datesy

Release 0.9.0

Aug 28, 2020

Contents

Datesy, making DATa handling EaSY – the intro to data handling in python

CHAPTER 1

Introduction

Making intro to data handling in python nice and easy!

The `datesy` package provides the conversion between the file types as well as basic data inspection functions.

It is designed for an easy start into the python world and data handling in it without having to think of unnecessary basics.

1.1 Main Usage

Datesy, making DATa handling EaSY, is mostly helpful if you are looking for:

1. inspecting complex data like searching for a path in a dictionary
2. mapping strings and their properties

1.2 Motivation

1.2.1 History

The idea to this package came during the work as a consultant with a customer where lot's of files needed to be read, transformed, inspected etc. and no adequate tools besides searching & filtering with Excel of files partly in the range of GBs were around.

With starting to share the python insights and the code fragments, the only logical next step was do create a really reusable code fragment - a python package.

1.2.2 Future Development

The package is designed to be an easy start into data handling with python. Therefore, its desire is to take care of standard tasks the programmer just does not want to think of but can concentrate on the actual tasks, the data handling.

Whenever there is a task that is done too often in data handling and inspection which can be taken care of in a standardized way, this package will happily be expanded for enabling you to simplify your job.

If needed/desired, further datafile-formats will be supported for having a nice and standardized way of loading/writing those as well.

1.3 Limitations

This package is designed to be used by anybody who is new to python. Therefore functions are explicitly held limited to their magic and described accordingly. There are few things really the big shit rather than simply helping with small tasks which you could have written yourself in a few lines of code but didn't want to think about. For deep data analysis other packages are far more powerful and maybe helpful to you. Think of `datesy` more of the little butler taking care of some basic tasks for you.

This package is compatible to PyPy's version 3.6.

CHAPTER 2

The datesy package

2.1 datesy.convert module

All actions of transforming data from different file formats are to be found here

datesy.convert.**rows_to_dict**(*rows*, *main_key_position=0*, *null_value='delete'*, *header_line=0*, *contains_open_ends=False*)

Convert a row of rows (e.g. csv) to dictionary

Parameters

- **rows** (*list*) – the row based data to convert to *dict*
- **main_key_position** (*int, optional*) – if the main_key is not on the top left, its position can be specified
- **null_value** (*any, optional*) – if an emtpy field in the lists shall be represented somehow in the dictionary
- **header_line** (*int, optional*) – if the header_line is not the first one, its position can be specified
- **contains_open_ends** (*bool, optional*) – if each row is not in the same length (due to last set entry as last element in row), a length check for corrupted data can be ignored

Returns dictionary containing the information from row-based data

Return type dict

datesy.convert.**dict_to_rows**(*data*, *main_key_name=None*, *main_key_position=None*, *if_empty_value=None*, *order=None*)

Convert a dictionary to rows (list(lists))

Parameters

- **data** (*dict*) – the data to convert in form of a dictionary
- **main_key_name** (*str, optional*) – if the data isn't provided as *{main_key: data}* the key needs to be specified

- **main_key_position** (*int, optional*) – if the main_key shall not be on top left of the data the position can be specified
- **if_empty_value** (*any, optional*) – if a main_key's sub_key is not set something different than *blank* can be defined
- **order** (*dict, list, None, optional*) – if a special order for the keys is required

Returns list of rows representing the csv based on the *main_element_position*

Return type list(lists)

datesy.convert.**pandas_data_frame_to_dict**(*data_frame*, *main_key_position=0*, *null_value='delete'*, *header_line=0*)

Converts a single file_name from xlsx to json

Parameters

- **data_frame** (*pandas.core.frame.DataFrame*) –
- **main_key_position** (*int, optional*) –
- **null_value** (*any, optional*) –
- **header_line** (*int, optional*) –

Returns the dictionary representing the xlsx based on *main_key_position*

Return type dict

datesy.convert.**dict_to_pandas_data_frame**(*data*, *main_key_name=None*, *order=None*, *inverse=False*)

Convert a dictionary to pandas.DataFrame

Parameters

- **data** (*dict*) – dictionary of handling
- **main_key_name** (*str, optional*) – if the json or dict does not have the main key as a single *{main_element : dict}* present, it needs to be specified
- **order** (*dict, list, optional*) – list with the column names in order or dict with specified key positions
- **inverse** (*bool, optional*) – if columns and rows shall be switched

Returns DataFrame representing the dictionary

Return type pandas.DataFrame

datesy.convert.**xml_to_standard_dict**(*ordered_data*, *reduce_orderedDicts=False*, *reduce_lists=False*, *manual_selection_for_list_reduction=False*)

Convert a xml/orderedDict to normal dictionary

Parameters

- **ordered_data** (*orderedDict*) – input xml data to convert to standard dict
- **reduce_orderedDicts** (*bool, optional*) – if collections.orderedDicts shall be converted to normal dicts
- **reduce_lists** (*bool, list, set, optional*) – if lists in the dictionary shall be converted to dictionaries with transformed keys (list_key + unique key from dictionary from list_element) if list or set is provided, only these values will be reduced

- **manual_selection_for_list_reduction** (*bool, optional*) – if manually decision on list reduction shall be used all keys in `reduce_lists` will be automatically reduced

Returns the normalized dictionary

Return type dict

2.2 datesy.inspect module

All actions of inspecting data are to be found here

`datesy.inspect.`**find_header_line**(*data*, *header_keys*)

Find the header line in row_based data_structure NOT IMPLEMENTED YET: Version 0.9 feature

Parameters

- **data** (*list, pandas.DataFrame*) –
- **header_keys** (*str, list, set*) – some key(s) to find in a row

Returns the header_line

Return type int

`datesy.inspect.`**find_key**(*data*, *key=None*, *regex_pattern=None*)

Find a key in a complex dictionary

Parameters

- **data** (*dict*) – the data structure to find the key
- **key** (*str, optional*) – a string to be found
- **regex_pattern** (*str, optional*) – a regex match to be found

Returns all matches and their path in the structure `{found_key: path_to_key}`

Return type dict

2.3 datesy.matching module

All actions of mapping data to other data as well as the functions helpful for that are to be found here

`datesy.matching.`**simplify_strings**(*to_simplify*, *lower_case=True*, *simplifier=True*)

Simplify a *string*, *set(strings)*, *list(strings)*, *keys in dict* Options for simplifying include: lower capitals, separators, both (standard), own set of simplifier

Parameters

- **to_simplify** (*list, set, string*) – the string(s) to simplify presented by itself or as part of another data format
- **lower_case** (*bool, optional*) – if the input shall be converted to only lower_case (standard: *True*)
- **simplifier** (*str, optional*) – the chars to be removed from the string. if type bool and True, standard chars `_ , | \n ' & " % * - \` used

Returns simplified values `{simplified_value: input_value}`

Return type dict

datesy.matching.**ease_match_similar**(*list_for_matching*, *list_to_be_matched_to*, *simplified=False*, *similarity_limit_for_matching=0.6*, *print_auto_matched=False*)

Return a dictionary with `list_for_matching` as keys and `list_to_be_matched_to` as values based on most similarity. Matching twice to the same value is possible! Similarity distance for stopping the matching is set by *distance_for_automatic_vs_manual_matching*. Faster than *datesy.matching.match_comprehensive* but when having very similar strings more likely to contain errors.

Parameters

- **list_for_matching** (*list, set*) – Iterable of strings which shall be matched
- **list_to_be_matched_to** (*list, set*) – Iterable of stings which shall be matched to
- **simplified** (*False, "capital", "separators", "all", list, str, optional*) – For reducing the values by all small letters or unifying & deleting separators *separators* or any other list of strings provided
- **print_auto_matched** (*bool, optional*) – Printing the matched entries during process (most likely for debugging)
- **similarity_limit_for_matching** (*float, optional*) – For not matching the most irrelevant match which could exist

Returns

- **match** (*dict*) – *{value_for_matching: value_to_be_mapped_to}*
- **no_match** (*set*) – A set of all values from *list_for_matching* that could not be matched

datesy.matching.**match_comprehensive**(*list_for_matching*, *list_to_be_matched_to*, *simplified=False*)

Return a dictionary with `list_for_matching` as keys and `list_to_be_matched_to` as values based on most similarity. All values of both iterables get compared to each other and highest similarities are picked. Slower than *datesy.matching.ease_match_similar* but more precise.

Parameters

- **list_for_matching** (*list, set*) – Iterable of strings which shall be matched
- **list_to_be_matched_to** (*list, set*) – Iterable of stings which shall be matched to
- **simplified** (*False, "capital", "separators", "all", list, str, optional*) – For reducing the values by all small letters or unifying & deleting separators *separators* or any other list of strings provided

Returns

- **match** (*dict*) – *{value_for_matching: value_to_be_mapped_to}*
- **no_match** (*set*) – A set of all values from *list_for_matching* that could not be matched

datesy.matching.**match_similar_with_manual_selection**(*list_for_matching*, *list_to_be_matched_to*, *simplified=False*, *minimal_distance_for_automatic_matching=0.1*, *print_auto_matched=False*, *similarity_limit_for_manual_checking=0.6*)

Return a dictionary with `list_for_matching` as keys and `list_to_be_matched_to` as values based on most similarity. All possible matches not matched automatically (set limit with *mini-*

mal_distance_for_automatic_matching) can be handled interactively. Similarity distance for stopping the matching is set by *distance_for_automatic_vs_manual_matching*.

Parameters

- **list_for_matching** (*list, set*) – Iterable of strings which shall be matched
- **list_to_be_matched_to** (*list, set*) – Iterable of stings which shall be matched to
- **simplified** (*False, "capital", "separators", "all", list, str, optional*) – For reducing the values by all small letters or unifying & deleting separators *separators* or any other list of strings provided
- **print_auto_matched** (*bool, optional*) – Printing the matched entries during process (most likely for debugging)
- **minimal_distance_for_automatic_matching** (*float, optional*) – If there is a vast difference between the most and second most matching value, automatically matching is provided This parameter provides the similarity distance to be reached for automatically matching
- **similarity_limit_for_manual_checking** (*float, optional*) – For not showing/matching the most irrelevant match which could exist

Returns

- **match** (*dict*) – *{value_for_matching: value_to_be_mapped_to}*
- **no_match** (*set*) – A set of all values from *list_for_matching* that could not be matched

2.4 datesy.sort module

datesy.sort.**create_sorted_list_from_order**(*order*, *all_elements=None*, *main_element=None*, *main_element_position=None*)

Create a sorted list based on the values in order based on the key values.

The function additionally allows to specify more elements for the sorted_list which don't matter in terms of order. Additionally, a main_element can be specified which has a leading position/is specified asides from order.

Parameters

- **order** (*dict, list*) – the dictionary with the positions (keys) and elements (values)
- **all_elements** (*list, set*) – all the strings which shall be put in order. if more keys in all_elements than in order: keys will be added in random order if less keys in all_elements than in order: only the keys in all_elements will be returned, additional ones get deleted
- **main_element** (*str*) – the main_element
- **main_element_position** (*int*) – the position of the main_element

Returns the sorted list with elements from all_elements and main_element

Return type list

CHAPTER 3

Examples

3.1 Installation/Usage

For installation run *pip3 install datesy* in terminal.

For using in Python3 script, import it at the beginning of the script:

```
import datesy

# your code
pass
```

3.2 Inspecting data

Check here all the examples for inspecting data (coming soon)

3.3 Converting data

Check here all the examples for converting data

datesy helps you to easily convert certain types of data. Typical data formats are row-based or in form of a dictionary.

3.3.1 Rows to dictionary

When e.g. reading a csv_file as stated above, a row-based data structure is returned. If for further processing the rows shall be dictionized, it's as simple as this:

```
example_rows = [
                ["Header1", "Header2", "Header3"],
                ["Value11", "Value12", "Value13"],
                ["Value21", "Value22", "Value23"]
               ]

example_rows = datesy.rows_to_dict(rows=example_dict)


example_dict = {
                  "Header1": {
                    "Value11": {
                      "Header2": "Value12",
                      "Header3": "Value13",
                    },
                    "Value21": {
                      "Header2": "Value22",
                      "Header3": "Value23"
                    }
                  }
                }
```

Relevant ID position / main key position:

It might occur, your most relevant key is not on the first position:

```
example_rows = [
                ["Header1", "Header2", "Header3"],
                ["Value11", "Value12", "Value13"],
                ["Value21", "Value22", "Value23"]
               ]
example_dict = datesy.rows_to_dict(rows=example_rows, main_key_position=2)

example_dict = {
                  "Header3": {
                    "Value13": {
                      "Header1": "Value11",
                      "Header2": "Value12"
                    },
                    "Value23": {
                      "Header1": "Value21",
                      "Header2": "Value22"
                    }
                  }
                }
```

As you can see, the third entry (*int=2*) is used as the main_key.

Missing values

Of course, data may be missing a value:

```
example_rows = [
                ["Header1", "Header2", "Header3"],
                ["Value11",, "Value13"],
```

(continues on next page)

(continued from previous page)

```
                ["Value21", "Value22", "Value23"]
               ]
example_dict = datesy.rows_to_dict(rows=example_rows, null_value="delete")

example_dict = {
                 "Header1": {
                   "Value11": {
                     "Header3": "Value13"
                   },
                   "Value21": {
                     "Header2": "Value22",
                     "Header3": "Value23"
                   }
                 }
               }
```

As you can see, the emtpy value in the rows is not represented in the dictionary. Instead of missing the header_key you can also put any other value than `delete` to this parameter for putting this to the exact spot:

```
example_rows = [
                ["Header1", "Header2", "Header3"],
                ["Value11",, "Value13"],
                ["Value21", "Value22", "Value23"]
               ]

example_dict = datesy.rows_to_dict(rows=example_rows, null_value=None)

example_dict = {
                 "Header1": {
                   "Value11": {
                     "Header2": None,
                     "Header3": "Value13"
                   },
                   "Value21": {
                     "Header2": "Value22",
                     "Header3": "Value23"
                   }
                 }
               }
```

Open ends / missing last row entries

If the rows do not contain emtpy values at the end of the row:

Normally, a check prevents handling this data as row-based data should always have the same length. Yet, if emtpy values at the end of the row are not stored like this, you can disable this check and still convert data:

```
example_rows = [
                ["Header1", "Header2", "Header3"],
                ["Value11", "Value12"],
                ["Value21", "Value22", "Value23"]
               ]

example_dict = datesy.rows_to_dict(rows=example_rows, contains_open_ends=True)
```

(continues on next page)

(continued from previous page)

```
example_dict = {
                  "Header1": {
                    "Value11": {
                      "Header2": "Value12"
                    },
                    "Value21": {
                      "Header2": "Value22",
                      "Header3": "Value23"
                    }
                  }
                }
```

Selecting the header_line

For irrelevant data at the top of the row-based data, you can set the header_line to the desired position:

```
example_rows = [
                 ["Undesired1", "Undesired2", "Undesired3"],
                 ["Header1", "Header2", "Header3"],
                 ["Value11", "Value12", "Value13"],
                 ["Value21", "Value22", "Value23"]
                ]

example_dict = datesy.rows_to_dict(rows=example_rows, header_line=1)

example_dict = {
                  "Header1": {
                    "Value11": {
                      "Header2": "Value12",
                      "Header3": "Value13"
                    },
                    "Value21": {
                      "Header2": "Value22",
                      "Header3": "Value23"
                    }
                  }
                }
```

3.3.2 Dictionary to rows

Just as simple is the converting vice_versa from dictionary to rows:

```
example_dict = {
               "Header1": {
                 "Value11": {
                   "Header2": "Value12",
                   "Header3": "Value13",
                 },
                 "Value21": {
                   "Header2": "Value22",
                   "Header3": "Value23"
                 }
               }
```

(continues on next page)

(continued from previous page)

```
            }

example_rows = datesy.dict_to_rows(data=example_dict)

example_rows = [
                 ["Header1", "Header2", "Header3"],
                 ["Value11", "Value12", "Value13"],
                 ["Value21", "Value22", "Value23"]
                ]
```

Missing keys / not set data

When having data where certain keys are not set:

```
example_dict = {
                  "Header1": {
                    "Value11": {
                      "Header2": "Value12"
                    },
                    "Value21": {
                      "Header2": "Value22",
                      "Header3": "Value23"
                    }
                  }
                }

example_rows = datesy.dict_to_rows(data=example_dict)

example_rows = [
                 ["Header1", "Header2", "Header3"],
                 ["Value11", "Value12", ],
                 ["Value21", "Value22", "Value23"]
                ]
```

Specify emtpy values:

Of course you can specify values to be set if a key is not set/emtpy:

```
example_dict = {
                  "Header1": {
                    "Value11": {
                      "Header2": "Value12"
                    },
                    "Value21": {
                      "Header2": "Value22",
                      "Header3": "Value23"
                    }
                  }
                }

example_rows = datesy.dict_to_rows(data=example_dict, if_emtpy_value=False)

example_rows = [
```

(continues on next page)

(continued from previous page)

```
                ["Header1", "Header2", "Header3"],
                ["Value11", "Value12", False],
                ["Value21", "Value22", "Value23"]
               ]
```

Ordering the header

Just like picking the most relevant key in *rows_to_dict*, you can specify a certain order for the row-based data:

```
example_dict = {
                 "Header1": {
                   "Value11": {
                     "Header2": "Value12",
                     "Header3": "Value13"
                   },
                   "Value21": {
                     "Header2": "Value22",
                     "Header3": "Value23"
                   }
                 }
               }

example_rows = datesy.dict_to_rows(data=example_dict, order=["Header2", "Header3",
↪"Header1"])

example_rows = [
                ["Header2", "Header3", "Header1"],
                ["Value12", "Value13", "Value11"],
                ["Value22", "Value23", "Value21"]
               ]
```

Data without main_key

What happens if you have data without a main_key like *Header1* specified? Simply tell *datesy* about it:

```
example_dict = {
                 "Value11": {
                   "Header2": "Value12",
                   "Header3": "Value13",
                 },
                 "Value21": {
                   "Header2": "Value22",
                   "Header3": "Value23"
                 }
               }


example_rows = datesy.dict_to_rows(data=example_dict, main_key_name="Header1")

example_rows = [
                ["Header1", "Header2", "Header3"],
                ["Value11", "Value12", "Value13"],
                ["Value21", "Value22", "Value23"]
               ]
```

3.4 Matching data

Check here all the examples for converting data (coming soon)

3.5 Sorting data

Check here all the examples for converting data (coming soon)

CHAPTER 4

Release Notes

4.1 0.9.0

- **separate package into**
  - datesy (actual data handling)
  - fil_io (file loading/writing/selection)
  - querious (SQL query helper)
  - pythomy (pythonic MySQL interaction)

4.2 0.8.1

4.2.1 Bug Fixes

- sql_query: possible to use strings as values when using `...where(column=value)`

4.3 0.8.0

database connection: connect to a database and interact with it in a pythonic way

4.3.1 Features

- database abstraction available
  - database
  - table

    - row
    - item
- database interaction now possible for:
    - mysql

4.4 0.7.1

saving jsons: beautified to human readability & sorting keys available

4.5 0.7.0

initial release

4.5.1 Features

- reading/writing file types
    - csv
    - json
    - xml
    - xls(x)
- converting data types
    - rows -> dict
    - dict -> rows
    - dict -> DataFrame
    - DataFrame -> dict
- matching strings
    - simplifying strings
    - fast/considerate matching
    - matching with manual selection

4.5.2 Bug Fixes

- initial release

CHAPTER 5

Indices and tables

- genindex
- modindex
- search

Python Module Index

d

Index