datesy

Release 0.7.1

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 easy handling (read/write) of standard file types, the conversion between the file types as well as basic data inspection functionalities.

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. loading/dumping data to a standard file format like *json*, *csv*, *xml*, *xls(x)*
2. inspecting complex data like searching for a path in a dictionary
3. 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

The *datesy* package is divided in 5 main components:

1. file I/O (subpackage)
2. dabase I/O (subpackage)
3. data converting
4. data inspecting
5. matching data sets

2.1 Subpackages

All actions of interacting with files are to be found here:

2.1.1 File I/O subpackage

All functions of *datesy* taking care of file I/O are listed here, separated by file_type. Only exception is the first module, providing functions for file selection.

file_selection module

The file_selection module provides multiple supporting functions for interaction with files

datesy.file_IO.file_selection.**get_latest_file_from_directory**(*directory*, *file_ending=None*, *pattern=None*, *regex=None*)

Return the latest file_name (optionally filtered) from a directory

Parameters

- **directory** (*str*) – the directory where to get the latest file_name from
- **file_ending** (*str, set, optional*) – the file_name ending specifying the file_name type
- **pattern** (*str, optional*) – pattern for the file_name to match `DataFile_*.json` where `*` could be a date or other strings
- **regex** (*str, optional*) – a regular_expression (regex) for pattern matching

Returns the *file_name* with the latest change date

Return type str

datesy.file_IO.file_selection.**get_file_list_from_directory**(*directory*, *file_ending=None*, *pattern=None*, *regex=None*)

Return all files (optionally filtered) from directory in a list

Parameters

- **directory** (*str*) – the directory containing the desired files
- **file_ending** (*str, set, optional*) – the file_name's ending specifying the file type
- **pattern** (*str, optional*) – pattern for the file_names to match `DataFile_*.json` where `*` could be a date or other strings
- **regex** (*str, optional*) – a regular_expression (regex) for pattern matching

Returns a list of all relative file_name directories

Return type list

datesy.file_IO.file_selection.**return_file_list_if_path**(*path*, *file_ending=None*, *pattern=None*, *regex=None*, *return_always_list=False*)

Return all files in directory (optionally specified with options) if path is a directory

Parameters

- **path** (*str*) – the path to test if directory
- **file_ending** (*str, set, optional*) – the file_name ending specifying the file_name type for the files in the directory
- **pattern** (*str, optional*) – pattern for the file_names in directory to match `DataFile_*.json` where `*` could be a date or other strings
- **regex** (*str, optional*) – a regular_expression (regex) for pattern matching of the file_names
- **return_always_list** (*bool, optional*) – if a single path shall be returned as in a list

Returns if directory the list of files else the path (in a list if *return_always_list* is set)

Return type list, str

datesy.file_IO.file_selection.**check_file_name_ending**(*file_name*, *ending*)

Check if the file_name has the expected file_ending

If one of the provided endings is the file_name's ending return *True*, else *False*

Parameters

- **file_name** (*str*) – The file_name to check the ending for The file_name may contain a path, so `file_name.ending` as well as `path/to/file_name.ending` will work
- **ending** (*str, set, list*) – The desired ending or multiple desired endings For single entries e.g. `.json` or `csv`, for multiple endings e.g. `['.json', 'csv']`

Returns *True* if the *file_name*'s ending is in the given *ending*, else *False*

Return type bool

json_file module

The json_file module takes care of all I/O interactions concerning json files

`datesy.file_IO.json_file.`**load**(*path*)
Load(s) json file(s) and returns the dictionary/-ies Specifying a file_name: one file will be loaded. Specifying a directory: all **.json* files will be loaded.

Parameters **path** (*str*) – path to a file_name or directory

Returns dictionary representing the json `{file_name: {data}}`

Return type dict

`datesy.file_IO.json_file.`**load_single**(*file_name*)
Load a single json file

Parameters **file_name** (*str*) – file_name to load from

Returns the loaded json as a dict `{data}`

Return type dict

`datesy.file_IO.json_file.`**load_these**(*file_name_list*)
Load specified json files and return the data in a dictionary with file_name as key

Parameters **file_name_list** (*list*) – list of file_names to load from

Returns the dictionaries from the files as values of file_name as key `{file_name: {data}}`

Return type dict(dict)

`datesy.file_IO.json_file.`**load_all**(*directory*)
Load all json files in the directory and return the data in a dictionary with file_name as key

Parameters **directory** (*str*) – the directory containing the json files

Returns the dictionaries from the files as values of file_name as key `{file_name: {data}}`

Return type dict(dict)

`datesy.file_IO.json_file.`**write**(*file_name*, *data*, *beautify=True*, *sort=False*)
Save json from dict to file

Parameters

- **file_name** (*str*) – the file_name to save under. if no ending is provided, saved as .json
- **data** (*dict*) – the dictionary to be saved as json
- **beautify** (*bool, optional*) – if the data is represented in single row or human readable presented (default: human readable)
- **sort** (*bool, optional*) – if the keys shall be ordered (default: false)

csv_file module

The csv_file module takes care of all I/O interactions concerning csv files

datesy.file_IO.csv_file.**load**(*path*, ***kwargs*)

Load(s) csv file(s) and returns the rows Specifying a file_name: one file will be loaded. Specifying a directory: all **.csv* files will be loaded.

Parameters

- **path** (*str*) – path to a file_name or directory
- **kwargs** (*optional*) – csv dialect options

Returns list of lists if a single file_name was provided: `[[row1.1, row1.2]]` dict of list of lists if multiple files provided: `{file_name : [[row1.1, row1.2]]}`

Return type list, dict

datesy.file_IO.csv_file.**load_single**(*file_name*, ***kwargs*)

Load a csv file and return the rows

Parameters

- **file_name** (*str*) – file_name to load from
- **kwargs** (*optional*) – csv dialect options

Returns list of lists representing the csv data `[[row1.1, row1.2]]`

Return type list

datesy.file_IO.csv_file.**load_these**(*file_name_list*, ***kwargs*)

Load specified csv files and return the rows in a dictionary with file_name as key

Parameters

- **file_name_list** (*list*) – list of file_names to load from
- **kwargs** (*optional*) – csv dialect options

Returns the rows from the files as values of file_name as key `{file_name : [[row1.1, row1.2]]}`

Return type dict

datesy.file_IO.csv_file.**load_all**(*directory*, ***kwargs*)

Load all csv files in the directory and return the rows in a dictionary with file_name as key

Parameters

- **directory** (*str*) – the directory containing the csv files
- **kwargs** (*optional*) – csv dialect options

Returns the rows from the files as values of file_name as key `{file_name : [[row1.1, row1.2]]}`

Return type dict

datesy.file_IO.csv_file.**write**(*file_name*, *data*, *main_key_name=None*, *main_key_position=0*, *order=None*, *if_empty_value=None*, ***kwargs*)

Save a row based document from dict or list to file If presented a dictionary, converting to rows is done by the dict_to_rows method of this package.

Parameters

- **file_name** (*str*) – the file_name to save under. if no ending is provided, saved as *file_name.csv*
- **data** (*dict, list*) – the dictionary or list to be saved as csv
- **main_key_name** (*str, optional*) – if the json or dict does not have the main key as a single key present (`{main_element_name: dict}`), it needs to be specified
- **main_key_position** (*int, optional*) – the position in csv of the dictionary main key
- **order** (*dict, list, optional*) – for defining a specific order of the keys. if dict, format: `{int: str}` either a dictionary with the specified positions in a dictionary with positions as keys (integers) or in a list
- **if_empty_value** (*any, optional*) – the value to set when no handling is available default is "delete" leading to be an empty value
- **kwargs** (*optional*) – csv dialect options

datesy.file_IO.csv_file.**write_from_rows**(*file_name*, *rows*, ***kwargs*)

Save row based document from rows to file

Parameters

- **file_name** (*str*) – the file_name to save the data under. if no ending is provided, saved as *file_name.csv*
- **rows** (*list*) – list of lists to write to file_name
- **kwargs** (*optional*) – csv dialect options

datesy.file_IO.csv_file.**write_from_dict**(*file_name*, *data*, *main_key_name=None*, *main_key_position=0*, *order=None*, *if_empty_value=None*, ***kwargs*)

Save a row based document from dict to file

Parameters

- **file_name** (*str*) – the file_name to save under. if no ending is provided, saved as *file_name.csv*
- **data** (*dict*) – the dictionary to be saved as csv
- **main_key_name** (*str, optional*) – if the json or dict does not have the main key as a single key present (`{main_element_name: dict}`), it needs to be specified
- **order** (*dict {int: str}, list, optional*) – for defining a specific order of the keys either a dictionary with the specified positions in a dictionary with positions as keys (integers) or in a list
- **if_empty_value** (*any, optional*) – the value to set when no handling is available default is "delete" leading to be an empty value
- **main_key_position** (*int, optional*) – the position in csv of the dictionary main key
- **kwargs** (*optional*) – csv dialect options

xls_file module

The xls_file module takes care of all I/O interactions concerning xls(x) files

datesy.file_IO.xls_file.**load_single_sheet**(*file_name*, *sheet=None*)
Load a xls(x) file's (first) sheet to a pandas.DataFrame

Parameters

- **file_name** (*str*) – file_name to load from
- **sheet** (*str, optional*) – a specified sheet_name to extract. default is first sheet

Returns pandas.DataFrame representing the xls(x) file

Return type pandas.DataFrame

datesy.file_IO.xls_file.**load_these_sheets**(*file_name*, *sheets*)
Load from a xls(x) file_name the specified sheets to a pandas.DataFrame as values to sheet_names as keys in a dictionary

Parameters

- **file_name** (*str*) – file_name to load from
- **sheets** (*list*) – sheet_names to load

Returns dictionary containing the sheet_names as keys and pandas.DataFrame representing the xls(x) sheets `{sheet_name: pandas.DataFrame}`

Return type dict(pandas.DataFrame)

datesy.file_IO.xls_file.**load_all_sheets**(*file_name*)
Load from a xls(x) file all its sheets to a pandas.DataFrame as values to sheet_names as keys in a dictionary

Parameters **file_name** (*str*) – file_name to load from

Returns dictionary containing the sheet_names as keys and pandas.DataFrame representing the xls(x) sheets `{sheet_name: pandas.DataFrame}`

Return type dict

datesy.file_IO.xls_file.**load_these_files**(*file_name_list*)
Load the specified xls(x) files with all their sheets to a pandas.DataFrame as values to sheet_names as keys in a dictionary

Parameters **file_name_list** (*list*) – list of file_names to load from

Returns the data from the sheets in a dictionary with sheet_name as key within again a dictionary with file_name as key `{file_name: {sheet_name: pandas.DataFrame}}`

Return type dict

datesy.file_IO.xls_file.**load_all_files**(*directory*)
Load all xls(x) files in the directory with all their sheets to a pandas.DataFrame as values to sheet_names as keys in a dictionary

Parameters **directory** (*str*) – the directory containing the xlsx files

Returns the data from the sheets in a dictionary with sheet_name as key within again a dictionary with file_name as key `{file_name: {sheet_name: pandas.DataFrame}}`

Return type dict

datesy.file_IO.xls_file.**load**(*path*)
Load all xls(x) files in the directory with all their sheets to a pandas.DataFrame as values to sheet_names as keys in a dictionary Specifying a file_name: one file will be loaded. Specifying a directory: all **.xls(x)* files will be loaded.

Parameters **path** (*str*) – path to a file_name or directory

Returns dictionary containing the sheets as *panda.DataFrames*: `{file_name: {sheet_name: pandas.DataFrame}}`

Return type dict

`datesy.file_IO.xls_file.`**`write_single_sheet_from_DataFrame`**(*file_name*, *data_frame*, *sheet_name=None*, *auto_size_cells=True*)

Save a pandas.DataFrame to file

Parameters

- **`file_name`** (*`str`*) – the file_name to save under. if no ending is provided, saved as .xlsx
- **`data_frame`** (*`pandas.DataFrame`*) – pandas.DataFrame to write to file_name
- **`sheet_name`** (*`str, optional`*) – a sheet_name containing the data
- **`auto_size_cells`** (*`bool, optional`*) – if the auto-sizing of the cells shall be active

`datesy.file_IO.xls_file.`**`write_multi_sheet_from_DataFrames`**(*file_name*, *data_frames*, *sheet_order=None*, *auto_size_cells=True*)

Save multiple pandas.DataFrames to one file

Parameters

- **`file_name`** (*`str`*) – the file_name to save under. if no ending is provided, saved as .xlsx
- **`data_frames`** (*`dict {sheet_name: DataFrame}`*) – dict of data_frames
- **`sheet_order`** (*`dict {int: str}, list, optional`*) – either a dictionary with the specified positions in a dictionary with positions as keys (integers) or in a list
- **`auto_size_cells`** (*`bool, optional`*) – if the auto-sizing of the cells shall be active

`datesy.file_IO.xls_file.`**`write_single_sheet_from_dict`**(*file_name*, *data*, *main_key_name=None*, *sheet=None*, *order=None*, *inverse=False*, *auto_size_cells=True*)

Save a dictionary (`{main_key_name: {data}}`) as xlsx document to file Uses the dict_to_pandas_data_frame method of this package for converting the dictionary to pandas.DataFrame.

Parameters

- **`file_name`** (*`str`*) – the file_name to save under. if no ending is provided, saved as .xlsx
- **`data`** (*`dict`*) – the dictionary to be saved as xlsx `{main_key_name: {data}}`
- **`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
- **`sheet`** (*`str, optional`*) – a sheet name for the handling
- **`order`** (*`dict, list, optional`*) – either a dictionary with the specified positions in a dictionary with positions as keys (integers) or in a list
- **`inverse`** (*`bool, optional`*) – if columns and rows shall be switched
- **`auto_size_cells`** (*`bool, optional`*) – if the auto-sizing of the cells shall be active

`datesy.file_IO.xls_file.`**`write_multi_sheet_from_dict_of_dicts`**(*file_name*, *data*, *order=None*, *auto_size_cells=True*)

Save dictionaries (`{sheet_name: {main_key_name: {data}}}`) as xlsx document to file Uses the

dict_to_pandas_data_frame method of this package for converting the dictionary to pandas.DataFrame.

Parameters

- **file_name** (*str*) – the file_name to save under. if no ending is provided, saved as .xlsx
- **data** (*dict*) – the dictionary to be saved as xlsx `{sheet_name: {main_key_name: {data}}}`
- **order** (*dict, list, optional*) – either a dictionary with the specified positions in a dictionary with positions as keys (integers) or in a list
- **auto_size_cells** (*bool, optional*) – if the auto-sizing of the cells shall be active

xml_file module

The xml_file module takes care of all I/O interactions concerning xml files

`datesy.file_IO.xml_file.`**load**(*path*)
Load(s) json file(s) and returns the dictionary/-ies Specifying a file_name: one file will be loaded. Specifying a directory: all **.json* files will be loaded.

Parameters **path** (*str*) – path to a file_name or directory

Returns dictionary representing the json `{file_name: {data}}`

Return type dict

`datesy.file_IO.xml_file.`**load_single**(*file_name*)
Load a single xml file

Parameters **file_name** (*str*) – file_name to load from

Returns the xml as ordered dict `{collections.OrderedDict}`

Return type dict

`datesy.file_IO.xml_file.`**load_these**(*file_name_list*)
Load specified xml files and return the data in a dictionary with file_name as key

Parameters **file_name_list** (*list*) – list of file_names to load from

Returns the dictionaries from the files as values of file_name as key `{file_name: {collections.OrderedDict}`

Return type dict(collections.OrderedDict)

`datesy.file_IO.xml_file.`**load_all**(*directory*)
Load all xml files in the directory and return the data in a dictionary with file_name as key

Parameters **directory** (*str*) – the directory containing the xml files

Returns the dictionaries from the files as values of file_name as key `{file_name: {collections.OrderedDict}}`

Return type dict(collections.OrderedDict)

`datesy.file_IO.xml_file.`**write**(*file_name*, *data*, *main_key_name=None*)
Save xml file from dict or collections.OrderedDict to file

Parameters

- **file_name** (*str*) – the file_name to save under. if no ending is provided, saved as .xml
- **data** (*dict, collections.OrderedDict*) – the dictionary to be saved as xml

- **main_key_name** (*str*) – if the dict/OrderedDict does not have the main key as a single key present ({main_element_name: dict}), it needs to be specified

All actions of interacting with databases are to be found here:

2.1.2 Database I/O subpackage

All functions of *datesy* taking care of database I/O are listed here. The functionality is the same for all supported database types. Simply exchange `_db_helper.Database` with the desired database.

Database

class datesy.database_IO._db_helper.**Database**(*host*, *port*, *user*, *password*, *database*, *auto_creation=False*)

Representing a database as an object

On initialization the connection to the database is established. For clean working please call `close()` at end of db usage.

Parameters

- **host** (*str*) – *url* or *ip* of host
- **port** (*int*) – port_no
- **user** (*str*) – user_name
- **password** (*str*) – password for this user
- **database** (*str*) – the database to connect to
- **auto_creation** (*bool, optional*) – if all tables shall be initiated as variables of object
- **kwargs** – specific information to database, see details to each database

close()

Close connection to database

name

Name of the database

Returns

Return type str

table(*table_name*)

Return a database_table as an object

Parameters **table_name** (*str*) – the desired table

Returns class Table as representation of table

Return type *Table*

tables

Get the available tables of database

Returns representing all tables of database

Return type list

update_table_data()

Update the data concerning the list of available tables

Returns available tables at database

Return type list

Table

class datesy.database_IO._db_helper.**Table**(*table_name*, *database*)
Create a representation of a database table

Parameters

- **table_name** (*str*) –
- **database** (Database) –

__getitem__(*key*)
Get row of primary key

works like `value = database.table[key]`

Parameters **key** (*any*) – matching value in primary column

Returns tuple items representing every matched row in database

Return type *Row*

__setitem__(*primary_key*, *row*)
Set/update a single row for primary key

works like `database.table[key] = row`

Parameters

- **primary_key** (*any, None*) – the value of the primary column. If None -> new row inserted
- **row** (*list, dict*) – the row data in either correct order or in a dict with column_name

__delitem__(*key*)
Delete single row for given primary key

works like `del database.table[key]`

Parameters **key** (*any*) – matching value in primary column

delete_where(**args*, ***kwargs*)
Delete rows matching the where conditions

Parameters

- **args** (*conditions*) –
- **kwargs** (*conditions*) –

execute_raw_sql(*sql_query*)
Execute raw sql statements

Parameters **sql_query** (*str*) – sql query

Returns data from database

Return type list

get_where(**args*, ***kwargs*)
Get rows where value matches the defined column: `columns=key`

Parameters ****kwargs** – column = key values

Returns tuple items representing every matched row in database

Return type list(*Row*)

insert(*row: (<class 'list'>, <class 'dict'>)*, *primary_key=None*)
Insert new row

Parameters

- **row** (*list, dict*) – row_data
- **primary_key** (*any, optional*) – primary_key optional for tables with primary_key

primary
Get the primary key of this table

Returns the primary column as string if one exists

Return type str, None

run_query(*debug=False*)
Run the currently composed query :param debug: if the query shall be printed to command_line :type debug: bool, optional

Returns data from database

Return type list

schema
Get schema of table

Returns dictionary containing the column as key and schema_info as dictionary for every column

Return type OrderedDict

update_primary_data()
Update the primary key of the table

Returns the primary column as string if one exists

Return type str, None

update_schema_data()
Update the schema of the table

Returns dictionary containing the column as key and schema_info as dictionary for every column

Return type OrderedDict

update_where(*values: (<class 'list'>, <class 'dict'>)*, **args*, *primary_key=None*, *limit_rows: int = False*, ***kwargs*)
Update all rows based on given conditions

Parameters

- **values** (*list, dict*) – new values to set. either as a full row in a list or specified columns with dictionary
- **args** – conditions
- **primary_key** (*any, optional*) – the value of the primary column (if table has primary_key)
- **limit_rows** (*int*) – number of rows to affect

- **kwargs** – conditions

Row

class datesy.database_IO._db_helper.**Row**(*table*, *data*)
Representation of a database row entry

Parameters

- **table** (Table) – table belonging to
- **data** (*list, tuple, dict*) – data to represent

__getitem__(*column*)
Get column_value of row

works like `value = database.table.row[column]`

Parameters **column** (*int, str*) – position in table or column_name

Returns

Return type *Item*

__setitem__(*column*, *value*)
Set new value to column

works like `database.table.row[column] = value`

Parameters

- **column** (*int, str*) – position in table or column_name
- **value** (*any*) – new value

__delitem__(*column*)
Delete/reset to default a column

works like `del database.table.row[column]`

Parameters **column** (*int, str*) – position in table or column_name

sync(**missing_columns*)
Update row from database to local

Parameters **missing_columns** (*str, optional*) – if rows shall be left out when updating (e.g. if known that a timestamp has changed and it shall be fetched)

Item

class datesy.database_IO._db_helper.**Item**(*row*, *column*, *table*, *value=None*)
Representation of a database entry

Parameters

- **row** (Row) –
- **column** (*str*) –
- **table** (Table) –
- **value** (*any*) –

__set__(*instance*, *value*)
Set new value

Parameters

- **instance** –
- **value** (*any*) – new value

__delete__(*instance*)
Delete/reset to default this value

column
Column this item belonging to

Returns column

Return type str

database
Database this item is belonging to

Returns database

Return type *Database*

sync()
Update entry from database to local

table
Table this item is belonging to

Returns table

Return type *Table*

value
Value of this item

Returns value

Return type any

2.2 Submodules

2.3 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

2.4 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

2.5 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 *minimal_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

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 File interaction

Check here all the examples for interacting with files

3.2.1 Selecting Files

The file selection provides multiple ways for retrieving the desired files. All selecting functions contain three possibility to match:

- file_ending: matching 100% of the part after the last `.`
- pattern: standard pattern for finding fixed strings with wildcards (like `SomeFixedName_*.csv` with * representing all kinds of string)
- regex: a standard regular_expression (regex) matching the filename

The easiest way to get the latest file matching containing the name `DataSource1` in the beginning and which is a `.csv` file:

```
# Explicit way
file_name = datesy.file_selection.get_latest_file_from_directory(
                directory="path/to/directory",
                pattern="DataSource1*",
                file_ending="csv"
                )

# Shortened way
file_name = datesy.file_selection.get_latest_file_from_directory(
                directory="path/to/directory",
                pattern="DataSource1*.csv"
                )
```

3.2.2 File I/O

The library provides a standardized way of interacting with files. For every file-type in the *file_IO* subpackage, there exist load- & write-functions following the same pattern. Only exception is the *xls* module due to the characteristics of sheets.

All-in-one/doing-all-the-magic loading functions

The most easy way to load data is with the *load_file-type* function. It is a shortcut for the specific ways of loading data in each file-type specific module:

```
data = datesy.load_csv(path="path/to/file.csv")
# data is list of lists representing the csv file

data = datesy.load_json(path="path/to/file.json")
# data is dictionary representing json file
```

The most easy way to write data is with the *write_file-type* function. It is again a shortcut to file-type specific modules:

```
# data is written to the csv file
datesy.write_csv(file_name="path/to/file.csv", data=data_to_write)

# data is written to the json file
datesy.load_json(file_name="path/to/file.json", data=data_to_write)
```

File-type specific modules: advanced reading/writing

For every file-type exist more specific functions for reading & writing the data. The presented examples from above are redirecting to the most general functions in the packages.

If using a IDE, the implemented functions will be shown to you directly with typing *datesy./datesy.json_file.*. If in interactive mode, simply type *datesy.__all__/datesy.json_file.__all__*. Switch the *json_file* to whatever submodule/-package you need.

Reading

The reading of the files is fairly simple

```
# load single json file
data = datesy.json_file.load_single(path="path/to/file.json")
# data is representing the json file


# load specific list of json files
data = datesy.json_file.load_these(file_name_list=["path/to/file1.json", "path/to/
↪file2.json"])
# data is representing both json files; {file_name: json_file_value}


# load all json files from a directory
data = datesy.json_file.load_all(directory="/path/to/directory")
# data is representing all json files of this directory; {file_name: json_file_value}



# doing all of the above depending if `path` is file, list_of_files or directory
data = datesy.load_json(path="path/to/any")
# depending if single file or multiple files either dictionary representing json file␣
↪or {file_name: json_file_value}
```

The last function is also reachable with the shortcut stated in the very beginning of the examples: `datesy.load_json`

Writing

For writing, the *datesy* package provides sometimes some more options for making life easier. The concept this package is designed, is to work most likely with data in form of a dictionary. Therefore, often shortcuts are provided.

Let's have a look to row-based file-type *csv* (*comma separated values*): You can provide either row-based data (in python this would be a list of lists), or you can provide a dictionary instead and let *datesy* take care of the conversion. This little magic is part of the *datesy.convert* module, more details below.

```
# lets start with row-based data
example_rows = [
                ["Header1", "Header2", "Header3"],
                ["Value11", "Value12", "Value13"],
                ["Value21", "Value22", "Value23"]
               ]
datesy.csv_file.write_from_rows(file_name="path/to/csv_file.csv", rows=example_rows)

# The result in the file:
# Header1,Header2,Header3
# Value11,Value12,Value13
# Value21,Value22,Value23


# in difference with data in form of a dictionary
example_dict = {
                  "Header1": {
                    "Value11": {
                      "Header2": "Value12",
                      "Header3": "Value13"
                    },
                    "Value21": {
```

(continues on next page)

(continued from previous page)

```
                    "Header2": "Value22",
                    "Header3": "Value23"
                  }
                }
              }
datesy.csv_file.write_from_dict(file_name="path/to/csv_file.csv", data=example_dict)

# The result in the file is the same:
# Header1,Header2,Header3
# Value11,Value12,Value13
# Value21,Value22,Value23

# additionally the data can be provided without the naming of the main_key
#  (in this case "Header1")
example_dict2 = {
                  "Value11": {
                    "Header2": "Value12",
                    "Header3": "Value13"
                  },
                  "Value21": {
                    "Header2": "Value22",
                    "Header3": "Value23"
                  }
                }

datesy.csv_file.write_from_dict(
    file_name="path/to/csv_file.csv",
    data=example_dict,
    main_key_name="Header1",
    main_key_position=0
)

# The result in the file is still the same:
# Header1,Header2,Header3
# Value11,Value12,Value13
# Value21,Value22,Value23
```

Again, there is a function combining both writing methods, available also with a shortcut stated in the very beginning of the examples: `datesy.write_csv`

xls/xlsx Files

The Microsoft Excel file interaction works slightly different since sheets are a feature not available to standard file formats like *json*, *csv* or *xml*. The standard output format is Pandas DataFrame.

Yet, interaction is still fairly simple:

```
data_frame = datesy.xls_file.load_single_sheet(file_name="path/to/file.xls")     # .
↪xlsx works with the same function
# returns a pandas.data_frame from first sheet

# you can specify a sheet_name
data_frame = datesy.xls_file.load_single_sheet(file_name="path/to/file.xls", sheet=
↪"Sheet_Name")
# returns a pandas.data_frame from sheet with provided name
```

(continues on next page)

(continued from previous page)

```
# of course multiple sheets can be loaded
data = datesy.xls_file.load_these_sheets(file_name="path/to/file.xls", sheets=["Sheet_
↪Name1", "Sheet_Name2"])
# just like the other loading functions, the sheet_name is the key in a dictionary␣
↪containing the data_frame as value
# {"Sheet_Name": DataFrame}

# loading all sheets
data = datesy.xls_file.load_all_sheets(file_name="path/to/file.xls")
# {"Sheet_Name": DataFrame}


# reading multiple files is possible as well
data = datesy.xls_file.load_these_files(file_name_list=["path/to/file1.xls", "path/to/
↪file2.xls"])
# {file_name: {sheet_name: DataFrame}}
```

3.3 Converting Data

Check here all the examples for converting data easily with *datesy*

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"],
                ["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"]
```

(continues on next page)

(continued from previous page)

```
                ]

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)

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"]
```

(continues on next page)

(continued from previous page)

```
                ]

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"
                }
              }
            }

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",
```

(continues on next page)

(continued from previous page)

```
                    "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 = [
                ["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"
                  }
                }
              }
```

(continues on next page)

(continued from previous page)

```
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"]
                ]
```

CHAPTER 4

Release Notes

4.1 0.8.0

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

4.1.1 Features

- database abstraction available
  - database
  - table
  - row
  - item
- database interaction now possible for:
  - mysql

4.2 0.7.1

saving jsons: beautified to human readability & sorting keys available

4.3 0.7.0

initial release

4.3.1 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

4.3.2 Bug Fixes

- initial release

CHAPTER 5

Indices and tables

- genindex
- modindex
- search

Python Module Index

d

Index

Symbols

C

D

E

F

G

I

L

M

N

P

R

S

T

U

V

W

X