fil$_{iO}$

Release 0.1.3

Sep 08, 2021

Contents

Fil_IO: simplifying data reading and writing

CHAPTER 1

Introduction

Selecting, reading and writing files as easy as it can be!

1.1 Main Usage

With `fil_io` you can:

1. selecting files based on e.g. file_ending, naming pattern or last change date
2. simple reading one or more files at once
3. simple writing files

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.

From this datesy and `fil_io` derived.

1.2.2 Future Development

With simplifying file IO, adding some more file types or improving selecting might come in handy.

1.3 Limitations

This package is designed to be an alternative and simplified API for file interaction. All and more functionality is already available either built-in from python or from other packages.

Yet, for quite some tasks this packages makes a lot more fun to use and is cleaning your code.

CHAPTER 2

The fil_io package

The *fil_io* package provides you some nice tools to read and write more than one file at a time. This includes smart file selection.

2.1 selection module

The file_selection module provides multiple supporting functions for interaction with files

`fil_io.select.`**`get_newest_file_from_directory`**(*directory*, *file_ending=None*, *pattern=None*, *regex=None*)

Return the latest file_name (optionally filtered) from a directory

Parameters

- **`directory`** (*`str, Path`*) – the directory where to get the latest file_name from
- **`file_ending`** (*`str, list, 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

`fil_io.select.`**`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, Path`*) – the directory containing the desired files
- **`file_ending`** (*`str, list, 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

`fil_io.select.`**return_file_list_if_directory**(*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, Path*) – the path to test if directory
- **file_ending** (*str, list, 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

`fil_io.select.`**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

2.2 json module

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

`fil_io.json.`**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*) – path to a file_name or directory

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

Return type dict

fil_io.json.**load_single**(*file_name*)
Load a single json file

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

Returns the loaded json as a dict {data}

Return type dict

fil_io.json.**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** (*Iterable*) – 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)

fil_io.json.**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, Path*) – 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)

fil_io.json.**write**(*data*, *file_name*, *beautify=True*, *sort=False*)
Save json from dict to file

Parameters

- **file_name** (*str, Path*) – 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)

2.3 csv module

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

fil_io.csv.**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*) – 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

fil_io.csv.**load_single**(*file_name*, ***kwargs*)
Load a csv file and return the rows

Parameters

- **file_name** (*str, Path*) – 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

fil_io.csv.**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** (*Iterable*) – 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

fil_io.csv.**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, Path*) – 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

fil_io.csv.**write**(*data*, *file_name*, *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.

Parameters

- **file_name** (*str, Path*) – 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

fil_io.csv.**write_from_rows**(*rows*, *file_name*, ***kwargs*)

Save row based document from rows to file

Parameters

- **file_name** (*str, Path*) – 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

fil_io.csv.**write_from_dict**(*data*, *file_name*, *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, Path*) – 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

2.4 xls module

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

fil_io.xls.**load_single_sheet**(*file_name*, *sheet=None*)

Load a xls(x) file's (first) sheet to a pandas.DataFrame

Parameters

- **file_name** (*str, Path*) – 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

fil_io.xls.**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, Path*) – 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)

fil_io.xls.**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, Path*) – 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

fil_io.xls.**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** (*Iterable*) – 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

fil_io.xls.**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, Path*) – 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

fil_io.xls.**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*) – path to a file_name or directory

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

Return type dict

fil_io.xls.**write_single_sheet_from_DataFrame**(*data_frame*, *file_name*, *sheet_name=None*, *auto_size_cells=True*)
Save a pandas.DataFrame to file

Parameters

- **file_name** (*str, Path*) – 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

fil_io.xls.**write_multi_sheet_from_DataFrames**(*data_frames*, *file_name*, *sheet_order=None*, *auto_size_cells=True*)
Save multiple pandas.DataFrames to one file

Parameters

- **file_name** (*str, Path*) – 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

fil_io.xls.**write_single_sheet_from_dict**(*data*, *file_name*, *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 for converting the dictionary to pandas.DataFrame.

Parameters

- **file_name** (*str, Path*) – 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

fil_io.xls.**write_multi_sheet_from_dict_of_dicts**(*data*, *file_name*, *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 for converting the dictionary to pandas.DataFrame.

Parameters

- **file_name** (*str, Path*) – 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

2.5 xml module

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

fil_io.xml.**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*) – path to a file_name or directory

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

Return type dict

`fil_io.xml.`**`load_single`**(*file_name*)
Load a single xml file

Parameters **`file_name`** (*`str, Path`*) – file_name to load from

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

Return type dict

`fil_io.xml.`**`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`** (*`Iterable`*) – 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)

`fil_io.xml.`**`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, Path`*) – 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)

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

Parameters

- **`file_name`** (*`str, Path`*) – 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

CHAPTER 3

Examples

The examples are split in two parts: the selection of files and the standard interface for reading/writing files.

3.1 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:

```
from fil_io import select

# Explicit way
file_name = select.get_newest_file_from_directory(
                directory="path/to/directory",
                pattern="DataSource1*",
                file_ending="csv"
                )

# Shortened way
file_name = select.get_newest_file_from_directory(
                directory="path/to/directory",
                pattern="DataSource1*.csv"
                )
```

3.2 File reading / writing

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.

Examples are given mostly with csv module, switch the `csv` *to whatever submodule/-package you need.*

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

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

```
from fil_io import csv

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

data = csv.load(path="directory/of/multiple/files")
# data is dictionary representing all csv files with {file_name: file_content}
```

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

```
# data is written to the csv file
from fil_io import csv
csv.write(file_name="path/to/file.csv", data=data_to_write)

# data is written to the json file
from fil_io import json
json.write(file_name="path/to/file.json", data=data_to_write)
```

3.2.2 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 after importing the file-specific module directly with typing `csv.` and hitting *tab*. If in interactive mode, simply run `csv.__all__`.

Reading

The reading of the files is fairly simple

```
from fil_io import csv

# load single csv file
data = csv.load_single(file_name="path/to/file.csv")
# data is representing the csv file


# load specific list of csv files
data = csv.load_these(file_name_list=["path/to/file1.csv", "path/to/file2.csv"])
# data is representing both csv files; {file_name: file_content}
```

(continues on next page)

(continued from previous page)

```
# load all csv files from a directory
data = csv.load_all(directory="/path/to/directory")
# data is representing all csv files of this directory; {file_name: file_content}



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

Writing

For writing, the *fil_io* 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 *fil_io* take care of the conversion. This little magic is part of the *fil_io.convert* module, more details below.

```
from fil_io import csv

# lets start with row-based data
example_rows = [
                ["Header1", "Header2", "Header3"],
                ["Value11", "Value12", "Value13"],
                ["Value21", "Value22", "Value23"]
               ]
csv.write_from_rows(file_name="path/to/csv.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": {
                      "Header2": "Value22",
                      "Header3": "Value23"
                    }
                  }
                }
csv.write_from_dict(file_name="path/to/csv.csv", data=example_dict)

# The result in the file is the same:
# Header1,Header2,Header3
```

(continues on next page)

(continued from previous page)

```
# 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"
                  }
                }

csv.write_from_dict(
    file_name="path/to/csv.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: `csv.write`

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:

```
from fil_io import xls

data_frame = xls.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 = xls.load_single_sheet(file_name="path/to/file.xls", sheet="Sheet_Name")
# returns a pandas.data_frame from sheet with provided name


# of course multiple sheets can be loaded
data = xls.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}
```

(continues on next page)

(continued from previous page)

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


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

CHAPTER 4

Release Notes

4.1 0.1.2

Windows filesystem support

4.2 0.1.1

initial release

4.2.1 Features

- reading/writing file types
  - csv
  - json
  - xml
  - xls(x)

CHAPTER 5

Indices and tables

- genindex
- modindex
- search

Python Module Index

f

Index