) – use local version of javascript file
- **uniqueid** (*str*) – unique identifier (if None, auto-created)

Examples

```
>>> dic = {'sape': {'value': 22}, 'jack': 4098, 'guido': 4127}
>>> obj = to_html(dic, depth=1, max_length=10, sort=False, local=True, uniqueid=
↪'123')
>>> print(obj._repr_html_())
<style>
    .renderjson a              { text-decoration: none; }
    .renderjson .disclosure    { color: red;
                                 font-size: 125%; }
    .renderjson .syntax        { color: darkgrey; }
    .renderjson .string        { color: black; }
    .renderjson .number        { color: black; }
    .renderjson .boolean       { color: purple; }
    .renderjson .key           { color: royalblue; }
    .renderjson .keyword       { color: orange; }
    .renderjson .object.syntax { color: lightseagreen; }
    .renderjson .array.syntax  { color: lightseagreen; }
    </style><div id="123" style="max-height: 600px; width:100%%;"></div>
                <script>
require(["jsonextended/renderjson.js"], function() {
    document.getElementById("123").appendChild(
        renderjson.set_max_string_length(10)
            //.set_icons(circled plus, circled minus)
            .set_icons(String.fromCharCode(8853), String.fromCharCode(8854))
```

(continues on next page)

(continued from previous page)

```
            .set_sort_objects(false)
            .set_show_to_level(1)({"guido": 4127, "jack": 4098, "sape": {"value":␣
↪22}}))
});</script>
```

jsonextended.edict.**to_json**(*dct*, *jfile*, *overwrite=False*, *dirlevel=0*, *sort_keys=True*, *indent=2*, *default_name='root.json'*, ***kwargs*)

output dict to json

Parameters

- **dct** (*dict*) –
- **jfile** (*str or file_like*) – if file_like, must have write method
- **overwrite** (*bool*) – whether to overwrite existing files
- **dirlevel** (*int*) – if jfile is path to folder, defines how many key levels to set as sub-folders
- **sort_keys** (*bool*) – if true then the output of dictionaries will be sorted by key
- **indent** (*int*) – if non-negative integer, then JSON array elements and object members will be pretty-printed on new lines with that indent level spacing.
- **kwargs** (*dict*) – keywords for json.dump

Examples

```
>>> from jsonextended.utils import MockPath
>>> file_obj = MockPath('test.json',is_file=True,exists=False)
>>> dct = {'a':{'b':1}}
>>> to_json(dct, file_obj)
>>> print(file_obj.to_string())
File("test.json") Contents:
{
  "a": {
    "b": 1
  }
}
```

```
>>> from jsonextended.utils import MockPath
>>> folder_obj = MockPath()
>>> dct = {'x':{'a':{'b':1},'c':{'d':3}}}
>>> to_json(dct, folder_obj, dirlevel=0,indent=None)
>>> print(folder_obj.to_string(file_content=True))
Folder("root")
  File("x.json") Contents:
   {"a": {"b": 1}, "c": {"d": 3}}
```

```
>>> folder_obj = MockPath()
>>> to_json(dct, folder_obj, dirlevel=1,indent=None)
>>> print(folder_obj.to_string(file_content=True))
Folder("root")
  Folder("x")
    File("a.json") Contents:
     {"b": 1}
```

(continues on next page)

(continued from previous page)

```
    File("c.json") Contents:
     {"d": 3}
```

jsonextended.edict.**unflatten**(*d*, *key_as_tuple=True*, *delim='.'*, *list_of_dicts=None*, *deep-copy=True*)

unflatten dictionary with keys as tuples or delimited strings

Parameters

- **d** (*dict*) –
- **key_as_tuple** (*bool*) – if true, keys are tuples, else, keys are delimited strings
- **delim** (*str*) – if keys are strings, then split by delim
- **list_of_dicts** (*str or None*) – if key starts with this treat as a list

Examples

```
>>> from pprint import pprint
```

```
>>> d = {('a','b'):1,('a','c'):2}
>>> pprint(unflatten(d))
{'a': {'b': 1, 'c': 2}}
```

```
>>> d2 = {'a.b':1,'a.c':2}
>>> pprint(unflatten(d2,key_as_tuple=False))
{'a': {'b': 1, 'c': 2}}
```

```
>>> d3 = {('a','__list__1', 'a'): 1, ('a','__list__0', 'b'): 2}
>>> pprint(unflatten(d3,list_of_dicts='__list__'))
{'a': [{'b': 2}, {'a': 1}]}
```

```
>>> unflatten({('a','b','c'):1,('a','b'):2})
Traceback (most recent call last):
...
KeyError: "child conflict for path: ('a', 'b'); 2 and {'c': 1}"
```

jsonextended.ejson module

jsonextended.ejson.**jkeys**(*jfile*, *key_path=None*, *in_memory=True*, *ignore_prefix=('.', '_')*)

get keys for initial json level, or at level after following key_path

Parameters

- **jfile** (*str, file_like or path_like*) – if str, must be existing file or folder, if file_like, must have 'read' method if path_like, must have 'iterdir' method (see pathlib.Path)
- **key_path** (*list[str]*) – a list of keys to index into the json before returning keys
- **in_memory** (*bool*) – if true reads json into memory before finding keys (this is faster but uses more memory)
- **ignore_prefix** (*list[str]*) – ignore folders beginning with these prefixes

Examples

```
>>> from jsonextended.utils import MockPath
>>> file_obj = MockPath('test.json',is_file=True,
... content='''
... {
...  "a": 1,
...  "b": [1.1,2.1],
...  "c": {"d":"e","f":"g"}
... }
... ''')
...
>>> jkeys(file_obj)
['a', 'b', 'c']
```

```
>>> jkeys(file_obj,["c"])
['d', 'f']
```

```
>>> from jsonextended.utils import get_test_path
>>> path = get_test_path()
>>> jkeys(path)
['dir1', 'dir2', 'dir3']
```

```
>>> path = get_test_path()
>>> jkeys(path, ['dir1','file1'], in_memory=True)
['initial', 'meta', 'optimised', 'units']
```

jsonextended.ejson.**to_dict**(*jfile*, *key_path=None*, *in_memory=True*, *ignore_prefix=('.', '_')*, *parse_decimal=False*)

input json to dict

Parameters

- **jfile** (*str, file_like or path_like*) – if str, must be existing file or folder, if file_like, must have 'read' method if path_like, must have 'iterdir' method (see pathlib.Path)
- **key_path** (*list[str]*) – a list of keys to index into the json before parsing it
- **in_memory** (*bool*) – if true reads full json into memory before filtering keys (this is faster but uses more memory)
- **ignore_prefix** (*list[str]*) – ignore folders beginning with these prefixes
- **parse_decimal** (*bool*) – whether to parse numbers as Decimal instances (retains exact precision)

Examples

```
>>> from pprint import pformat
```

```
>>> from jsonextended.utils import MockPath
>>> file_obj = MockPath('test.json',is_file=True,
... content='''
... {
...  "a": 1,
...  "b": [1.1,2.1],
```

(continues on next page)

(continued from previous page)

```
...   "c": {"d":"e"}
... }
... ''')
...
```

```
>>> dstr = pformat(to_dict(file_obj))
>>> print(dstr.replace("u'","'"))
{'a': 1, 'b': [1.1, 2.1], 'c': {'d': 'e'}}
```

```
>>> dstr = pformat(to_dict(file_obj,parse_decimal=True))
>>> print(dstr.replace("u'","'"))
{'a': 1, 'b': [Decimal('1.1'), Decimal('2.1')], 'c': {'d': 'e'}}
```

```
>>> str(to_dict(file_obj,["c","d"]))
'e'
```

```
>>> from jsonextended.utils import get_test_path
>>> path = get_test_path()
>>> jdict1 = to_dict(path)
>>> pprint(jdict1,depth=2)
dir1:
  dir1_1: {...}
  file1: {...}
  file2: {...}
dir2:
  file1: {...}
dir3:
```

```
>>> jdict2 = to_dict(path,['dir1','file1','initial'],in_memory=False)
>>> pprint(jdict2,depth=1)
crystallographic: {...}
primitive: {...}
```

jsonextended.example_mockpaths module

mock files and folder structure for testing

Examples

```
>>> jsonfile1
MockFile("dir1/file1.json")
>>> jsonfile2
MockFile("file2.json")
>>> csvfile1
MockFile("dir1/subdir1/file1.csv")
>>> csvfile2
MockFile("dir1/subdir1/file1.literal.csv")
>>> kpfile1
MockFile("dir1/subdir2/subsubdir21/file1.keypair")
```

```
>>> print(directory1.to_string(indentlvl=3,file_content=False))
Folder("dir1")
   File("file1.json")
   Folder("subdir1")
     File("file1.csv")
     File("file1.literal.csv")
   Folder("subdir2")
     Folder("subsubdir21")
       File("file1.keypair")
```

```
>>> print(directory1.to_string(indentlvl=3,file_content=True))
Folder("dir1")
   File("file1.json") Contents:
    {"key2": {"key3": 4, "key4": 5}, "key1": [1, 2, 3]}
   Folder("subdir1")
     File("file1.csv") Contents:
       # a csv file
      header1,header2,header3
      val1,val2,val3
      val4,val5,val6
      val7,val8,val9
     File("file1.literal.csv") Contents:
       # a csv file with numbers
      header1,header2,header3
      1,1.1,string1
      2,2.2,string2
      3,3.3,string3
   Folder("subdir2")
     Folder("subsubdir21")
       File("file1.keypair") Contents:
         # a key-pair file
        key1 val1
        key2 val2
        key3 val3
        key4 val4
```

jsonextended.mockpath module

class jsonextended.mockpath.**MockPath**(*path='root'*, *is_file=False*, *exists=True*, *structure=()*, *content=''*, *parent=None*)

Bases: object

a mock path, mimicking pathlib.Path, supporting context open method for read/write

Parameters

- **path** (*str*) – the path string
- **is_file** (*bool*) – if True is file, else folder
- **content** (*str*) – content of the file
- **structure** – structure of the directory

Examples

```
>>> file_obj = MockPath("path/to/test.txt",is_file=True,
...                     content="line1\nline2\nline3")
...
>>> file_obj
MockFile("path/to/test.txt")
>>> file_obj.name
'test.txt'
>>> file_obj.parent
MockFolder("path/to")
>>> print(str(file_obj))
path/to/test.txt
>>> print(file_obj.to_string())
File("test.txt") Contents:
line1
line2
line3
>>> file_obj.is_file()
True
>>> file_obj.is_dir()
False
>>> with file_obj.open('r') as f:
...     print(f.readline().strip())
line1
>>> with file_obj.open('w') as f:
...     f.write('newline1\nnewline2')
>>> print(file_obj.to_string())
File("test.txt") Contents:
newline1
newline2
```

```
>>> with file_obj.maketemp() as temp:
...     with temp.open() as f:
...         print(f.readline().strip())
newline1
```

```
>>> dir_obj = MockPath(
...   structure=[{'dir1':[{'subdir':[file_obj.copy_path_obj()]},file_obj.copy_
↪path_obj()]},
...              {'dir2':[file_obj.copy_path_obj()]},file_obj.copy_path_obj()]
... )
>>> dir_obj
MockFolder("root")
>>> dir_obj.name
'root'
>>> dir_obj.is_file()
False
>>> dir_obj.is_dir()
True
>>> print(dir_obj.to_string())
Folder("root")
  Folder("dir1")
    Folder("subdir")
      File("test.txt")
    File("test.txt")
  Folder("dir2")
```

(continues on next page)

(continued from previous page)

```
    File("test.txt")
  File("test.txt")
```

```
>>> "dir1/test.txt" in dir_obj
True
```

```
>>> dir_obj["dir1/test.txt"]
MockFile("root/dir1/test.txt")
```

```
>>> list(dir_obj.iterdir())
[MockFolder("root/dir1"), MockFolder("root/dir2"), MockFile("root/test.txt")]
```

```
>>> list(dir_obj.glob("*"))
[MockFolder("root/dir1"), MockFolder("root/dir2"), MockFile("root/test.txt")]
```

```
>>> list(dir_obj.glob("dir1/*"))
[MockFolder("root/dir1/subdir"), MockFile("root/dir1/test.txt")]
```

```
>>> list(dir_obj.glob("**"))
[MockFolder("root/dir1"), MockFolder("root/dir1/subdir"), MockFolder("root/dir2")]
```

```
>>> list(dir_obj.glob("**/*"))
[MockFolder("root/dir1"), MockFolder("root/dir1/subdir"), MockFile("root/dir1/
↪subdir/test.txt"), MockFile("root/dir1/test.txt"), MockFolder("root/dir2"),␣
↪MockFile("root/dir2/test.txt"), MockFile("root/test.txt")]
```

>>> list(dir_obj.glob("**/dir1")) # [MockFolder("root/dir1")]

```
>>> new = dir_obj.joinpath('dir3')
>>> new.mkdir()
>>> list(dir_obj.iterdir())
[MockFolder("root/dir1"), MockFolder("root/dir2"), MockFolder("root/dir3"),␣
↪MockFile("root/test.txt")]
```

```
>>> dir_obj.joinpath("test.txt").unlink()
>>> list(dir_obj.iterdir())
[MockFolder("root/dir1"), MockFolder("root/dir2"), MockFolder("root/dir3")]
```

```
>>> dir_obj.joinpath("dir3").rmdir()
>>> list(dir_obj.iterdir())
[MockFolder("root/dir1"), MockFolder("root/dir2")]
```

```
>>> print(dir_obj.to_string())
Folder("root")
  Folder("dir1")
    Folder("subdir")
      File("test.txt")
    File("test.txt")
  Folder("dir2")
    File("test.txt")
```

```
>>> dir_obj.joinpath("dir1/subdir")
MockFolder("root/dir1/subdir")
```

```
>>> dir_obj.joinpath("dir1", "subdir")
MockFolder("root/dir1/subdir")
```

```
>>> new = dir_obj.joinpath("dir1/subdir/other")
>>> new
MockVirtualPath("root/dir1/subdir/other")
```

```
>>> new.touch()
>>> new
MockFile("root/dir1/subdir/other")
```

```
>>> new.unlink()
>>> new
MockVirtualPath("root/dir1/subdir/other")
```

```
>>> new.mkdir()
>>> new
MockFolder("root/dir1/subdir/other")
```

```
>>> newfile = MockPath('newfile.txt', is_file=True)
>>> new.copy_from(newfile)
>>> print(new.to_string())
Folder("other")
  File("newfile.txt")
```

```
>>> file_obj = MockPath("newfile2.txt",is_file=True, content='test')
>>> file_obj.copy_to(new)
>>> print(new.to_string())
Folder("other")
  File("newfile.txt")
  File("newfile2.txt")
```

```
>>> file_obj.name = "newfile3.txt"
>>> with file_obj.maketemp() as temp:
...     new.copy_from(temp)
>>> print(new.to_string())
Folder("other")
  File("newfile.txt")
  File("newfile2.txt")
  File("newfile3.txt")
```

```
>>> print(new.copy_path_obj().to_string())
Folder("other")
  File("newfile.txt")
  File("newfile2.txt")
  File("newfile3.txt")
```

```
>>> with new.maketemp(getoutput=True) as tempdir:
...     tempdir.joinpath("new").mkdir()
...     tempdir.joinpath("new/file.txt").touch()
```

(continues on next page)

(continued from previous page)

```
>>> print(new.to_string())
Folder("other")
  Folder("new")
    File("file.txt")
  File("newfile.txt")
  File("newfile2.txt")
  File("newfile3.txt")
```

absolute()

add_child(*child*)

children

chmod(*mode*)
Change the mode (permissions) of a file

Parameters

- **path** (*str*) –
- **mode** (*int*) – new permissions (see os.chmod)

Examples

To make a file executable cur_mode = folder.stat("exec.sh").st_mode folder.chmod("exec.sh", cur_mode | stat.S_IXUSR | stat.S_IXGRP | stat.S_IXOTH)

copy_from(*source*)
copy from a source to a mock directory

Parameters **source** (*str or pathlib.Path or MockPath*) – any file like object or path to a file

copy_path_obj()
copy mock path (removing path and parent)

copy_to(*target*)
copy from a mock path to a target

Parameters **target** (*str, pathlib.Path or MockPath*) –

exists()

file_content

glob(*regex*, *recurse=False*, *toplevel=True*)

Parameters

- **regex** (*str*) – the path regex, with * to match 0 or more (non-recursive) paths and ** to match zero or more (recursive) directories
- **recurse** (*bool*) –

Yields **path** (*MockPath*)

is_dir()

is_file()

iterdir()

joinpath(**paths*)

maketemp(*getoutput=False*, *dir=None*)
make a named temporary file or folder containing the path contents

Parameters

- **getoutput** (*bool*) – if True, (on exit) new paths will be read/added to the path
- **dir** (*None or str*) – directory to place temp in (see tempfile.mkstemp)

Yields **temppath** (*path.Path*) – path to temporary

mkdir(*parents=False*)

Parameters

- **mode** –
- **parents** (*bool*) – If True, any missing parents of this path are created as needed If False, a missing parent raises FileNotFoundError.

name

open(*mode='r'*, *encoding=None*)
context manager for opening a file

Parameters

- **mode** (*str*) –
- **encoding** (*None or str*) –

parent

path

relative_to(*other*)

rename(*target*)

rmdir()

samefile(*other*)

stat()
Retrieve information about a file

Parameters **path** (*str*) –

Returns **attr** – see os.stat, includes st_mode, st_size, st_uid, st_gid, st_atime, and st_mtime attributes

Return type object

to_string(*indentlvl=2*, *file_content=False*, *color=False*)
convert to string

touch()

unlink()

jsonextended.mockpath.**colortxt**(*text*, *color=None*, *on_color=None*, *attrs=None*)
Colorize text.

Available text colors: red, green, yellow, blue, magenta, cyan, white.

Available text highlights: on_red, on_green, on_yellow, on_blue, on_magenta, on_cyan, on_white.

Available attributes: bold, dark, underline, blink, reverse, concealed.

Examples

```
>>> txt = colortxt('Hello, World!', 'red', 'on_grey', ['bold'])
>>> print(txt)
[1m[40m[31mHello, World![0m
```

jsonextended.plugins module

jsonextended.plugins.**decode**(*dct*, *intype='json'*, *raise_error=False*)
decode dict objects, via decoder plugins, to new type

Parameters

- **intype** (*str*) – use decoder method from_<intype> to encode
- **raise_error** (*bool*) – if True, raise ValueError if no suitable plugin found

Examples

```
>>> load_builtin_plugins('decoders')
[]
```

```
>>> from decimal import Decimal
>>> decode({'_python_Decimal_':'1.3425345'})
Decimal('1.3425345')
```

```
>>> unload_all_plugins()
```

jsonextended.plugins.**encode**(*obj*, *outtype='json'*, *raise_error=False*)
encode objects, via encoder plugins, to new types

Parameters

- **outtype** (*str*) – use encoder method to_<outtype> to encode
- **raise_error** (*bool*) – if True, raise ValueError if no suitable plugin found

Examples

```
>>> load_builtin_plugins('encoders')
[]
```

```
>>> from decimal import Decimal
>>> encode(Decimal('1.3425345'))
{'_python_Decimal_': '1.3425345'}
>>> encode(Decimal('1.3425345'),outtype='str')
'1.3425345'
```

```
>>> encode(set([1,2,3,4,4]))
{'_python_set_': [1, 2, 3, 4]}
>>> encode(set([1,2,3,4,4]),outtype='str')
'{1, 2, 3, 4}'
```

```
>>> unload_all_plugins()
```

jsonextended.plugins.**get_plugins**(*category*)
get plugins for category

jsonextended.plugins.**load_builtin_plugins**(*category=None*, *overwrite=False*)
load plugins from builtin directories

Parameters

- **name** (*None or str*) –
- **category** (*None or str*) – if str, apply for single plugin category

Examples

```
>>> from pprint import pprint
>>> pprint(view_plugins())
{'decoders': {}, 'encoders': {}, 'parsers': {}}
```

```
>>> errors = load_builtin_plugins()
>>> errors
[]
```

```
>>> pprint(view_plugins(),width=200)
{'decoders': {'decimal.Decimal': 'encode/decode Decimal type',
              'fractions.Fraction': 'encode/decode Fraction type',
              'numpy.ndarray': 'encode/decode numpy.ndarray',
              'pint.Quantity': 'encode/decode pint.Quantity object',
              'python.set': 'decode/encode python set'},
 'encoders': {'decimal.Decimal': 'encode/decode Decimal type',
              'fractions.Fraction': 'encode/decode Fraction type',
              'numpy.ndarray': 'encode/decode numpy.ndarray',
              'pint.Quantity': 'encode/decode pint.Quantity object',
              'python.set': 'decode/encode python set'},
 'parsers': {'csv.basic': 'read *.csv delimited file with headers to
↪{header:[column_values]}',
             'csv.literal': 'read *.literal.csv delimited files with headers to
↪{header:column_values}, with number strings converted to int/float',
             'hdf5.read': 'read *.hdf5 (in read mode) files using h5py',
             'ipynb': 'read Jupyter Notebooks',
             'json.basic': 'read *.json files using json.load',
             'keypair': "read *.keypair, where each line should be; '<key> <pair>'
↪",
             'yaml.ruamel': 'read *.yaml files using ruamel.yaml'}}
```

```
>>> unload_all_plugins()
```

jsonextended.plugins.**load_plugin_classes**(*classes*, *category=None*, *overwrite=False*)
load plugins from class objects

Parameters

- **classes** (*list*) – list of classes
- **category** (*None or str*) – if str, apply for single plugin category
- **overwrite** (*bool*) – if True, allow existing plugins to be overwritten

Examples

```
>>> from pprint import pprint
>>> pprint(view_plugins())
{'decoders': {}, 'encoders': {}, 'parsers': {}}
```

```
>>> class DecoderPlugin(object):
...     plugin_name = 'example'
...     plugin_descript = 'a decoder for dicts containing _example_ key'
...     dict_signature = ('_example_',)
...
>>> errors = load_plugin_classes([DecoderPlugin])
```

```
>>> pprint(view_plugins())
{'decoders': {'example': 'a decoder for dicts containing _example_ key'},
 'encoders': {},
 'parsers': {}}
```

```
>>> unload_all_plugins()
```

jsonextended.plugins.**load_plugins_dir**(*path*, *category=None*, *overwrite=False*)
load plugins from a directory

Parameters

- **path** (*str or path_like*) –
- **category** (*None or str*) – if str, apply for single plugin category
- **overwrite** (*bool*) – if True, allow existing plugins to be overwritten

jsonextended.plugins.**load_source**(*modname*, *fname*)

jsonextended.plugins.**parse**(*fpath*, ***kwargs*)
parse file contents, via parser plugins, to dict like object NB: the longest file regex will be used from plugins

Parameters

- **fpath** (*file_like*) – string, object with 'open' and 'name' attributes, or object with 'readline' and 'name' attributes
- **kwargs** – to pass to parser plugin

Examples

```
>>> load_builtin_plugins('parsers')
[]
```

```
>>> from pprint import pformat
```

```
>>> json_file = StringIO('{"a":[1,2,3.4]}')
>>> json_file.name = 'test.json'
```

```
>>> dct = parse(json_file)
>>> print(pformat(dct).replace("u'","'"))
{'a': [1, 2, 3.4]}
```

```
>>> reset = json_file.seek(0)
>>> from decimal import Decimal
>>> dct = parse(json_file, parse_float=Decimal,other=1)
>>> print(pformat(dct).replace("u'","'"))
{'a': [1, 2, Decimal('3.4')]}
```

```
>>> class NewParser(object):
...     plugin_name = 'example'
...     plugin_descript = 'loads test.json files'
...     file_regex = 'test.json'
...     def read_file(self, file_obj, **kwargs):
...         return {'example':1}
>>> load_plugin_classes([NewParser],'parsers')
[]
>>> reset = json_file.seek(0)
>>> parse(json_file)
{'example': 1}
```

```
>>> unload_all_plugins()
```

jsonextended.plugins.**parser_available**(*fpath*)

test if parser plugin available for fpath

Examples

```
>>> load_builtin_plugins('parsers')
[]
>>> test_file = StringIO('{"a":[1,2,3.4]}')
>>> test_file.name = 'test.json'
>>> parser_available(test_file)
True
>>> test_file.name = 'test.other'
>>> parser_available(test_file)
False
```

```
>>> unload_all_plugins()
```

jsonextended.plugins.**plugins_context**(*classes*, *category=None*)

context manager to load plugin class(es) then unload on exit

Parameters

- **classes** (*list*) – list of classes
- **category** (*None or str*) – if str, apply for single plugin category

Examples

```
>>> from pprint import pprint
```

```
>>> class DecoderPlugin(object):
...     plugin_name = 'example'
...     plugin_descript = 'a decoder for dicts containing _example_ key'
```

(continues on next page)

(continued from previous page)

```
...     dict_signature = ('_example_',)
...
```

```
>>> with plugins_context([DecoderPlugin]):
...     pprint(view_plugins())
{'decoders': {'example': 'a decoder for dicts containing _example_ key'},
 'encoders': {},
 'parsers': {}}
```

```
>>> pprint(view_plugins())
{'decoders': {}, 'encoders': {}, 'parsers': {}}
```

jsonextended.plugins.**unload_all_plugins**(*category=None*)

clear all plugins

Parameters **category** (*None or str*) – if str, apply for single plugin category

Examples

```
>>> from pprint import pprint
>>> pprint(view_plugins())
{'decoders': {}, 'encoders': {}, 'parsers': {}}
```

```
>>> class DecoderPlugin(object):
...     plugin_name = 'example'
...     plugin_descript = 'a decoder for dicts containing _example_ key'
...     dict_signature = ('_example_',)
...
>>> errors = load_plugin_classes([DecoderPlugin])
```

```
>>> pprint(view_plugins())
{'decoders': {'example': 'a decoder for dicts containing _example_ key'},
 'encoders': {},
 'parsers': {}}
```

```
>>> unload_all_plugins()
>>> pprint(view_plugins())
{'decoders': {}, 'encoders': {}, 'parsers': {}}
```

jsonextended.plugins.**unload_plugin**(*name*, *category=None*)

remove single plugin

Parameters

- **name** (*str*) – plugin name
- **category** (*str*) – plugin category

Examples

```
>>> from pprint import pprint
>>> pprint(view_plugins())
{'decoders': {}, 'encoders': {}, 'parsers': {}}
```

```
>>> class DecoderPlugin(object):
...     plugin_name = 'example'
...     plugin_descript = 'a decoder for dicts containing _example_ key'
...     dict_signature = ('_example_',)
...
>>> errors = load_plugin_classes([DecoderPlugin],category='decoders')
```

```
>>> pprint(view_plugins())
{'decoders': {'example': 'a decoder for dicts containing _example_ key'},
 'encoders': {},
 'parsers': {}}
```

```
>>> unload_plugin('example','decoders')
>>> pprint(view_plugins())
{'decoders': {}, 'encoders': {}, 'parsers': {}}
```

jsonextended.plugins.**view_interfaces**(*category=None*)
return a view of the plugin minimal class attribute interface(s)

Parameters **category** (*None or str*) – if str, apply for single plugin category

Examples

```
>>> from pprint import pprint
>>> pprint(view_interfaces())
{'decoders': ['plugin_name', 'plugin_descript', 'dict_signature'],
 'encoders': ['plugin_name', 'plugin_descript', 'objclass'],
 'parsers': ['plugin_name', 'plugin_descript', 'file_regex', 'read_file']}
```

jsonextended.plugins.**view_plugins**(*category=None*)
return a view of the loaded plugin names and descriptions

Parameters **category** (*None or str*) – if str, apply for single plugin category

Examples

```
>>> from pprint import pprint
>>> pprint(view_plugins())
{'decoders': {}, 'encoders': {}, 'parsers': {}}
```

```
>>> class DecoderPlugin(object):
...     plugin_name = 'example'
...     plugin_descript = 'a decoder for dicts containing _example_ key'
...     dict_signature = ('_example_',)
...
>>> errors = load_plugin_classes([DecoderPlugin])
```

```
>>> pprint(view_plugins())
{'decoders': {'example': 'a decoder for dicts containing _example_ key'},
 'encoders': {},
 'parsers': {}}
```

```
>>> view_plugins('decoders')
{'example': 'a decoder for dicts containing _example_ key'}
```

```
>>> unload_all_plugins()
```

jsonextended.utils module

jsonextended.utils.**class_to_str**(*obj*)
get class string from object

Examples

```
>>> class_to_str(list).split('.')[1]
'list'
```

jsonextended.utils.**get_data_path**(*data*, *module*, *check_exists=True*)
return a directory path to data within a module

Parameters **data** (*str or list[str]*) – file name or list of sub-directories and file name (e.g. ['lammps','data.txt'])

jsonextended.utils.**get_module_path**(*module*)
return a directory path to a module

jsonextended.utils.**get_test_path**()
returns test path object

Examples

```
>>> path = get_test_path()
>>> path.name
'_example_data_folder'
```

jsonextended.utils.**load_memit**()
load memory usage ipython magic, require memory_profiler package to be installed

to get usage: %memit?

Author: Vlad Niculae <vlad@vene.ro> Makes use of memory_profiler from Fabian Pedregosa available at https://github.com/fabianp/memory_profiler

jsonextended.utils.**memory_usage**()
return memory usage of python process in MB

from http://fa.bianp.net/blog/2013/different-ways-to-get-memory-consumption-or-lessons-learned-from-memory_profiler/ psutil is quicker

```
>>> isinstance(memory_usage(),float)
True
```

jsonextended.utils.**natural_sort**(*iterable*)
human order sorting of number strings

Examples

```
>>> sorted(['011','1', '21'])
['011', '1', '21']
```

```
>>> natural_sort(['011','1', '21'])
['1', '011', '21']
```

6.1.3 Module contents

a module to extend the python json package functionality;

- decoding/encoding between the on-disk JSON structure and in-memory nested dictionary structure, including:
  - treating path structures, with nested directories and multiple .json files, as a single json.
  - on-disk indexing of the json structure (using the ijson package)
  - extended data type serialisation (numpy.ndarray, Decimals, pint.Quantities,...)
- viewing and manipulating the nested dictionaries:
  - enhanced pretty printer
  - Javascript rendered, expandable tree in the Jupyter Notebook
  - filter, merge, flatten, unflatten functions
- Units schema concept to apply and convert physical units (using the pint package)
- Parser abstract class for dealing with converting other file formats to JSON

Notes

On-disk indexing of the json structure, before reading into memory, to reduce memory overhead when dealing with large json structures/files (using the ijson package), e.g.

> path = get_test_path() %memit jdict1 = to_dict(path,['dir1','file2','meta'],in_memory=True) maximum of 3: 12.242188 MB per loop
>
> %memit jdict1 = to_dict(path,['dir1','file2','meta'],in_memory=False) maximum of 3: 6.996094 MB per loop

Examples

```
>>> from jsonextended import ejson, edict, utils
```

```
>>> path = utils.get_test_path()
>>> path.is_dir()
True
```

```
>>> ejson.jkeys(path)
['dir1', 'dir2', 'dir3']
```

```
>>> jdict1 = ejson.to_dict(path)
>>> edict.pprint(jdict1,depth=2)
dir1:
  dir1_1: {...}
  file1: {...}
  file2: {...}
dir2:
  file1: {...}
dir3:
```

```
>>> jdict2 = ejson.to_dict(path,['dir1','file1'])
>>> edict.pprint(jdict2,depth=1)
initial: {...}
meta: {...}
optimised: {...}
units: {...}
```

```
>>> filtered = edict.filter_keys(jdict2,['vol*'],use_wildcards=True)
>>> edict.pprint(filtered)
initial:
  crystallographic:
    volume: 924.62752781
  primitive:
    volume: 462.313764
optimised:
  crystallographic:
    volume: 1063.98960509
  primitive:
    volume: 531.994803
```

```
>>> edict.pprint(edict.flatten(filtered))
(initial, crystallographic, volume):   924.62752781
(initial, primitive, volume):          462.313764
(optimised, crystallographic, volume): 1063.98960509
(optimised, primitive, volume):        531.994803
```

CHAPTER 7

Releases

7.1 v0.7.0 - Improved edict.filter_keyvals and added edict.filter_keyfuncs

- added "OR"/"AND" *logic* parameter
- removed *errors* parameter from edict.filter_keyvals (instead use edict.filter_keyfuncs)

7.1.1 v0.7.1 - added deep_copy option to edict functions

7.1.2 v0.7.2 - corrected filter_keyvals for dict_like siblings

- added additional tests

7.1.3 v0.7.3 - added plugin class contextmanager

7.1.4 v0.7.4 - added fraction.Fraction encoder plugin

7.1.5 v0.7.6 - bug fix for mock path read context

7.1.6 v0.7.7 - added allow_other_keys for plugin.decoders

7.1.7 0.7.8 - add sdist to pypi (required for conda-forge)

7.1.8 v0.7.9 - add files required by setup.py to manifest (for sdist)

7.1.9 v0.7.10 - Improve testing

- change from nose to pytest

- add testing for flake8 linting on travis
- add testing for document creation on travis

7.2 v0.6.0 - Improvements to LazyLoad

- in *edict.LazyLoad*
  - changed *ignore_prefixes* -> *ignore_regexes* for greater flexibility
  - added logging (at debug level) for each file parsed (helpful for longer loading times)
  - added better exception handling for file parsing (to help with debugging)
- added *edict.dump* in order to better mirror standard *json* module (its exactly the same as *edict.to_json*)

7.2.1 v0.6.1 - added remove_lkey to edict.apply and parse_errors to LazyLoad

7.2.2 v0.6.2 - added .yaml parser plugin

7.2.3 v0.6.3 - edict.remove_paths; allow list of path keys

7.2.4 v0.6.4 - edict.merge added list_of_dicts option

7.3 v0.5.0 - Major Improvements to MockPath

split off into separate package

paths relative to base

index

handle maketemp of folder

7.3.1 v0.5.1 - API Documentation update

7.3.2 v0.5.2 - Minor improvement

7.3.3 v0.5.3 - Minor Bug Fix

7.3.4 v0.5.4 - Minor improvement

- added byte decoding to mock path write class

7.3.5 v0.5.5 - Minor improvements of MockPath

added stat and chmod (dummy) methods

7.3.6 v0.5.6 - Minor improvements of MockPath

7.3.7 v0.5.7 - Reorder Documentation of Versions

7.4 v0.4.0 - Apply functions

- added apply and combine_apply functions
- refactored edict to avoid deepcopy recursion (flatten, unflatten)

7.4.1 v0.4.1 - General functionality improvement

- added more support for list of dict structures
- option to keep siblings when filtering by keyval
- added wildcard option to remove_keys and value plus/minus error to filter_keyvals

7.4.2 v0.4.2 - Added ReadTheDoc Site

7.4.3 v0.4.3 - minor bug fixes and improvements

7.4.4 v0.4.4 - Addition of Diff Evaluator

edict.diff, which can optionally use numpy.allclose to assess arrays of floating point numbers

7.4.5 v0.4.5 - Minor improvements

7.4.6 v0.4.6 - Minor improvements of MockPath

CHAPTER 8

Indices and tables

- genindex
- modindex
- search

Python Module Index

j

Index

G

H

I

J

K

L

M

N

O

P

R

S

T

U

V

Y