diggrtoolbox Documentation

F. Rämisch and P. Mühleder

Apr 15, 2020

Contents:

diggrtoolbox is a collection of various loosely coupled or completely independent tools, which were developed during the first phase of the diggr (databased infrastructure for global game culture research) project at the university library in Leipzig.

The tools are mostly small helpers meant to ease the handling of data and data structures we encountered during this research project.

Note: The main development paradigm for this library was and is: Providing tools, which have few to no additional/external dependencies, especially no requirement for any services to be run in the network, e.g. elasticsearch, CouchDB, etc. It is a toolbox made for Digital Humanities Researchers who do not have access to a huge technical infrastructure.

CHAPTER 1

Getting started

1.1 diggrtoolbox

This collection of tools was developed in the Databased infrastructure for global game culture reasearch (diggr) group at the University Library in Leipzig. Being a collection means, that these helpers are organised into individual packages. Each package is built for one purpose, but the functionality and purpose across functionality may be differ.

For the full documentation have a look at https://diggrtoolbox.readthedocs.io

1.1.1 Requirements

This Software was tested with Python 3.5 and 3.6. There are no further requirements. *diggrtoolboxes* uses only packages and modules which are shipped with Python. Only exception: If you plan development on *diggrtoolbox* you need to have *pytest* to run the tests.

1.1.2 Components

- *deepget*: A small helper easing access to data in deeply nested dicts/list, by separating the definition of the route and actual call.
- *ZipSingleAccess*: Allows access to a JSON document in a ZIP-File.
- *ZipMultiAccess*: Allows access to a JSON document in a ZIP-File, where some parts of the original JSON document are separated into separate json documents. This eases the handling of large files, which otherwise would clog the RAM.
- *TreeExplore*: Class to help exploring deeply nested dicts/lists/both. It provides various helpful display and search functions. It can help exploring raw dumps aquired from APIs on the internet. The search function returns a route-object which can be fed to deepget, in order to retrieve specific datasets.
- *treehash*: Allows comparison of complex data structures by hashing it. It allows to compare deeply nested dicts/lists/both without having to compare its individual components.

1.1.3 Authors

- Florian Rämisch <raemisch@ub.uni-leipzig.de>
- Peter Mühleder <muehleder@ub.saw-leipzig.de>

1.1.4 License

- MIT License.

1.1.5 Copyright

1.2 Installation

It is recommended to use *diggrtoolbox* in a virtualenvironment such as virtualenv. Please refer to the documentation of virtualenv and/or virtualenvwrapper or pipenv to see how to set it up.

The latest version of *diggrtoolbox* can be obtained from github.

1.2.1 Install the latest version

You can install the latest version via pip:

```
pip install git+https://github.com/diggr/diggrtoolbox
```

1.2.2 Development

If you plan to develop *diggrtoolbox* it is recommended to clone the github repository:

```
git clone git@github.com/diggr/diggrtoolbox
```

Installation is performed using pip, but in editable mode, i.e. such that changes in the source take effect immediately:

```
pip install -e ./diggrtoolbox
```

1.3 Examples

To demonstrate possible applications of the tools of the toolbox, this page will contain example use cases.

1.3.1 UnifiedAPI / DiggrAPI

This is the latest addition to the toolbox. It allows the user to have an easier access to the unifiedAPI without having to memorize addresses. You can set filters, select datasets, etc.

The following will create an instance, and select the dataset mobygames.

```
>>> from diggrtoolbox.unified_api import DiggrAPI
>>> d = DiggrAPI("http://localhost:6660).dataset("mobygames")
```

If you now get() this, you will get a list of all ids.

```
>>> ids = d.get()
```

Let's suppose you are interested in links. Apply a filter, and then iterate over all ids, and run your process

```
>>> d.filter("links")
>>> for id_ in ids:
>>>     data = d.item(id_).get()
>>>     # further processing
```

To clean up the code a bit, you can get the result immediately after setting an item id (or slug), by initializing DiggrAPI with *get_on_item=True*. If the "magic" (i.e. filtering the content of the request instead of returning the raw response) does not fit your needs, you can also set *raw=True*.

```
>>> d = DiggrAPI("http://localhost:6660", get_on_item=True, raw=True)
>>> d.dataset("mobygames").filter("links")
>>> raw_data = d.item("id_")
```

1.3.2 ZipSingleAccess

Imagine you have a lot of data stored in one JSON-file. Often these files can be compressed to take a lot less space on your harddrive. When you want to work with the content of these files, of course you don't want to upack them first:

```
>>> import diggrtoolbox as dt
>>> z = ZipSingleAccess("data/compressed_file.zip")
>>> j = z.json()
>>> isinstance(j, dict)
True
>>> print(j.keys())
dict_keys(['id', 'data', 'raw'])
```

1.3.3 ZipMultiAccess

Sometimes the data, you want so load from a file, which is bigger than the RAM you have. This is a problem, as it makes it impossible to work with files of this size without some tricks.

In the natural sciences this problem is tackled by using HDF5, a special file format, allowing to partially load the file, and only serve the parts needed for the next computation step. Unfortunately, this file is not quite made to store tree like structures like nested dicts/lists.

With ZipMultiAccess we make the first step into this direction. You save subtrees of your data in a subfolder, and then load it from the ZIP when you need it:

```
>>> import diggrtoolbox as dt
>>> z = ZipMultiAccess("data/compressed_files.zip")
>>> j = z.json()
>>> isinstance(j, list)
True
>>> len(j)
38386
```

(continues on next page)

(continued from previous page)

```
>>> isinstance(j[0], dict)
True
>>> print(j[0].keys())
dict_keys(['id', 'data', 'raw', 'matches'])
>>> print(j[0]['matches'])
{'n_matches': 3}
>>> m1 = z.get(j[0]['id'])
>>> isinstance(m, list)
True
>>> len(m)
3
```

In the above example we have a list of 38386 which we matched with other games from another database. The match data is huge, so putting all data into one file resulted in a big freeze, as the amount of memory required to hold put all information into one Python object was larger, than the amount the machine had available.

All match data was put into separate files, in a subfolder *matches* and then referenced with the id in the filename. The name of the subfolder can be chosen arbitrarily.

There are multiple ways of accessing the additional files:

```
>>> z[j[0]['id']] == z.get(j[0]['id'])
True
```

1.3.4 TreeExplore

The TreeExplore class provides easy access to nested dicts/list or combinations of both:

```
>>> import diggrtoolbox as dt
>>> test_dict = {'id' : 123456789,
>>>              'data' : {'name': 'diggr project',
>>>                        'city': 'Leipzig',
>>>                        'field': 'Video Game Culture'},
>>>              'references':[{'url': 'http://diggr.link',
>>>                             'name': 'diggr website'},
>>>                             {'url': 'http://ub.uni-leipzig.de',
>>>                              'name': 'UBL website'}]}
>>> tree = dt.TreeExplore(test_dict)
>>> results = tree.search("leipzig")
Search-Term: leipzig
Route: references, 1, url,
Embedding: 'http://ub.uni-leipzig.de'
>>> print(results)
[{'embedding': 'http://ub.uni-leipzig.de',
  'route': ['references', 1, 'url'],
  'unique_in_embedding': False,
  'term': 'leipzig'}]
```

1.3.5 treehash

Imagine you have a datastructure, which you use as a reference at some point in your workflow. It is provided as a JSON-file at some point online, e.g. the diggr platform mapping for the MediaartsDB.

This file is updated frequently. You write a program to check if the contents of the file change, compared with the version you have locally:

```
import requests
import diggrtoolbox as dt

URL = 'https://diggr.github.io/platform_mapping/mediaartdb.json'
```

If the hashes turn out to be different, and you'd like to investigate the differences in more detail, we recommend using a diff-tool like dictdiffer.

1.3.6 deepget

The deepget function can be used easy with the results object of the TreeExplore search function, as demonstrated below:

```
>>> import diggrtoolbox as dt
>>> test_dict = {'id' : 123456789,
                 'data' : {'name' : 'diggr project',
                           'city' : 'Leipzig',
                           'field': 'Video Game Culture'},
                 'references':[{'url' : 'http://diggr.link',
                                'name' : 'diggr website'},
                               {'url' : 'http://ub.uni-leipzig.de',
                                'name' : 'UBL website'}]}
>>> tree = dt.TreeExplore(test_dict)
>>> results = tree.quiet_search("leipzig")
>>> for result in results:
        print(dt.deepget(test_dict, result['route']))
http://ub.uni-leipzig.de
```

The *TreeExplore* class itself also provides an easy method for accessing nested objects. Either a key, index, result dict or route can be used:

```
>>> print(tree[result])
http://ub.uni-leipzig.de
>>> print(tree[result['route']])
http://ub.uni-leipzig.de
>>> print(tree['references'][1]['url'])
http://ub.uni-leipzig.de
```

1.4 diggrtoolbox

1.4.1 diggrtoolbox package

Subpackages

diggrtoolbox.configgr package

Submodules

diggrtoolbox.configgr.configgr module

The Configgr provides a simple and easy to use configuration method.

Author: F. Rämisch <raemisch@ub.uni-leipzig.de> Copyright: Universitätsbibliothek Leipzig, 2018 License: GNU General Public License v3

class diggrtoolbox.configgr.configgr.**Configgr**(*config_filename*, *inspect_locals=True*, *try_lower_on_fail=True*)

Bases: object

Developers define a default configuration for their programs using constants in the source . These constants are inspected, upon instanciation, and saved into the config object. The config file is read, and all settings are imported too. Constants are overwritten in the config, out of course are still usable in the program config.

This results in the fact, that you can set a default behaviour in the source code, let the user configure a setting in a config file, but comment it out upon shipping, to indicate that configuration of this setting is not required.

Module contents

class diggrtoolbox.configgr.**Configgr**(*config_filename*, *inspect_locals=True*, *try_lower_on_fail=True*)

Bases: object

Developers define a default configuration for their programs using constants in the source . These constants are inspected, upon instanciation, and saved into the config object. The config file is read, and all settings are imported too. Constants are overwritten in the config, out of course are still usable in the program config.

This results in the fact, that you can set a default behaviour in the source code, let the user configure a setting in a config file, but comment it out upon shipping, to indicate that configuration of this setting is not required.

diggrtoolbox.deepget package

Submodules

diggrtoolbox.deepget.deepget module

Deepget is a small function enabling the user to "cherrypick" specific values from deeply nested dicts or lists.

Author: Florian Rämisch <raemisch@ub.uni-leipzig.de> Copyright: Universitätsbibliothek Leipzig, 2018 License: GPLv3

diggrtoolbox.deepget.deepget.**deepget**(*obj*, *keys*)

Deepget is a small function enabling the user to "cherrypick" specific values from deeply nested dicts or lists. This is useful, if the just one specific value is needed, which is hidden in multiple hierarchies.

Example

```
>>> import diggrtoolbox as dt
>>> ENTRY = {'data' : {'raw': {'key1': 'value1',
                               'key2': 'value2'}}}
>>> KEY2 = ['data', 'raw', 'key2']
>>> dt.deepget(ENTRY, KEY2) == 'value2'
True
```

Module contents

diggrtoolbox.deepget.**deepget**(*obj*, *keys*)

Deepget is a small function enabling the user to "cherrypick" specific values from deeply nested dicts or lists.

This is useful, if the just one specific value is needed, which is hidden in multiple hierarchies.

Example

```
>>> import diggrtoolbox as dt
>>> ENTRY = {'data' : {'raw': {'key1': 'value1',
                               'key2': 'value2'}}}
>>> KEY2 = ['data', 'raw', 'key2']
>>> dt.deepget(ENTRY, KEY2) == 'value2'
True
```

diggrtoolbox.linking package

Subpackages

diggrtoolbox.linking.resources package

Module contents

Submodules

diggrtoolbox.linking.config module

diggrtoolbox.linking.helpers module

diggrlink helpers module contains helper functions used for dataset linking

diggrtoolbox.linking.helpers.**extract_all_numbers**(*a*)
: returns all numbers (roman and arabic) in string :a:

diggrtoolbox.linking.helpers.**load_excluded_titles**()
: Load list of excudled titles from resource file

diggrtoolbox.linking.helpers.**load_series**()
: Load list of series to remove from title

diggrtoolbox.linking.helpers.**remove_numbers**(*a*)
: removes all numbers (arabic and roman) from string a

diggrtoolbox.linking.helpers.**remove_tm**(*a*)
: Removes trademark symbols from string :a:

diggrtoolbox.linking.helpers.**std**(*a*)
: standardizes string :a: (removes punctuation, blanks, macrons; sets string to lower case)

diggrtoolbox.linking.helpers.**word_before_after**(*a*, *sep*)
: returns word before and after :sep: in string :a:

diggrtoolbox.linking.link module

link module for linking datasets

diggrtoolbox.linking.link.**match_titles**(*titles_a*, *titles_b*, *rules=[<function first_letter_rule>*, *<function numbering_rule>]*)
: Returns match value for two lists of titles.

Titles_a List of title strings

Titles_b List of title string

Rules List of matching rules

diggrtoolbox.linking.rules module

module contains general matching rules

diggrtoolbox.linking.rules.**first_letter_rule**(*a*, *b*)
checks if first letters of strings :a: and :b: when the strings contain max. 1 word

diggrtoolbox.linking.rules.**numbering_rule**(*a*, *b*)
Check two stings for number at the end or inbetween followed by a colon. If a number is found in both strings and if they do not match, return penalty value.

Module contents

diggrtoolbox.platform_mapping package

Submodules

diggrtoolbox.platform_mapping.platform_mapping module

This file provides a class which

class diggrtoolbox.platform_mapping.platform_mapping.**PlatformMapper**(*dataset*, *sep=','*)

Bases: object

Reads in diggr plattform mapping file and provides a mapping dict

std(*source_name*)

diggrtoolbox.platform_mapping.platform_mapping.**get_platform_mapping**(*database*, *with_metadata=False*)
This function gets the platform mapping :param database: name of the video game database the mapping should be obtained for :param with_metadata: if set, a metadata block will be returned additionally, default: False :return: a dict with the mapping, and optionally a dict with the metadata

Module contents

diggrtoolbox.rdfutils package

Submodules

diggrtoolbox.rdfutils.jsonld_loader module

Module contents

diggrtoolbox.schemaload package

Submodules

diggrtoolbox.schemaload.schemaload module

Provides two methods which combine opening files and verification against given schema.

diggrtoolbox.schemaload.schemaload.**load_file_with_schema**(*filename*, *schema*)
Loads data from a file and exits the program if errors occur. If this functionality is not required please use the schema_load function. :param filename: filename of the file with the data :param schema: filename of the file with the schema :return: the data in the datafile as python object (list or dict)

diggrtoolbox.schemaload.schemaload.**schema_load**(*data_filename*, *schema_filename*)
Opens the given file and returns its content as python object, if it contains valid JSON data. Otherwise exceptions are raised, which need to be catched in the calling function :param data_filename: full path to the input file :param schema_filename: full path to the input file :return: dict or list

Module contents

diggrtoolbox.standardize package

Submodules

diggrtoolbox.standardize.standardize module

diggrtoolbox.standardize.standardize.**remove_bracketed_text**(*s*)
Removes text in brackets from string :s: .

diggrtoolbox.standardize.standardize.**remove_html**(*s*)
Removes html tags from string :s: .

diggrtoolbox.standardize.standardize.**remove_punctuation**(*s*)
Removes punctuation from string

diggrtoolbox.standardize.standardize.**std**(*s*, *lower=True*, *rm_punct=True*, *rm_bracket=True*, *rm_spaces=False*, *rm_strings=None*)
Combined string stardardization function. :lower: lower case :rm_punct: remove punctuation :rm_bracket: remove brackets () [] :rm_spaces: remove white spaces :rm_stirng: list of substrings to be removed from string before comparison

diggrtoolbox.standardize.standardize.**std_url**(*url*)
Standardizes urls by removing protocoll and final slash.

Module contents

diggrtoolbox.standardize.**remove_html**(*s*)
Removes html tags from string :s: .

diggrtoolbox.standardize.**remove_bracketed_text**(*s*)
Removes text in brackets from string :s: .

diggrtoolbox.standardize.**remove_punctuation**(*s*)
Removes punctuation from string

diggrtoolbox.standardize.**std_url**(*url*)
Standardizes urls by removing protocoll and final slash.

diggrtoolbox.standardize.**std**(*s*, *lower=True*, *rm_punct=True*, *rm_bracket=True*, *rm_spaces=False*, *rm_strings=None*)
Combined string stardardization function. :lower: lower case :rm_punct: remove punctuation :rm_bracket: remove brackets () [] :rm_spaces: remove white spaces :rm_stirng: list of substrings to be removed from string before comparison

diggrtoolbox.treeexplore package

Submodules

diggrtoolbox.treeexplore.treeexplore module

Getting data structures to work with, sometimes is hard, especially, when you need to find specific information in nested jsons and no schema is provided, or the data and its changing fast.

Author: F. Rämisch <raemisch@ub.uni-leipzig.de> Copyright: 2018, Universitätsbibliothek Leipzig License: GNU General Public License v3

class diggrtoolbox.treeexplore.treeexplore.**TreeExplore**(*tree*, *tab_symbol=' '*)
Bases: object

TreeExplore provides easy to use methods to explore complex data structures obtained e.g. from online REST-APIs. As data structures behind often grew over the years, the internal structure of these objects to be obtained often is not logical.

By providing a full text search and a show method, this tool can be helpful when first investigating, what information is to be found in the data and what is its structure.

Example

```
>>> import diggrtoolbox as dt
>>> test_dict = {'id' : 123456789,
>>>              'data' : {'name' : 'diggr project',
>>>                        'city' : 'Leipzig',
>>>                        'field': 'Video Game Culture'},
>>>              'references':[{'url' : 'http://diggr.link',
>>>                             'name' : 'diggr website'},
>>>                             {'url' : 'http://ub.uni-leipzig.de',
>>>                              'name' : 'UBL website'}]}
>>> tree = dt.TreeExplore(test_dict)
>>> results = tree.search("leipzig")
Search-Term: leipzig
Route: references, 1, url,
Embedding: 'http://ub.uni-leipzig.de'
>>> print(results)
[{'embedding': 'http://ub.uni-leipzig.de',
  'route': ['references', 1, 'url'],
  'unique_in_embedding': False,
  'term': 'leipzig'}]
```

Note: Currently the search is case sensitive only!

find(*term*)

find_key(*key*)

find_value(*value*)

quiet_search(*term*)
Wrapper for the _search function to ease access to a nonprinting search function.

Parameters **term** (*str, int, float*) – the term/object to be found in the tree.

search(*term*)
Wrapper for the _search function, stripping all the parameters not to be used by the end user.

Parameters **term** (*str, int, float*) – the term/object to be found in the tree.

show(*tree=None*, *indent=0*)
Visualizes the whole tree. If no tree-like structure (dict/list/both) is given, the self.tree is used. This function is called recursively with the nested subtrees.

Parameters

- **tree** (*dict, list*) – The tree to be shown.
- **indent** (*int*) – Current indentation level of this tree

show_search_result(*result*)
Displays a search result together with its embedding and path.

Parameters **result** (*dict*) – the result dict generated by _prepare_search_result

diggrtoolbox.treeexplore.treehash module

TreeHash is a Function enabling the user to compare nested dicts and lists by generating a hash.

diggrtoolbox.treeexplore.treehash.**treehash**(*var*)
Returns the hash of any dict or list, by using a string conversion via the json library.

Module contents

class diggrtoolbox.treeexplore.**TreeExplore**(*tree*, *tab_symbol=' '*)
Bases: object

TreeExplore provides easy to use methods to explore complex data structures obtained e.g. from online REST-APIs. As data structures behind often grew over the years, the internal structure of these objects to be obtained often is not logical.

By providing a full text search and a show method, this tool can be helpful when first investigating, what information is to be found in the data and what is its structure.

Example

```
>>> import diggrtoolbox as dt
>>> test_dict = {'id' : 123456789,
>>>              'data' : {'name' : 'diggr project',
>>>                        'city' : 'Leipzig',
```

(continues on next page)

(continued from previous page)

```
>>>                         'field': 'Video Game Culture'},
>>>               'references':[{'url' : 'http://diggr.link',
>>>                              'name' : 'diggr website'},
>>>                              {'url' : 'http://ub.uni-leipzig.de',
>>>                               'name' : 'UBL website'}]}
>>> tree = dt.TreeExplore(test_dict)
>>> results = tree.search("leipzig")
Search-Term: leipzig
Route: references, 1, url,
Embedding: 'http://ub.uni-leipzig.de'
>>> print(results)
[{'embedding': 'http://ub.uni-leipzig.de',
  'route': ['references', 1, 'url'],
  'unique_in_embedding': False,
  'term': 'leipzig'}]
```

Note: Currently the search is case sensitive only!

find(*term*)

find_key(*key*)

find_value(*value*)

quiet_search(*term*)
Wrapper for the _search function to ease access to a nonprinting search function.

Parameters **term** (*str, int, float*) – the term/object to be found in the tree.

search(*term*)
Wrapper for the _search function, stripping all the parameters not to be used by the end user.

Parameters **term** (*str, int, float*) – the term/object to be found in the tree.

show(*tree=None*, *indent=0*)
Visualizes the whole tree. If no tree-like structure (dict/list/both) is given, the self.tree is used. This function is called recursively with the nested subtrees.

Parameters

- **tree** (*dict, list*) – The tree to be shown.
- **indent** (*int*) – Current indentation level of this tree

show_search_result(*result*)
Displays a search result together with its embedding and path.

Parameters **result** (*dict*) – the result dict generated by _prepare_search_result

diggrtoolbox.treeexplore.**treehash**(*var*)
Returns the hash of any dict or list, by using a string conversion via the json library.

diggrtoolbox.unified_api package

Submodules

diggrtoolbox.unified_api.diggr_api module

class diggrtoolbox.unified_api.diggr_api.**DiggrAPI**(*base_url*, *get_on_item=False*, *raw=False*)

Bases: object

This class provides easy access to the diggr unified API. On initialization you have to provide the address of your desired unified API endpoint. You can now set the dataset and filters, which are persistent until reset. This allows you to iterate over a dataset without having to apply a filter each time.

The get() method will do some magic to determine the correct way of creating the directory string depending on the content and dataset selected. I.e. prepend a "/slug", if the identifier is a slug and not an id, or replace slashes in gamefaqs ids.

Example

```
>>> d = DiggrAPI("http://localhost:6660").dataset("mobygames").filter(
↪"companies")
>>> result = d.item("1").get()
```

For the sake of readability you may want to execute the query immediately after the item is set.

```
>>> d = DiggrAPI("http://localhost:6660", get_on_item=True)
>>> d.dataset("mobygames").filter("companies")
>>> results = []
>>> for i in range(10):
>>>     results.append(d.item(i))
```

DATASETS = ('mobygames', 'gamefaqs', 'mediaartdb')

FILTERS = ('companies', 'links', 'cluster')

dataset(*dataset*)
Selects a dataset.

directory
Returns the directory string from self.query. Raises ValueError if no dataset or item is set.

filter(*filterstring*)
Applies a filter. Must be in self.FILTERS.

get()
Runs the query and returns the result.

item(*id_or_slug*)
Selects an item, can be given a numeric id or a slug. Returns self or the result of the query if get_on_item is set.

Module contents

class diggrtoolbox.unified_api.**DiggrAPI**(*base_url*, *get_on_item=False*, *raw=False*)

Bases: object

This class provides easy access to the diggr unified API. On initialization you have to provide the address of your desired unified API endpoint. You can now set the dataset and filters, which are persistent until reset. This allows you to iterate over a dataset without having to apply a filter each time.

The get() method will do some magic to determine the correct way of creating the directory string depending on the content and dataset selected. I.e. prepend a "/slug", if the identifier is a slug and not an id, or replace slashes in gamefaqs ids.

Example

```
>>> d = DiggrAPI("http://localhost:6660").dataset("mobygames").filter(
↪"companies")
>>> result = d.item("1").get()
```

For the sake of readability you may want to execute the query immediately after the item is set.

```
>>> d = DiggrAPI("http://localhost:6660", get_on_item=True)
>>> d.dataset("mobygames").filter("companies")
>>> results = []
>>> for i in range(10):
>>>     results.append(d.item(i))
```

DATASETS = ('mobygames', 'gamefaqs', 'mediaartdb')

FILTERS = ('companies', 'links', 'cluster')

dataset(*dataset*)
Selects a dataset.

directory
Returns the directory string from self.query. Raises ValueError if no dataset or item is set.

filter(*filterstring*)
Applies a filter. Must be in self.FILTERS.

get()
Runs the query and returns the result.

item(*id_or_slug*)
Selects an item, can be given a numeric id or a slug. Returns self or the result of the query if get_on_item is set.

diggrtoolbox.zipaccess package

Submodules

diggrtoolbox.zipaccess.zip_access module

Zip Access is a small tool providing access to zipped json files.

class diggrtoolbox.zipaccess.zip_access.**ZipAccess**(*filename*, *file_ext='.json'*)
Bases: object

Baseclass for the ZipSingleAccess and ZipMultiAccess classes

json(*content_filename=None*)
Opens the zipfile and returns the first zipped JSON file as python object

class diggrtoolbox.zipaccess.zip_access.**ZipListAccess**(*filename*, *file_ext='.json'*)
Bases: *diggrtoolbox.zipaccess.zip_access.ZipAccess*

Class to read a Zipfile.

read_archive()
Reads archive zipfile and returns contents as list of dicts.

class diggrtoolbox.zipaccess.zip_access.**ZipMultiAccess**(*filename*, *file_ext='.json'*)
Bases: *diggrtoolbox.zipaccess.zip_access.ZipAccess*

This class is meant to provide access to a Zip file containing one base json file and a folder with other json files extending the first

ZipMultiAccess provides a __getitem__ method to allow more easy access to the contents.

get(*file_id*)
Returns a specific object, which is not the base object.

Parameters **file_id** (*str*) – Identifier of the object to be returned.

class diggrtoolbox.zipaccess.zip_access.**ZipSingleAccess**(*filename*, *file_ext='.json'*)
Bases: *diggrtoolbox.zipaccess.zip_access.ZipAccess*

This class is meant to provide access to a single JSON-file in a zipfile.

json()
Opens the zipfile and returns the zipped JSON file as python object

Module contents

class diggrtoolbox.zipaccess.**ZipSingleAccess**(*filename*, *file_ext='.json'*)
Bases: *diggrtoolbox.zipaccess.zip_access.ZipAccess*

This class is meant to provide access to a single JSON-file in a zipfile.

json()
Opens the zipfile and returns the zipped JSON file as python object

class diggrtoolbox.zipaccess.**ZipMultiAccess**(*filename*, *file_ext='.json'*)
Bases: *diggrtoolbox.zipaccess.zip_access.ZipAccess*

This class is meant to provide access to a Zip file containing one base json file and a folder with other json files extending the first

ZipMultiAccess provides a __getitem__ method to allow more easy access to the contents.

get(*file_id*)
Returns a specific object, which is not the base object.

Parameters **file_id** (*str*) – Identifier of the object to be returned.

class diggrtoolbox.zipaccess.**ZipListAccess**(*filename*, *file_ext='.json'*)
Bases: *diggrtoolbox.zipaccess.zip_access.ZipAccess*

Class to read a Zipfile.

read_archive()
Reads archive zipfile and returns contents as list of dicts.

Module contents

diggrtoolbox is the main package around all the small tools which were developed in the diggr group. Each tool is located in a separated subpackage.

All tools are made available at package level, as every subpackage often only contains one class/function, separation into the subpackages appeared to be not the best idea.

@author F. Rämisch <raemisch@ub.uni-leipzig.de> @author P. Mühleder <muehleder@ub.uni-leipzig.de> @license https://opensource.org/licenses/MIT MIT License

class diggrtoolbox.**Configgr**(*config_filename*, *inspect_locals=True*, *try_lower_on_fail=True*)

Bases: object

Developers define a default configuration for their programs using constants in the source . These constants are inspected, upon instanciation, and saved into the config object. The config file is read, and all settings are imported too. Constants are overwritten in the config, out of course are still usable in the program config.

This results in the fact, that you can set a default behaviour in the source code, let the user configure a setting in a config file, but comment it out upon shipping, to indicate that configuration of this setting is not required.

diggrtoolbox.**deepget**(*obj*, *keys*)

Deepget is a small function enabling the user to "cherrypick" specific values from deeply nested dicts or lists. This is useful, if the just one specific value is needed, which is hidden in multiple hierarchies.

Example

```
>>> import diggrtoolbox as dt
>>> ENTRY = {'data' : {'raw': {'key1': 'value1',
                               'key2': 'value2'}}}
>>> KEY2 = ['data', 'raw', 'key2']
>>> dt.deepget(ENTRY, KEY2) == 'value2'
True
```

diggrtoolbox.**match_titles**(*titles_a*, *titles_b*, *rules=[<function first_letter_rule>, <function numbering_rule>]*)

Returns match value for two lists of titles.

Titles_a List of title strings

Titles_b List of title string

Rules List of matching rules

class diggrtoolbox.**PlatformMapper**(*dataset*, *sep=', '*)

Bases: object

Reads in diggr plattform mapping file and provides a mapping dict

std(*source_name*)

class diggrtoolbox.**TreeExplore**(*tree*, *tab_symbol=' '*)

Bases: object

TreeExplore provides easy to use methods to explore complex data structures obtained e.g. from online REST-APIs. As data structures behind often grew over the years, the internal structure of these objects to be obtained often is not logical.

By providing a full text search and a show method, this tool can be helpful when first investigating, what information is to be found in the data and what is its structure.

Example

```
>>> import diggrtoolbox as dt
>>> test_dict = {'id' : 123456789,
>>>              'data' : {'name' : 'diggr project',
>>>                        'city' : 'Leipzig',
>>>                        'field': 'Video Game Culture'},
>>>              'references':[{'url' : 'http://diggr.link',
>>>                             'name' : 'diggr website'},
>>>                             {'url' : 'http://ub.uni-leipzig.de',
>>>                              'name' : 'UBL website'}]}
>>> tree = dt.TreeExplore(test_dict)
>>> results = tree.search("leipzig")
Search-Term: leipzig
Route: references, 1, url,
Embedding: 'http://ub.uni-leipzig.de'
>>> print(results)
[{'embedding': 'http://ub.uni-leipzig.de',
  'route': ['references', 1, 'url'],
  'unique_in_embedding': False,
  'term': 'leipzig'}]
```

Note: Currently the search is case sensitive only!

find(*term*)

find_key(*key*)

find_value(*value*)

quiet_search(*term*)
Wrapper for the _search function to ease access to a nonprinting search function.

Parameters **term** (*str, int, float*) – the term/object to be found in the tree.

search(*term*)
Wrapper for the _search function, stripping all the parameters not to be used by the end user.

Parameters **term** (*str, int, float*) – the term/object to be found in the tree.

show(*tree=None*, *indent=0*)
Visualizes the whole tree. If no tree-like structure (dict/list/both) is given, the self.tree is used. This function is called recursively with the nested subtrees.

Parameters

- **tree** (*dict, list*) – The tree to be shown.
- **indent** (*int*) – Current indentation level of this tree

show_search_result(*result*)
Displays a search result together with its embedding and path.

Parameters **result** (*dict*) – the result dict generated by _prepare_search_result

diggrtoolbox.**treehash**(*var*)
Returns the hash of any dict or list, by using a string conversion via the json library.

class diggrtoolbox.**ZipSingleAccess**(*filename*, *file_ext='.json'*)
Bases: *diggrtoolbox.zipaccess.zip_access.ZipAccess*

This class is meant to provide access to a single JSON-file in a zipfile.

json()
Opens the zipfile and returns the zipped JSON file as python object

class diggrtoolbox.**ZipMultiAccess**(*filename*, *file_ext='.json'*)
Bases: *diggrtoolbox.zipaccess.zip_access.ZipAccess*

This class is meant to provide access to a Zip file containing one base json file and a folder with other json files extending the first

ZipMultiAccess provides a __getitem__ method to allow more easy access to the contents.

get(*file_id*)
Returns a specific object, which is not the base object.

Parameters **file_id** (*str*) – Identifier of the object to be returned.

class diggrtoolbox.**ZipListAccess**(*filename*, *file_ext='.json'*)
Bases: *diggrtoolbox.zipaccess.zip_access.ZipAccess*

Class to read a Zipfile.

read_archive()
Reads archive zipfile and returns contents as list of dicts.

- genindex
- search

CHAPTER 2

Authors, Copyright, License

Python Module Index

d

Index

G

I

J

L

M

N

P

Q

R

S

T

W

Z