SPSS Converter

Release 0.1.1

Insight Industry Inc.

Mar 31, 2021

CONTENTS:

Simple utility for converting data to/from SPSS data files

Branch	Unit Tests
latest	
v.0.1	

CHAPTER

ONE

QUICKSTART: PATTERNS AND BEST PRACTICES

1.1 Installation

To install the **SPSS Converter**, just execute:

```
$ pip install spss-converter
```

1.2 Convert between SPSS and CSV

SPSS –> CSV

CSV –> SPSS

```
# Convert "my-spss-file.sav" to "my-csv-file.csv".
spss_converter.to_csv('my-spss-file.sav', target = 'my-csv-file.csv')
```

```
# Convert "my-csv-file.csv" to "my-spss-file.sav"
spss_converter.from_csv('my-csv-file.csv', target = 'my-spss-file.sav')
```

1.3 Convert between SPSS and JSON

SPSS –> JSON

JSON –> SPSS

```
# Convert "my-spss-file.sav" to "my-json-file.json" using a "records" layout
spss_converter.to_json('my-spss-file.sav',
                       target = 'my-json-file.json',
                       layout = 'records')

# Convert "my-spss-file.sav" to "my-json-file.json" using a "table" layout
spss_converter.to_json('my-spss-file.sav',
                       target = 'my-json-file.json',
                       layout = 'table')
```

```
# Convert "my-json-file.json" to "my-spss-file.sav" using a "records" layout
spss_converter.from_json('my-json-file.json',
                         target = 'my-spss-file.sav',
                         layout = 'records')

# Convert "my-json-file.sav" to "my-spss-file.json" using a "table" layout
spss_converter.from_json('my-json-file.json',
                         target = 'my-spss-file.sav',
                         layout = 'table')
```

1.4 Convert between SPSS and YAML

SPSS –> YAML

YAML –> SPSS

```
# Convert "my-spss-file.sav" to "my-yaml-file.yaml" using a "records" layout
spss_converter.to_yaml('my-spss-file.sav',
                       target = 'my-yaml-file.yaml',
                       layout = 'records')

# Convert "my-spss-file.sav" to "my-yaml-file.yaml" using a "table" layout
spss_converter.to_yaml('my-spss-file.sav',
                       target = 'my-yaml-file.yaml',
                       layout = 'table')
```

```
# Convert "my-yaml-file.yaml" to "my-spss-file.sav" using a "records" layout
spss_converter.from_yaml('my-yaml-file.yaml',
                         target = 'my-spss-file.sav',
                         layout = 'records')

# Convert "my-yaml-file.sav" to "my-spss-file.yaml" using a "table" layout
spss_converter.from_yaml('my-yaml-file.yaml',
                         target = 'my-spss-file.sav',
                         layout = 'table')
```

1.5 Convert between SPSS and Pandas `DataFrame`

SPSS –> `DataFrame`

`DataFrame` –> SPSS

```
# Convert "my-spss-file.sav" to df
df, meta = spss_converter.to_dataframe('my-spss-file.sav')
```

```
# Convert the Pandas DataFrame df to "my-spss-file.sav"
spss_converter.from_dataframe(df, target = 'my-spss-file.sav', metadata = meta)
```

1.6 Convert between SPSS and `dict`

SPSS –> `dict`

`dict` –> SPSS

```
# Convert "my-spss-file.sav" to a dict using a "records" layout
as_dict = spss_converter.to_dict('my-spss-file.sav',
                                 layout = 'records')

# Convert "my-spss-file.sav" to a dict using a "table" layout
as_dict = spss_converter.to_dict('my-spss-file.sav',
                                 layout = 'table')
```

```
# Convert as_dict to "my-spss-file.sav"
spss_converter.from_dict(as_dict,
                         target = 'my-spss-file.sav')
```

1.7 Convert between SPSS and Excel

SPSS –> Excel

Excel –> SPSS

```
# Convert "my-spss-file.sav" to "my-excel-file.xlsx".
spss_converter.to_excel('my-spss-file.sav', target = 'my-excel-file.xlsx')
```

```
# Convert "my-csv-file.csv" to "my-spss-file.sav"
spss_converter.from_excel('my-excel-file.xlsx', target = 'my-spss-file.sav')
```

1.8 Get the Metadata from an SPSS File

```
# Retrieve Metadata from the SPSS file "my-spss-file.sav"
meta = spss_converter.get_metadata('my-spss-file.sav')
```

1.9 Change the Metadata for a Given `DataFrame`

```
# Apply the metadata in updated_meta to the dataframe in df.
spss_converter.apply_metadata(df, updated_meta)
```

CHAPTER

TWO

USING THE SPSS CONVERTER

2.1 Introduction

The **SPSS Converter** library is a simple wrapper around the Pyreadstat and Pandas libraries that provides a clean and simple API for reading data files in a variety of formats and converting them to a variety of formats. The semantics are super simple, and should be as simple as: `spss_converter.to_csv('my-spss-file.sav')` or `spss_converter.from_json('my-json-file.json')`.

2.2 Converting Data from SPSS

To read from SPSS files and convert them to a different format you can use functions whose names start with `spss_converter.to_*`. The examples below provide specifics:

2.2.1 Converting to Pandas DataFrame

To convert from an SPSS file to a Pandas `DataFrame`, simply call the *`to_dataframe()`* function:

```
import spss_converter

df, metadata = spss_converter.to_dataframe('my-spss-file.sav')
```

The code above will read your data from the file `my-spss-file.sav`, convert it into a Pandas `DataFrame`, and generate an *`spss_converter.Metadata`* representation of the SPSS file's meta-data, which includes its data map, labeling, etc.

See also:

- *`to_dataframe()`*

2.2.2 Converting to CSV

To read data from an SPSS file and convert it into a CSV file, simply call the *`to_csv()`* function:

```
import spss_converter

as_csv = spss_converter.to_csv('my-spss-file.sav')
# Will store the contents of the CSV as a string in as_csv.

spss_converter.to_csv('my-spss-file.sav', target = 'my-csv-file.csv')
# Will save the CSV data to the file my-csv-file.csv.
```

Both lines of code above will read the SPSS data from `my-spss-file.sav`, but the first line will store it in the `str` variable `as_csv`. The second will instead write it to the file `my-csv-file.csv`.

See also:

- *`to_csv()`*

2.2.3 Converting to JSON

To read data from an SPSS file and convert it into a JSON object, simply call the *`to_json()`* function:

As Records (default)

As Table

```
import spss_converter

as_json = spss_converter.to_json('my-spss-file.sav', layout = 'records')
# Stores the JSON data as a string in the variable as_json.

spss_converter.to_json('my-spss-file.sav',
                       target = 'my-json-file.json',
```

(continues on next page)

(continued from previous page)

```
                        layout = 'records')
# Stores the JSON data in the file "my-json-file.json".
```

```
import spss_converter

as_json = spss_converter.to_json('my-spss-file.sav', layout = 'table')
# Stores the JSON data as a string in the variable as_json.

spss_converter.to_json('my-spss-file.sav',
                        target = 'my-json-file.json',
                        layout = 'table')
# Stores the JSON data in the file "my-json-file.json".
```

The **SPSS Converter** supports two different layouts for JSON representation of data:

- **Records**. This layout returns a JSON collection (array) of JSON objects. Each object in the collection represents one record from the SPSS file. The object is a a set of key/value pairs where each key represents a variable/column in the SPSS file and its value represents the value of that variable/column for that respondent. This is the default layout.
- **Table**. This layout returns a JSON object that includes a `schema` with the data map, and a separate `data` key which contains a collection (array) of objects where each object represents a single record from the SPSS data file.

Note: If no `target` is supplied, then the JSON representation is stored in-memory in the return value. If a `target` is supplied, then the JSON representation will be written to this file.

See also:

- *to_json()*

2.2.4 Converting to YAML

To read data from an SPSS file and convert it into a YAML object, simply call the *to_yaml()* function:

As Records (default)

As Table

```
import spss_converter

as_yaml = spss_converter.to_yaml('my-spss-file.sav', layout = 'records')
# Stores the YAML data as a string in the variable as_yaml.

spss_converter.to_yaml('my-spss-file.sav',
                        target = 'my-yaml-file.yaml',
                        layout = 'records')
# Stores the YAML data in the file "my-yaml-file.yaml".
```

```
import spss_converter

as_yaml = spss_converter.to_yaml('my-spss-file.sav', layout = 'table')
# Stores the YAML data as a string in the variable as_yaml.

```

(continues on next page)

(continued from previous page)

```
spss_converter.to_yaml('my-spss-file.sav',
                       target = 'my-yaml-file.yaml',
                       layout = 'table')
# Stores the YAML data in the file "my-yaml-file.yaml".
```

The **SPSS Converter** supports two different layouts for YAML representation of data:

- **Records**. This layout returns a YAML collection (array) of YAML objects. Each object in the collection represents one record from the SPSS file. The object is a a set of key/value pairs where each key represents a variable/column in the SPSS file and its value represents the value of that variable/column for that respondent. This is the default layout.
- **Table**. This layout returns a YAML object that includes a `schema` with the data map, and a separate `data` key which contains a collection (array) of objects where each object represents a single record from the SPSS data file.

Note: If no `target` is supplied, then the YAML representation is stored in-memory in the return value. If a `target` is supplied, then the JSON representation will be written to this file.

See also:

- *to_yaml()*

2.2.5 Converting to Excel

To read data from an SPSS file and convert it into a Microsoft Excel file, simply call the *to_excel()* function:

```
import spss_converter

as_excel = spss_converter.to_excel('my-spss-file.sav')
# Will store the contents of the Excel file as a binary object in as_excel.

spss_converter.to_excel('my-spss-file.sav', target = 'my-excel-file.xlsx')
# Will save the Excel data to the file my-excel-file.xlsx.
```

Both lines of code above will read the SPSS data from `my-spss-file.sav`, but the first line will store it in the `bytes` variable `as_excel`. The second will instead write it to the file `my-excel-file.xlsx`.

See also:

- *to_excel()*

2.2.6 Converting to `dict`

To read data from an SPSS file and convert it into a `dict` object, simply call the *to_dict()* function:

As Records (default)

As Table

```
import spss_converter

as_dict = spss_converter.to_dict('my-spss-file.sav', layout = 'records')
# Stores the data as a dict or list of dict in the variable as_dict.
```

```
import spss_converter

as_dict = spss_converter.to_dict('my-spss-file.sav', layout = 'table')
# Stores the data as a dict or list of dict in the variable as_dict.
```

The **SPSS Converter** supports two different layouts for `dict` representation of data:

- **Records**. This layout returns a `list` of `dict` objects. Each object in the list represents one record from the SPSS file. The object is a `dict` whose keys each represent a variable/column in the SPSS file and whose values represent the value of that variable/column for that respondent. This is the default layout.
- **Table**. This layout returns a `dict` object that includes a `schema` key with the data map, and a separate `data` key which contains a `list` of objects where each object represents a single record from the SPSS data file.

See also:

- *to_dict()*

2.3 Converting Data to SPSS

To convert other sources of data to SPSS format, you can simply use any function whose names start with `spss_converter.from_*`. The examples below provide specifics:

2.3.1 Converting from Pandas `DataFrame`

To generate an SPSS file from a Pandas `DataFrame`, simply call the `from_dataframe()` function:

Note: The examples below all assume that the variable `df` contains the `DataFrame` whose data will be converted to SPSS format and the variable `meta` contains the *Metadata* that describes that data frame.

```
import spss_converter

as_spss = spss_converter.from_dataframe(df, metadata = meta)
# Will store the SPSS data in-memory in a binary bytes object named as_spss.

spss_converter.from_dataframe(df, target = 'my-spss-file.sav', metadata = meta)
# Will store the SPSS data to the hard drive in the file named "my-spss-file.sav".
```

The code above will convert the data in the `DataFrame` named `df`, and generate it in SPSS format either in-memory or on the hard drive.

See also:

- *from_dataframe()*

2.3.2 Converting from CSV

To read data from a CSV file and convert it into SPSS format, simply call the *from_csv()* function:

```
import spss_converter

as_spss = spss_converter.from_csv('my-csv-file.csv')
# Will store the contents of the CSV file as an in-memory binary object called as_
↪spss.

spss_converter.from_csv('my-csv-file.csv', target = 'my-spss-file.sav')
# Will save the CSV data to the file my-spss-file.sav.
```

Both lines of code above will read the data from `my-csv-file.csv`, but the first line will store it in the `bytesIO` variable `as_spss`. The second will instead write it to the file `my-spss-file.sav`.

See also:

- *from_csv()*

2.3.3 Converting from `dict`

To read data from a `dict` and convert it into an SPSS format, simply call the *from_dict()* function:

```
import spss_converter

as_spss = spss_converter.from_dict(as_dict)
# Stores the data in-memory in the variable as_spss.

spss_converter.from_dict(as_dict, target = 'my-spss-file.sav')
# Stores the data on the hard drive in the file named "my-spss-file.sav".
```

See also:

- *from_dict()*

2.3.4 Converting from JSON

To read data from a JSON file and convert it into SPSS format, simply call the *from_json()* function:

As Records (default)

As Table

```
import spss_converter

as_spss = spss_converter.from_json('my-json-file.json', layout = 'records')
# Stores the SPSS data in-memory in the variable as_spss.

spss_converter.from_json('my-json-file.json',
                         target = 'my-spss-file.sav',
                         layout = 'records')
# Stores the SPSS data in the file "my-spss-file.sav".
```

```
import spss_converter

```

(continues on next page)

(continued from previous page)

```
as_spss = spss_converter.from_json('my-json-file.json', layout = 'table')
# Stores the SPSS data in-memory in the variable as_spss.

spss_converter.from_json('my-json-file.json',
                         target = 'my-spss-file.sav',
                         layout = 'table')
# Stores the SPSS data in the file "my-spss-file.sav".
```

The **SPSS Converter** supports two different layouts for JSON representation of data:

- **Records**. This layout expects a JSON collection (array) of JSON objects. Each object in the collection represents one record in the SPSS file. The object is a a set of key/value pairs where each key represents a variable/column in the SPSS file and its value represents the value of that variable/column for that respondent. This is the default layout.
- **Table**. This layout returns a JSON object that includes a `schema` with the data map, and a separate `data` key which contains a collection (array) of objects where each object represents a single record in the SPSS data file.

Note: If no `target` is supplied, then the SPSS representation is stored in-memory in the return value. If a `target` is supplied, then the SPSS representation will be written to this file.

Tip: The *`from_json()`* function can accept either a filename or a string with JSON data.

See also:

- *`from_json()`*

2.3.5 Converting from YAML

To read data from a YAML file and convert it into SPSS format, simply call the *`from_yaml()`* function:

As Records (default)

As Table

```
import spss_converter

as_spss = spss_converter.from_yaml('my-yaml-file.yaml', layout = 'records')
# Stores the SPSS data in-memory in the variable as_spss.

spss_converter.from_yaml('my-yaml-file.yaml',
                         target = 'my-spss-file.sav',
                         layout = 'records')
# Stores the SPSS data in the file "my-spss-file.sav".
```

```
import spss_converter

as_spss = spss_converter.from_yaml('my-yaml-file.yaml', layout = 'table')
# Stores the SPSS data in-memory in the variable as_spss.

spss_converter.from_yaml('my-yaml-file.yaml',
                         target = 'my-spss-file.sav',
```

(continues on next page)

(continued from previous page)

```
                          layout = 'table')
# Stores the SPSS data in the file "my-spss-file.sav".
```

The **SPSS Converter** supports two different layouts for YAML representation of data:

- **Records**. This layout expects a YAML collection (array) of YAML objects. Each object in the collection represents one record in the SPSS file. The object is a a set of key/value pairs where each key represents a variable/column in the SPSS file and its value represents the value of that variable/column for that respondent. This is the default layout.
- **Table**. This layout returns a YAML object that includes a `schema` with the data map, and a separate `data` key which contains a collection (array) of objects where each object represents a single record in the SPSS data file.

Note: If no `target` is supplied, then the SPSS representation is stored in-memory in the return value. If a `target` is supplied, then the SPSS representation will be written to this file.

Tip: The *`from_yaml()`* function can accept either a filename or a string with YAML data.

See also:

- *`from_yaml()`*

2.3.6 Converting to Excel

To read data from an Excel file and convert it into SPSS format, simply call the *`from_excel()`* function:

```
import spss_converter

as_excel = spss_converter.from_excel('my-excel-file.xlsx')
# Will store the contents of the SPSS data as a binary object in-memory in as_excel.

spss_converter.from_excel('my-excel-file.xlsx', target = 'my-spss-file.sav')
# Will save the SPSS data to the file my-spss-file.xlsx.
```

Both lines of code above will read the data from `my-excel-file.xlsx`, but the first line will store it in the `bytes` variable `as_excel`. The second will instead write it to the file `my-spss-file.sav`.

See also:

- *`from_excel()`*

2.4 Working with Metadata

Key to working with SPSS data is understanding the distinction between the raw data's storage format and the metadata that describes that data. Fundamentally, think of metadata as the map of how a value stored in the raw data (such as a numerical value `1`) can actually represent a human-readable labeled value (such as the labeled value `"Female"`).

The metadata for an SPSS file can itself be quite verbose and define various rules for what can and should be expected when analyzing the records in the SPSS file. Within the **SPSS Converter**, this meta-data is represented using the *Metadata* class.

Various functions that read SPSS data produce *Metadata* instances, and these instances can be manipulated to restate and adjust the human-readable labels applied to your SPSS data.

See also:

- *Metadata*
- *get_metadata()*
- *apply_metadata()*
- *ColumnMetadata*

CHAPTER

THREE

API REFERENCE

3.1 Reading Data from SPSS

3.1.1 to_dataframe

to_dataframe(*data: Union[bytes, _io.BytesIO, os.PathLike[Any]]*, *limit: Optional[int] = None*, *offset: int = 0*, *exclude_variables: Optional[List[str]] = None*, *include_variables: Optional[List[str]] = None*, *metadata_only: bool = False*, *apply_labels: bool = False*, *labels_as_categories: bool = True*, *missing_as_NaN: bool = False*, *convert_datetimes: bool = True*, *dates_as_datetime64: bool = False*, ***kwargs*)

Reads SPSS data and returns a `tuple` with a Pandas `DataFrame` object and relevant `Metadata`.

Parameters

- **data** (Path-like filename, `bytes` or `BytesIO`) – The SPSS data to load. Accepts either a series of bytes or a filename.
- **limit** (`int` or `None`) – The number of records to read from the data. If `None` will return all records. Defaults to `None`.
- **offset** (`int`) – The record at which to start reading the data. Defaults to 0 (first record).
- **exclude_variables** (iterable of `str` or `None`) – A list of the variables that should be ignored when reading data. Defaults to `None`.
- **include_variables** (iterable of `str` or `None`) – A list of the variables that should be explicitly included when reading data. Defaults to `None`.
- **metadata_only** (`bool`) – If `True`, will return no data records in the resulting `DataFrame` but will return a complete `Metadata` instance. Defaults to `False`.
- **apply_labels** (`bool`) – If `True`, converts the numerically-coded values in the raw data to their human-readable labels. Defaults to `False`.
- **labels_as_categories** (`bool`) – If `True`, will convert labeled or formatted values to Pandas categories. Defaults to `True`.

  > **Caution:** This parameter will only have an effect if the `apply_labels` parameter is `True`.

- **missing_as_NaN** (`bool`) – If `True`, will return any missing values as `NaN`. Otherwise will return missing values as per the configuration of missing value representation stored in the underlying SPSS data. Defaults to `False`, which applies the missing value representation configured in the SPSS data itself.
- **convert_datetimes** (`bool`) – if `True`, will convert the native integer representation of datetime values in the SPSS data to Pythonic `datetime`, or `date`, etc. representations (or Pandas `datetime64`, depending on the `dates_as_datetime64` parameter). If `False`, will leave the original integer representation. Defaults to `True`.
- **dates_as_datetime64** (`bool`) – If `True`, will return any date values as Pandas `datetime64` types. Defaults to `False`.

  > **Caution:** This parameter is only applied if `convert_datetimes` is set to `True`.

Returns A `DataFrame` representation of the SPSS data (or `None`) and a `Metadata` representation of the data's meta-data (value and labels / data map).

Return type pandas.DataFrame/None and Metadata

3.1.2 to_csv

to_csv(*data: Union[os.PathLike[Any], _io.BytesIO, bytes], target: Optional[Union[os.PathLike[Any], _io.StringIO]] = None, include_header: bool = True, delimter: str = '|', null_text: str = 'NaN', wrapper_character: str = '"', escape_character: str = '\\', line_terminator: str = '\r\n', decimal: str = '.', limit: Optional[int] = None, offset: int = 0, exclude_variables: Optional[List[str]] = None, include_variables: Optional[List[str]] = None, metadata_only: bool = False, apply_labels: bool = False, labels_as_categories: bool = True, missing_as_NaN: bool = False, convert_datetimes: bool = True, dates_as_datetime64: bool = False, **kwargs*)

Convert the SPSS data into a CSV string where each row represents a record of SPSS data.

Parameters

- **data** (Path-like filename, bytes or BytesIO) – The SPSS data to load. Accepts either a series of bytes or a filename.
- **target** (Path-like / StringIO / str / None) – The destination where the CSV representation should be stored. Accepts either a filename, file-pointer or a StringIO, or None. If None, will return a str object stored in-memory. Defaults to None.
- **include_header** (bool) – If True, will include a header row with column labels. If False, will not include a header row. Defaults to True.
- **delimiter** (str) – The delimiter used between columns. Defaults to |.
- **null_text** (str) – The text value to use in place of empty values. Only applies if wrap_empty_values is True. Defaults to 'NaN'.
- **wrapper_character** (str) – The string used to wrap string values when wrapping is necessary. Defaults to '.
- **escape_character** (str) – The character to use when escaping nested wrapper characters. Defaults to \.
- **line_terminator** (str) – The character used to mark the end of a line. Defaults to \r\n.
- **decimal** (str) – The character used to indicate a decimal place in a numerical value. Defaults to ..
- **limit** (int or None) – The number of records to read from the data. If None will return all records. Defaults to None.
- **offset** (int) – The record at which to start reading the data. Defaults to 0 (first record).
- **exclude_variables** (iterable of str or None) – A list of the variables that should be ignored when reading data. Defaults to None.
- **include_variables** (iterable of str or None) – A list of the variables that should be explicitly included when reading data. Defaults to None.
- **metadata_only** (bool) – If True, will return no data records in the resulting DataFrame but will return a complete Metadata instance. Defaults to False.
- **apply_labels** (bool) – If True, converts the numerically-coded values in the raw data to their human-readable labels. Defaults to False.

- **labels_as_categories** (bool) – If True, will convert labeled or formatted values to Pandas categories. Defaults to True.

  Caution: This parameter will only have an effect if the apply_labels parameter is True.

- **missing_as_NaN** (bool) – If True, will return any missing values as NaN. Otherwise will return missing values as per the configuration of missing value representation stored in the underlying SPSS data. Defaults to False, which applies the missing value representation configured in the SPSS data itself.
- **convert_datetimes** (bool) – if True, will convert the native integer representation of datetime values in the SPSS data to Pythonic datetime, or date, etc. representations (or Pandas datetime64, depending on the dates_as_datetime64 parameter). If False, will leave the original integer representation. Defaults to True.
- **dates_as_datetime64** (bool) – If True, will return any date values as Pandas datetime64 types. Defaults to False.

  Caution: This parameter is only applied if convert_datetimes is set to True.

Returns None if target was not None, otherwise a str representation of the CSV file.

Return type None or str

3.1.3 to_excel

to_excel(*data: Union[os.PathLike[Any], _io.BytesIO, bytes], target: Optional[Union[os.PathLike[Any], _io.BytesIO, pandas.io.excel._base.ExcelWriter]] = None, sheet_name: str = 'Sheet1', start_row: int = 0, start_column: int = 0, null_text: str = 'NaN', include_header: bool = True, limit: Optional[int] = None, offset: int = 0, exclude_variables: Optional[List[str]] = None, include_variables: Optional[List[str]] = None, metadata_only: bool = False, apply_labels: bool = False, labels_as_categories: bool = True, missing_as_NaN: bool = False, convert_datetimes: bool = True, dates_as_datetime64: bool = False, **kwargs*)

Convert the SPSS data into an Excel file where each row represents a record of SPSS data.

Parameters

- **data** (Path-like filename, bytes or BytesIO) – The SPSS data to load. Accepts either a series of bytes or a filename.
- **target** (Path-like / BytesIO / ExcelWriter) – The destination where the Excel file should be stored. Accepts either a filename, file-pointer or a BytesIO, or an ExcelWriter instance.
- **sheet_name** (str) – The worksheet on which the SPSS data should be written. Defaults to 'Sheet1'.
- **start_row** (int) – The row number (starting at 0) where the SPSS data should begin. Defaults to 0.
- **start_column** (int) – The column number (starting at 0) where the SPSS data should begin. Defaults to 0.

- **null_text** (str) – The way that missing values should be represented in the Excel file. Defaults to '' (an empty string).
- **include_header** (bool) – If True, will include a header row with column labels. If False, will not include a header row. Defaults to True.
- **limit** (int or None) – The number of records to read from the data. If None will return all records. Defaults to None.
- **offset** (int) – The record at which to start reading the data. Defaults to 0 (first record).
- **exclude_variables** (iterable of str or None) – A list of the variables that should be ignored when reading data. Defaults to None.
- **include_variables** (iterable of str or None) – A list of the variables that should be explicitly included when reading data. Defaults to None.
- **metadata_only** (bool) – If True, will return no data records in the resulting DataFrame but will return a complete Metadata instance. Defaults to False.
- **apply_labels** (bool) – If True, converts the numerically-coded values in the raw data to their human-readable labels. Defaults to False.
- **labels_as_categories** (bool) – If True, will convert labeled or formatted values to Pandas categories. Defaults to True.

  Caution: This parameter will only have an effect if the apply_labels parameter is True.

- **missing_as_NaN** (bool) – If True, will return any missing values as NaN. Otherwise will return missing values as per the configuration of missing value representation stored in the underlying SPSS data. Defaults to False, which applies the missing value representation configured in the SPSS data itself.
- **convert_datetimes** (bool) – if True, will convert the native integer representation of datetime values in the SPSS data to Pythonic datetime, or date, etc. representations (or Pandas datetime64, depending on the dates_as_datetime64 parameter). If False, will leave the original integer representation. Defaults to True.
- **dates_as_datetime64** (bool) – If True, will return any date values as Pandas datetime64 types. Defaults to False.

  Caution: This parameter is only applied if convert_datetimes is set to True.

Returns None if target was not None, otherwise a BytesIO representation of the Excel file.

Return type None or str

3.1.4 to_json

to_json(*data: Union[os.PathLike[Any], _io.BytesIO, bytes], target: Optional[Union[os.PathLike[Any], _io.StringIO]] = None, layout: str = 'records', double_precision: int = 10, limit: Optional[int] = None, offset: int = 0, exclude_variables: Optional[List[str]] = None, include_variables: Optional[List[str]] = None, metadata_only: bool = False, apply_labels: bool = False, labels_as_categories: bool = True, missing_as_NaN: bool = False, convert_datetimes: bool = True, dates_as_datetime64: bool = False, **kwargs*)

Convert the SPSS `data` into a JSON string.

Parameters

- **data** (Path-like filename, `bytes` or `BytesIO`) – The SPSS data to load. Accepts either a series of bytes or a filename.
- **target** (Path-like / `StringIO` / `str` / `None`) – The destination where the JSON representation should be stored. Accepts either a filename, file-pointer or `StringIO`, or `None`. If `None`, will return a `str` object stored in-memory. Defaults to `None`.
- **layout** (`str`) – Indicates the layout schema to use for the JSON representation of the data. Accepts:
  - `records`, where the resulting JSON object represents an array of objects where each object corresponds to a single record, with key/value pairs for each column and that record's corresponding value
  - `table`, where the resulting JSON object contains a metadata (data map) describing the data schema along with the resulting collection of record objects

  Defaults to `records`.
- **double_precision** (class:*int <python:int>*) – Indicates the precision (places beyond the decimal point) to apply for floating point values. Defaults to `10`.
- **limit** (`int` or `None`) – The number of records to read from the data. If `None` will return all records. Defaults to `None`.
- **offset** (`int`) – The record at which to start reading the data. Defaults to 0 (first record).
- **exclude_variables** (iterable of `str` or `None`) – A list of the variables that should be ignored when reading data. Defaults to `None`.
- **include_variables** (iterable of `str` or `None`) – A list of the variables that should be explicitly included when reading data. Defaults to `None`.
- **metadata_only** (`bool`) – If `True`, will return no data records in the resulting `DataFrame` but will return a complete `Metadata` instance. Defaults to `False`.
- **apply_labels** (`bool`) – If `True`, converts the numerically-coded values in the raw data to their human-readable labels. Defaults to `False`.
- **labels_as_categories** (`bool`) – If `True`, will convert labeled or formatted values to Pandas categories. Defaults to `True`.

  Caution: This parameter will only have an effect if the `apply_labels` parameter is `True`.

- **missing_as_NaN** (`bool`) – If `True`, will return any missing values as `NaN`. Otherwise will return missing values as per the configuration of missing value representation stored in

the underlying SPSS data. Defaults to `False`, which applies the missing value representation configured in the SPSS data itself.

- **`convert_datetimes`** (`bool`) – if `True`, will convert the native integer representation of datetime values in the SPSS data to Pythonic `datetime`, or `date`, etc. representations (or Pandas `datetime64`, depending on the `dates_as_datetime64` parameter). If `False`, will leave the original integer representation. Defaults to `True`.
- **`dates_as_datetime64`** (`bool`) – If `True`, will return any date values as Pandas `datetime64` types. Defaults to `False`.

> **Caution:** This parameter is only applied if `convert_datetimes` is set to `True`.

Returns `None` if `target` was not `None`, otherwise a `str` representation of the JSON output.

Return type `None` or `str`

3.1.5 to_yaml

`to_yaml`(*data: Union[os.PathLike[Any], _io.BytesIO, bytes]*, *target: Optional[Union[os.PathLike[Any], _io.StringIO]] = None*, *layout: str = 'records'*, *double_precision: int = 10*, *limit: Optional[int] = None*, *offset: int = 0*, *exclude_variables: Optional[List[str]] = None*, *include_variables: Optional[List[str]] = None*, *metadata_only: bool = False*, *apply_labels: bool = False*, *labels_as_categories: bool = True*, *missing_as_NaN: bool = False*, *convert_datetimes: bool = True*, *dates_as_datetime64: bool = False*, ***kwargs*)

Convert the SPSS `data` into a YAML string.

Parameters

- **`data`** (Path-like filename, `bytes` or `BytesIO`) – The SPSS data to load. Accepts either a series of bytes or a filename.
- **`target`** (Path-like / `StringIO` / `str` / `None`) – The destination where the YAML representation should be stored. Accepts either a filename, file-pointer or `StringIO`, or `None`. If `None`, will return a `str` object stored in-memory. Defaults to `None`.
- **`layout`** (`str`) – Indicates the layout schema to use for the JSON representation of the data. Accepts:
  - `records`, where the resulting YAML object represents an array of objects where each object corresponds to a single record, with key/value pairs for each column and that record's corresponding value
  - `table`, where the resulting JSON object contains a metadata (data map) describing the data schema along with the resulting collection of record objects

  Defaults to `records`.
- **`double_precision`** (class:*int <python:int>*) – Indicates the precision (places beyond the decimal point) to apply for floating point values. Defaults to `10`.
- **`limit`** (`int` or `None`) – The number of records to read from the data. If `None` will return all records. Defaults to `None`.
- **`offset`** (`int`) – The record at which to start reading the data. Defaults to 0 (first record).

- **exclude_variables** (iterable of `str` or `None`) – A list of the variables that should be ignored when reading data. Defaults to `None`.
- **include_variables** (iterable of `str` or `None`) – A list of the variables that should be explicitly included when reading data. Defaults to `None`.
- **metadata_only** (`bool`) – If `True`, will return no data records in the resulting `DataFrame` but will return a complete `Metadata` instance. Defaults to `False`.
- **apply_labels** (`bool`) – If `True`, converts the numerically-coded values in the raw data to their human-readable labels. Defaults to `False`.
- **labels_as_categories** (`bool`) – If `True`, will convert labeled or formatted values to Pandas categories. Defaults to `True`.

  Caution: This parameter will only have an effect if the `apply_labels` parameter is `True`.

- **missing_as_NaN** (`bool`) – If `True`, will return any missing values as `NaN`. Otherwise will return missing values as per the configuration of missing value representation stored in the underlying SPSS data. Defaults to `False`, which applies the missing value representation configured in the SPSS data itself.
- **convert_datetimes** (`bool`) – if `True`, will convert the native integer representation of datetime values in the SPSS data to Pythonic `datetime`, or `date`, etc. representations (or Pandas `datetime64`, depending on the `dates_as_datetime64` parameter). If `False`, will leave the original integer representation. Defaults to `True`.
- **dates_as_datetime64** (`bool`) – If `True`, will return any date values as Pandas `datetime64` types. Defaults to `False`.

  Caution: This parameter is only applied if `convert_datetimes` is set to `True`.

Returns `None` if `target` was not `None`, otherwise a `str` representation of the YAML output.

Return type `None` or `str`

3.1.6 to_dict

to_dict(*data: Union[os.PathLike[Any], _io.BytesIO, bytes]*, *layout: str = 'records'*, *double_precision: int = 10*, *limit: Optional[int] = None*, *offset: int = 0*, *exclude_variables: Optional[List[str]] = None*, *include_variables: Optional[List[str]] = None*, *metadata_only: bool = False*, *apply_labels: bool = False*, *labels_as_categories: bool = True*, *missing_as_NaN: bool = False*, *convert_datetimes: bool = True*, *dates_as_datetime64: bool = False*, ***kwargs*)

Convert the SPSS `data` into a Python `dict`.

Parameters

- **data** (Path-like filename, `bytes` or `BytesIO`) – The SPSS data to load. Accepts either a series of bytes or a filename.
- **layout** (`str`) – Indicates the layout schema to use for the JSON representation of the data. Accepts:

  - `records`, where the resulting YAML object represents an array of objects where each object corresponds to a single record, with key/value pairs for each column and that record's corresponding value
  - `table`, where the resulting JSON object contains a metadata (data map) describing the data schema along with the resulting collection of record objects

  Defaults to `records`.

- **`double_precision`** (class:*int* <*python:int*>) – Indicates the precision (places beyond the decimal point) to apply for floating point values. Defaults to `10`.
- **`limit`** (`int` or `None`) – The number of records to read from the data. If `None` will return all records. Defaults to `None`.
- **`offset`** (`int`) – The record at which to start reading the data. Defaults to 0 (first record).
- **`exclude_variables`** (iterable of `str` or `None`) – A list of the variables that should be ignored when reading data. Defaults to `None`.
- **`include_variables`** (iterable of `str` or `None`) – A list of the variables that should be explicitly included when reading data. Defaults to `None`.
- **`metadata_only`** (`bool`) – If `True`, will return no data records in the resulting `DataFrame` but will return a complete `Metadata` instance. Defaults to `False`.
- **`apply_labels`** (`bool`) – If `True`, converts the numerically-coded values in the raw data to their human-readable labels. Defaults to `False`.
- **`labels_as_categories`** (`bool`) – If `True`, will convert labeled or formatted values to Pandas categories. Defaults to `True`.

  Caution: This parameter will only have an effect if the `apply_labels` parameter is `True`.

- **`missing_as_NaN`** (`bool`) – If `True`, will return any missing values as `NaN`. Otherwise will return missing values as per the configuration of missing value representation stored in the underlying SPSS data. Defaults to `False`, which applies the missing value representation configured in the SPSS data itself.
- **`convert_datetimes`** (`bool`) – if `True`, will convert the native integer representation of datetime values in the SPSS data to Pythonic `datetime`, or `date`, etc. representations (or Pandas `datetime64`, depending on the `dates_as_datetime64` parameter). If `False`, will leave the original integer representation. Defaults to `True`.
- **`dates_as_datetime64`** (`bool`) – If `True`, will return any date values as Pandas `datetime64` types. Defaults to `False`.

  Caution: This parameter is only applied if `convert_datetimes` is set to `True`.

Returns `None` if `target` was not `None`, otherwise a `list` of `dict` if `layout` is `records`, or a `dict` if `layout` is `table`.

Return type `None` or `str`

3.1.7 get_metadata

get_metadata(*data*)

Retrieve the metadata that describes the coded representation of the data, corresponding formatting information, and their related human-readable labels.

Parameters **data** (Path-like filename, `bytes` or `BytesIO`) – The SPSS data to load. Accepts either a series of bytes or a filename.

Returns The metadata that describes the raw data and its corresponding labels.

Return type `Metadata`

3.2 Writing Data to SPSS

3.2.1 from_dataframe

from_dataframe(*df: pandas.core.frame.DataFrame*, *target: Optional[Union[PathLike[Any], _io.BytesIO]] = None*, *metadata: Optional[spss_converter.Metadata.Metadata] = None*, *compress: bool = False*)

Create an SPSS dataset from a Pandas `DataFrame`.

Parameters

- **df** (`pandas.DataFrame`) – The `DataFrame` to serialize to an SPSS dataset.
- **target** (Path-like / `BytesIO` / `None`) – The target to which the SPSS dataset should be written. Accepts either a filename/path, a `BytesIO` object, or `None`. If `None` will return a `BytesIO` object containing the SPSS dataset. Defaults to `None`.
- **metadata** (`Metadata` / `None`) – The `Metadata` associated with the dataset. If `None`, will attempt to derive it form `df`. Defaults to `None`.
- **compress** (`bool`) – If `True`, will return data in the compressed ZSAV format. If `False`, will return data in the standards SAV format. Defaults to `False`.

Returns A `BytesIO` object containing the SPSS data if `target` is `None` or not a filename, otherwise `None`

Return type `BytesIO` or `None`

Raises

- **ValueError** – if `df` is not a `pandas.DataFrame`
- **ValueError** – if `metadata` is not a `Metadata`

3.2.2 from_csv

from_csv(*as_csv: Union[str, PathLike[Any], _io.BytesIO]*, *target: Optional[Union[PathLike[Any], _io.BytesIO]] = None*, *compress: bool = False*, *delimiter='|'*, ***kwargs*)

Convert a CSV file into an SPSS dataset.

Tip: If you pass any additional keyword arguments, those keyword arguments will be passed onto the `pandas.read_csv()` function.

Parameters

- **as_csv** (`str` / File-location / `BytesIO`) – The CSV data that you wish to convert into an SPSS dataset.
- **target** (Path-like / `BytesIO` / `None`) – The target to which the SPSS dataset should be written. Accepts either a filename/path, a `BytesIO` object, or `None`. If `None` will return a `BytesIO` object containing the SPSS dataset. Defaults to `None`.
- **compress** (`bool`) – If `True`, will return data in the compressed ZSAV format. If `False`, will return data in the standards SAV format. Defaults to `False`.
- **delimiter** (`str`) – The delimiter used between columns. Defaults to `|`.
- **kwargs** (`dict`) – Additional keyword arguments which will be passed onto the `pandas.read_csv()` function.

Returns A `BytesIO` object containing the SPSS data if `target` is `None` or not a filename, otherwise `None`

Return type `BytesIO` or `None`

3.2.3 from_excel

from_excel(*as_excel*, *target: Optional[Union[PathLike[Any], _io.BytesIO]] = None*, *compress: bool = False*, ***kwargs*)

Convert Excel data into an SPSS dataset.

Tip: If you pass any additional keyword arguments, those keyword arguments will be passed onto the `pandas.read_excel()` function.

Parameters

- **as_excel** (`str` / File-location / `BytesIO` / `bytes` / `ExcelFile`) – The Excel data that you wish to convert into an SPSS dataset.
- **target** (Path-like / `BytesIO` / `None`) – The target to which the SPSS dataset should be written. Accepts either a filename/path, a `BytesIO` object, or `None`. If `None` will return a `BytesIO` object containing the SPSS dataset. Defaults to `None`.
- **compress** (`bool`) – If `True`, will return data in the compressed ZSAV format. If `False`, will return data in the standards SAV format. Defaults to `False`.
- **kwargs** (`dict`) – Additional keyword arguments which will be passed onto the `pandas.read_excel()` function.

Returns A `BytesIO` object containing the SPSS data if `target` is `None` or not a filename, otherwise `None`

Return type `BytesIO` or `None`

3.2.4 from_json

from_json(*as_json: Union[str, PathLike[Any], _io.BytesIO]*, *target: Optional[Union[PathLike[Any], _io.BytesIO]] = None*, *compress: bool = False*, ***kwargs*)

Convert JSON data into an SPSS dataset.

Tip: If you pass any additional keyword arguments, those keyword arguments will be passed onto the `pandas.read_json()` function.

Parameters

- **as_json** (`str` / File-location / `BytesIO`) – The JSON data that you wish to convert into an SPSS dataset.
- **target** (Path-like / `BytesIO` / `None`) – The target to which the SPSS dataset should be written. Accepts either a filename/path, a `BytesIO` object, or `None`. If `None` will return a `BytesIO` object containing the SPSS dataset. Defaults to `None`.
- **compress** (`bool`) – If `True`, will return data in the compressed ZSAV format. If `False`, will return data in the standards SAV format. Defaults to `False`.
- **kwargs** (`dict`) – Additional keyword arguments which will be passed onto the `pandas.read_json()` function.

Returns A `BytesIO` object containing the SPSS data if `target` is `None` or not a filename, otherwise `None`

Return type `BytesIO` or `None`

3.2.5 from_yaml

from_yaml(*as_yaml: Union[str, PathLike[Any], _io.BytesIO]*, *target: Optional[Union[PathLike[Any], _io.BytesIO]] = None*, *compress: bool = False*, ***kwargs*)

Convert YAML data into an SPSS dataset.

Tip: If you pass any additional keyword arguments, those keyword arguments will be passed onto the `DataFrame.from_dict()` method.

Parameters

- **as_yaml** (`str` / File-location / `BytesIO`) – The YAML data that you wish to convert into an SPSS dataset.

- **target** (Path-like / BytesIO / None) – The target to which the SPSS dataset should be written. Accepts either a filename/path, a BytesIO object, or None. If None will return a BytesIO object containing the SPSS dataset. Defaults to None.
- **compress** (bool) – If True, will return data in the compressed ZSAV format. If False, will return data in the standards SAV format. Defaults to False.
- **kwargs** (dict) – Additional keyword arguments which will be passed onto the DataFrame.from_dict() method.

Returns A BytesIO object containing the SPSS data if target is None or not a filename, otherwise None

Return type BytesIO or None

3.2.6 from_dict

from_dict(*as_dict: dict*, *target: Optional[Union[PathLike[Any], _io.BytesIO]] = None*, *compress: bool = False*, ***kwargs*)

Convert a dict object into an SPSS dataset.

Tip: If you pass any additional keyword arguments, those keyword arguments will be passed onto the DataFrame.from_dict() method.

Parameters

- **as_dict** (dict) – The dict data that you wish to convert into an SPSS dataset.
- **target** (Path-like / BytesIO / None) – The target to which the SPSS dataset should be written. Accepts either a filename/path, a BytesIO object, or None. If None will return a BytesIO object containing the SPSS dataset. Defaults to None.
- **compress** (bool) – If True, will return data in the compressed ZSAV format. If False, will return data in the standards SAV format. Defaults to False.
- **kwargs** (dict) – Additional keyword arguments which will be passed onto the DataFrame.from_dict() method.

Returns A BytesIO object containing the SPSS data if target is None or not a filename, otherwise None

Return type BytesIO or None

3.2.7 apply_metadata

apply_metadata(*df: pandas.core.frame.DataFrame*, *metadata: Union[spss_converter.Metadata.Metadata, dict, pyreadstat._readstat_parser.metadata_container]*, *as_category: bool = True*)

Updates the `DataFrame` `df` based on the `metadata`.

Parameters

- **df** (`pandas.DataFrame`) – The `DataFrame` to update.
- **metadata** (`Metadata`, `pyreadstat.metadata_container`, or compatible `dict`) – The `Metadata` to apply to `df`.
- **as_category** (`bool`) – if `True`, will variables with formats will be transformed into categories in the `DataFrame`. Defaults to `True`.

Returns A copy of `df` updated to reflect `metadata`.

Return type `DataFrame`

3.3 Utility Classes

3.3.1 Metadata

class Metadata(***kwargs*)

Object representation of *metadata* retrieved from an SPSS file.

classmethod from_dict(*as_dict: dict*)

Create a *Metadata* instance from a `dict` representation.

Parameters **as_dict** (`dict`) – A `dict` representation of the *Metadata*.

Returns A *Metadata* instance

Return type *Metadata*

classmethod from_pyreadstat(*as_metadata*)

Create a *Metadata* instance from a Pyreadstat metadata object.

Parameters **as_metadata** (`Pyreadstat.metadata_container`) – The Pyreadstat metadata object from which the *Metadata* instance should be created.

Returns The *Metadata* instance.

Return type *Metadata*

to_dict() → dict

Return a `dict` representation of the instance.

Return type `dict`

to_pyreadstat()

Create a Pyreadstat metadata representation of the *Metadata* instance.

Returns

The Pyreadstat metadata.

Return type `metadata_container <pyreadstat:_readstat_parser.metadata_container`

property column_metadata
Collection of metadata that describes each column or variable within the dataset.

Returns A `dict` where the key is the name of the column/variable and the value is a *`ColumnMetadata`* object or compatible `dict`.

Return type `dict` / `None`

property columns
The number of columns/variables in the dataset.

Return type `int`

property file_encoding
The file encoding for the dataset.

Return type `str` or `None`

property file_label
The file label.

Note: This property is irrelevant for SPSS, but is relevant for SAS data.

Return type `str` / `None`

property notes
Set of notes related to the file.

Return type `str` / `None`

property rows
The number of cases or rows in the dataset.

Return type `int`

property table_name
The name of the data table.

Return type `str` / `None`

3.3.2 ColumnMetadata

class ColumnMetadata(***kwargs*)
Object representation of the *metadata* that describes a column or variable form an SPSS file.

add_to_pyreadstat(*pyreadstat*)
Update `pyreadstat` to include the metadata for this column/variable.

Parameters **pyreadstat** (`metadata_container <pyreadstat:_readstat_parser.metadata_container`) – The Pyreadstat metadata object where the *`ColumnMetadata`* data should be updated.

Returns

The Pyreadstat metadata.

Return type `metadata_container <pyreadstat:_readstat_parser.metadata_container`

classmethod from_dict(*as_dict: dict*)
Create a new *ColumnMetadata* instance from a `dict` representation.

Parameters **as_dict** (`dict`) – The `dict` representation of the *ColumnMetadata*.

Returns The *ColumnMetadata* instance.

Return type *ColumnMetadata*

classmethod from_pyreadstat_metadata(*name: str*, *as_metadata*)
Create a new *ColumnMetadata* instance from a Pyreadstat metadata object.

Parameters

- **name** (`str`) – The name of the variable for which a *ColumnMetadata* instance should be created.
- **as_metadata** (`Pyreadstat.metadata_container`) – The Pyreadstat metadata object from which the column's metadata should be extracted.

Returns The *ColumnMetadata* instance.

Return type *ColumnMetadata*

to_dict() → dict
Generate a `dict` representation of the instance.

Return type `dict`

property alignment
The alignment to apply to values from this column/variable when displaying data. Defaults to `'unknown'`.

Accepts either `'unknown'`, `'left'`, `'center'`, or `'right'` as either a case-insensitive `str` or a `VariableAlignmentEnum`.

Return type `VariableAlignmentEnum`

property display_width
The maximum width at which the value is displayed. Defaults to 0.

Return type `int`

property label
The label applied ot the column/variable.

Return type `str` / `None`

property measure
A classification of the type of measure (or value type) represented by the variable. Defaults to `'unknown'`.

Accepts either `'unknown'`, `'nominal'`, `'ordinal'`, or `'scale'`.

Return type `VariableMeasureEnum`

property missing_range_metadata
Collection of meta data that defines the numerical ranges that are to be considered missing in the underlying data.

Returns `list` of `dict` with keys `'low'` and `'high'` for the low/high values of the range to apply when raw values are missing (`None`).

Return type `list` of `dict` or `None`

property missing_value_metadata
Value used to represent misisng values in the raw data. Defaults to `None`.

> **Note:** This is not actually relevant for SPSS data, but is an artifact for SAS and SATA data.

Return type `list` of `int` or `str` / `None`

property name
The name of the column/variable.

Return type `str` / `None`

property storage_width
The width of data to store in the data file for the value. Defaults to 0.

Rytpe `int`

property value_metadata
Collection of values possible for the column/variable, with corresponding labels for each value.

Returns `dict` whose keys are the values in the raw data and whose values are the labels for each value. May be `None` for variables whose value is not coded.

Return type `dict` / `None`

CHAPTER

FOUR

ERROR REFERENCE

4.1 Handling Errors

4.1.1 Stack Traces

Because the **SPSS Converter** produces exceptions which inherit from the standard library, it leverages the same API for handling stack trace information. This means that it will be handled just like a normal exception in unit test frameworks, logging solutions, and other tools that might need that information.

4.2 SPSS Converter Errors

4.2.1 SPSSConverterError (from `ValueError`)

class SPSSConverterError
Base exception raised by the **SPSS Converter** library.

4.2.2 ColumnNameNotFoundError (from `SPSSConverterError`)

class ColumnNameNotFoundError
Exception raised when a given column or variable name is not found.

4.2.3 InvalidDataFormatError (from `SPSSConverterError`)

class InvalidDataFormatError
Exception raised when a value did not conform to an expected type.

4.2.4 InvalidLayoutError (from `SPSSConverterError`)

class InvalidLayoutError
Exception raised when a layout value was not recognized.

CHAPTER

FIVE

CONTRIBUTING TO THE SPSS CONVERTER

Note: As a general rule of thumb, **SPSS Converter** applies **PEP 8** styling, with some important differences.

Branch	Unit Tests
latest	

- – v.0.1
  –

What makes an API idiomatic?

One of my favorite ways of thinking about idiomatic design comes from a talk given by Luciano Ramalho at Pycon 2016[5] where he listed traits of a Pythonic API as being:

- don't force [the user] to write boilerplate code
- provide ready to use functions and objects
- don't force [the user] to subclass unless there's a *very good* reason
- include the batteries: make easy tasks easy
- are simple to use but not simplistic: make hard tasks possible
- leverage the Python data model to:
  - provide objects that behave as you expect
  - avoid boilerplate through introspection (reflection) and metaprogramming.

Contents:

[5] https://www.youtube.com/watch?v=k55d3ZUF3ZQ

5.1 Design Philosophy

SPSS Converter is meant to be a "beautiful" and "usable" library. That means that it should offer an idiomatic API that:

- works out of the box as intended,
- minimizes "bootstrapping" to produce meaningful output, and
- does not force users to understand how it does what it does.

In other words:

> Users should simply be able to drive the car without looking at the engine.

5.2 Style Guide

5.2.1 Basic Conventions

- Do not terminate lines with semicolons.
- Line length should have a maximum of *approximately* 90 characters. If in doubt, make a longer line or break the line between clear concepts.
- Each class should be contained in its own file.
- If a file runs longer than 2,000 lines... it should probably be refactored and split.
- All imports should occur at the top of the file.
- Do not use single-line conditions:

```
# GOOD
if x:
  do_something()

# BAD
if x: do_something()
```

- When testing if an object has a value, be sure to use `if x is None:` or `if x is not None`. Do **not** confuse this with `if x:` and `if not x:`.
- Use the `if x:` construction for testing truthiness, and `if not x:` for testing falsiness. This is **different** from testing:
  - `if x is True:`
  - `if x is False:`
  - `if x is None:`
- As of right now, because we feel that it negatively impacts readability and is less-widely used in the community, we are **not** using type annotations.

5.2.2 Naming Conventions

- `variable_name` and not `variableName` or `VariableName`. Should be a noun that describes what information is contained in the variable. If a `bool`, preface with `is_` or `has_` or similar question-word that can be answered with a yes-or-no.
- `function_name` and not `function_name` or `functionName`. Should be an imperative that describes what the function does (e.g. `get_next_page`).
- `CONSTANT_NAME` and not `constant_name` or `ConstantName`.
- `ClassName` and not `class_name` or `Class_Name`.

5.2.3 Design Conventions

- Functions at the module level can only be aware of objects either at a higher scope or singletons (which effectively have a higher scope).
- Functions and methods can use **one** positional argument (other than `self` or `cls`) without a default value. Any other arguments must be keyword arguments with default value given.

```
def do_some_function(argument):
  # rest of function...

def do_some_function(first_arg,
                     second_arg = None,
                     third_arg = True):
  # rest of function ...
```

- Functions and methods that accept values should start by validating their input, throwing exceptions as appropriate.
- When defining a class, define all attributes in `__init__`.
- When defining a class, start by defining its attributes and methods as private using a single-underscore prefix. Then, only once they're implemented, decide if they should be public.

- Don't be afraid of the private attribute/public property/public setter pattern:

```
class SomeClass(object):
  def __init__(*args, **kwargs):
    self._private_attribute = None

  @property
  def private_attribute(self):
    # custom logic which  may override the default return

    return self._private_attribute

  @setter.private_attribute
  def private_attribute(self, value):
    # custom logic that creates modified_value

    self._private_attribute = modified_value
```

- Separate a function or method's final (or default) `return` from the rest of the code with a blank line (except for single-line functions/methods).

5.2.4 Documentation Conventions

We are very big believers in documentation (maybe you can tell). To document **SPSS Converter** we rely on several tools:

Sphinx[1]

Sphinx[1] is used to organize the library's documentation into this lovely readable format (which is also published to ReadTheDocs[2]). This documentation is written in reStructuredText[3] files which are stored in `<project>/docs`.

Tip: As a general rule of thumb, we try to apply the ReadTheDocs[2] own Documentation Style Guide[4] to our RST documentation.

Hint: To build the HTML documentation locally:

1. In a terminal, navigate to `<project>/docs`.
2. Execute `make html`.

When built locally, the HTML output of the documentation will be available at `./docs/_build/index.html`.

---

[1] http://sphinx-doc.org
[2] https://readthedocs.org
[3] http://www.sphinx-doc.org/en/stable/rest.html
[4] http://documentation-style-guide-sphinx.readthedocs.io/en/latest/style-guide.html

Docstrings

- Docstrings are used to document the actual source code itself. When writing docstrings we adhere to the conventions outlined in **PEP 257**.

5.3 Dependencies

Python 3.x
* Pandas v0.24 or higher * Pyreadstat v1.0 or higher * OpenPyXL v.3.0.7 or higher * PyYAML v3.10 or higher * simplejson v3.0 or higher * Validator-Collection v1.5.0 or higher

5.4 Preparing Your Development Environment

In order to prepare your local development environment, you should:

1. Fork the Git repository.
2. Clone your forked repository.
3. Set up a virtual environment (optional).
4. Install dependencies:

```
spss-converter/ $ pip install -r requirements.txt
```

And you should be good to go!

5.5 Ideas and Feature Requests

Check for open issues or create a new issue to start a discussion around a bug or feature idea.

5.6 Testing

If you've added a new feature, we recommend you:

- create local unit tests to verify that your feature works as expected, and
- run local unit tests before you submit the pull request to make sure nothing else got broken by accident.

See also:

For more information about the **SPSS Converter** testing approach please see: *Testing SPSS Converter*

5.7 Submitting Pull Requests

After you have made changes that you think are ready to be included in the main library, submit a pull request on Github and one of our developers will review your changes. If they're ready (meaning they're well documented, pass unit tests, etc.) then they'll be merged back into the main repository and slated for inclusion in the next release.

5.8 Building Documentation

In order to build documentation locally, you can do so from the command line using:

```
spss-converter/ $ cd docs
spss-converter/docs $ make html
```

When the build process has finished, the HTML documentation will be locally available at:

```
spss-converter/docs/_build/html/index.html
```

Note: Built documentation (the HTML) is **not** included in the project's Git repository. If you need local documentation, you'll need to build it.

5.9 Contributors

Thanks to everyone who helps make **SPSS Converter** useful:

- Chris Modzelewski (@insightindustry)

5.10 References

CHAPTER

SIX

TESTING THE SPSS CONVERTER

Contents

6.1 Testing Philosophy

Note: Unit tests for the **SPSS Converter** are written using pytest[1] and a comprehensive set of test automation are provided by tox[2].

There are many schools of thought when it comes to test design. When building **SPSS Converter**, we decided to focus on practicality. That means:

- **DRY is good, KISS is better.** To avoid repetition, our test suite makes extensive use of fixtures, parametrization, and decorator-driven behavior. This minimizes the number of test functions that are nearly-identical. However, there are certain elements of code that are repeated in almost all test functions, as doing so will make future readability and maintenance of the test suite easier.
- **Coverage matters...kind of.** We have documented the primary intended behavior of every function in the **SPSS Converter** library, and the most-likely failure modes that can be expected. At the time of writing, we have about 85% code coverage. Yes, yes: We know that is less than 100%. But there are edge cases which are

[1] https://docs.pytest.org/en/latest/
[2] https://tox.readthedocs.io

almost impossible to bring about, based on confluences of factors in the wide world. Our goal is to test the key functionality, and as bugs are uncovered to add to the test functions as necessary.

6.2 Test Organization

Each individual test module (e.g. `test_read.py`) corresponds to a conceptual grouping of functionality. For example:

- `test_read.py` tests functions that de-serialize data from SPSS files, as defined in `spss_converter/read.py`

Certain test modules are tightly coupled, as the behavior in one test module may have implications on the execution of tests in another. These test modules use a numbering convention to ensure that they are executed in their required order, so that `test_1_NAME.py` is always executed before `test_2_NAME.py`.

6.3 Configuring & Running Tests

6.3.1 Installing with the Test Suite

Installing via pip

From Local Development Environment

```
$ pip install spss-converter[dev]
```

See also:

When you *create a local development environment*, all dependencies for running and extending the test suite are installed.

6.3.2 Command-line Options

The **SPSS Converter** does not use any custom command-line options in its test suite.

Tip: For a full list of the CLI options, including the defaults available, try:

```
spss-converter $ cd tests/
spss-converter/tests/ $ pytest --help
```

6.3.3 Configuration File

Because the **SPSS Converter** has a very simple test suite, we have not prepared a `pytest.ini` configuration file.

6.3.4 Running Tests

Entire Test Suite

Test Module

Test Function

```
tests/ $ pytest
```

```
tests/ $ pytest tests/test_module.py
```

```
tests/ $ pytest tests/test_module.py -k 'test_my_test_function'
```

6.4 Skipping Tests

Note: Because of the simplicity of the **SPSS Converter**, the test suite does not currently support any test skipping.

6.5 Incremental Tests

Note: The **SPSS Converter** test suite does support incremental testing using, however at the moment none of the tests designed rely on this functionality.

A variety of test functions are designed to test related functionality. As a result, they are designed to execute incrementally. In order to execute tests incrementally, they need to be defined as methods within a class that you decorate with the `@pytest.mark.incremental` decorator as shown below:

```
@pytest.mark.incremental
class TestIncremental(object):
    def test_function1(self):
        pass
    def test_modification(self):
        assert 0
    def test_modification2(self):
        pass
```

This class will execute the `TestIncremental.test_function1()` test, execute and fail on the `TestIncremental.test_modification()` test, and automatically fail `TestIncremental.test_modification2()` because of the `.test_modification()` failure.

To pass state between incremental tests, add a `state` argument to their method definitions. For example:

```
@pytest.mark.incremental
class TestIncremental(object):
    def test_function(self, state):
        state.is_logged_in = True
        assert state.is_logged_in = True
    def test_modification1(self, state):
        assert state.is_logged_in is True
```

(continues on next page)

(continued from previous page)

```
        state.is_logged_in = False
        assert state.is_logged_in is False
    def test_modification2(self, state):
        assert state.is_logged_in is True
```

Given the example above, the third test (`test_modification2`) will fail because `test_modification` updated the value of `state.is_logged_in`.

Note: `state` is instantiated at the level of the entire test session (one run of the test suite). As a result, it can be affected by tests in other test modules.

CHAPTER

SEVEN

RELEASE HISTORY

Contributors

- Chris Modzelewski (@insightindustry)

Contents

7.1 Release 0.1.1

- Adjusted encoding of requirements files.

7.2 Release 0.1.0

- First public release

CHAPTER

EIGHT

GLOSSARY

Metadata A collection of information that allows a human being to understand what raw data represents. Think of it as a "data map" that tells you a) what to expect within the raw data stored in a given format, b) what that data actually means / signifies.

See also:

- *Metadata*
- *ColumnMetadata*

Multiple Response Set A way of representing "select all answers that apply" survey questions in SPSS data, where each answer maps to its own variable/column in the raw data, but the set of variables/columns should be grouped within the multiple response set.

> **Warning:** Because Pyreadstat does not yet support Multiple Response Sets, the **SPSS Converter** also does not support them.

CHAPTER

NINE

SPSS CONVERTER LICENSE

The **SPSS Converter** is a simple utility that facilitates the easy conversion of SPSS data to / from a variety of formats, including:

- CSV
- JSON
- YAML
- Excel
- Pandas `DataFrame`

Contents

CHAPTER

TEN

INSTALLATION

To install the **SPSS Converter**, just execute:

```
$ pip install spss-converter
```

10.1 Dependencies

Python 3.x

* Pandas v0.24 or higher
* Pyreadstat v1.0 or higher
* OpenPyXL v.3.0.7 or higher
* PyYAML v3.10 or higher
* simplejson v3.0 or higher
* Validator-Collection v1.5.0 or higher

CHAPTER

ELEVEN

WHY THE SPSS CONVERTER?

If you work with SPSS data in the Python ecosystem, you probably use a combination of two or three key libraries: Pandas, Pyreadstat, and savReaderWriter. All three libraries are vital tools, incredibly well-constructed, designed, and managed. But over the years, I have found that converting from SPSS to other file formats using these libraries requires some fairly repetitive boilerplate code. So why not make it easier?

The **SPSS Converter** library is a simple wrapper around the Pyreadstat and Pandas libraries that provides a clean and simple API for reading data files in a variety of formats and converting them to a variety of formats. The semantics are super simple, and should be as simple as: `spss_converter.to_csv('my-spss-file.sav')` or `spss_converter.from_json('my-json-file.json')`.

11.1 Key SPSS Converter Features

- With one function call, convert an SPSS file into:
  - a Pandas `DataFrame`
  - CSV
  - JSON
  - YAML
  - Excel
  - a `dict`
- With one function call, create an SPSS data file from data in:
  - a Pandas `DataFrame`
  - CSV
  - JSON
  - YAML
  - Excel
  - a `dict`
- With one function call, generate a Pythonic data map or meta-data collection from your SPSS data file.
- Decide which variables (columns) you want to include / exclude when doing your conversion.

11.2 SPSS Converter vs Alternatives

The **SPSS Converter** library is a simple wrapper around the Pyreadstat and Pandas libraries that simplifies the syntax for converting between different file formats.

While I am (I think understandably) biased in favor of the **SPSS Converter**, there some perfectly reasonable alternatives:

Pyreadstat + Pandas

savReaderWriter

Obviously, since the **SPSS Converter** is just a wrapper around Pyreadstat and Pandas, you can simply call their functions directly.

Both libraries are excellent, stable, and use fairly straightforward syntax. However:

- using those libraries directly does double the number of function calls you need to make to convert between different data formats, and
- those libraries (and Pyreadstat in particular) provide limited validation or Pythonic object representation (less "batteries included" in its syntactical approach).

Of course, these differences are largely stylistic in nature.

Tip: When to use it?

Honestly, since initially building this wrapper I rarely use Pyreadstat and Pandas directly. Mostly, this is a matter of syntactical taste and personal preference.

However, I would definitely look to those libraries directly if I were:

- writing this kind of wrapper
- working in older versions of Python (< 3.7)
- working with other formats of data than SPSS

Using SPSS Converter

Using Pyreadstat and Pandas

```
from spss_converter import read, write

# SPSS File to CSV
read.to_csv('my-spss-file.sav',
            target = 'my-csv-file.csv')

# CSV to SPSS File
write.from_csv('my-csv-file.csv',
               target = 'my-spss-file.sav')

# SPSS File to Excel file
read.to_excel('my-spss-file.sav',
              target = 'my-excel-file.xlsx')

# Excel to SPSS file
write.from_excel('my-excel-file.xlsx',
                 target = 'my-spss-file.sav')

# ... similar pattern for other formats
```

```
import pyreadstat
import pandas

# SPSS File to CSV
df, metadata = pyreadstat.read_sav('my-spss-file.sav')
csv_file = df.to_csv('my-csv-file.csv')

# CSV to SPSS file
df = pandas.read_csv('my-csv-file.csv')
spss_file = pyreadstat.write_sav(df,
                                 'my-spss-file.sav')

# SPSS File to Excel File
df, metadata = pyreadstat.read_sav('my-spss-file.sav')
excel_file = df.to_excel('my-excel-file.xlsx')

# Excel file to SPSS file
df = pandas.read_excel('my-excel-file.xlsx')
spss_file = pyreadstat.write_sav(df,
                                 'my-spss-file.sav')

# .. similar pattern for other formats
```

The savReaderWriter library is a powerful library for converting SPSS data to/from different formats. Its core strength is its ability to get very granular metadata about the SPSS data and to sequentially iterate through its records.

However, the library has three significant limitations when it comes to format conversion:

- The library only provides read and write access for SPSS data, and this means that you would have to write the actual "conversion" logic yourself. This can get quite complicated, particularly when dealing with data serialization challenges.
- The library depends on the SPSS I/O module, which is packaged with the library. This module has both licensing implications and is a "heavy" module for distribution.
- The library's most-recent commits date back to 2017, and it would seem that it is no longer being actively maintained.

Tip: When to use it?

- When you actually need to dive into the data at the level of particular cases or values.
- When your data has *Multiple Response Sets*, which are not (yet) supported by either Pyreadstat or the **SPSS Converter**.

CHAPTER

TWELVE

QUESTIONS AND ISSUES

You can ask questions and report issues on the project's Github Issues Page

CHAPTER

THIRTEEN

CONTRIBUTING

We welcome contributions and pull requests! For more information, please see the *Contributor Guide*.

CHAPTER

FOURTEEN

TESTING

We use TravisCI for our build automation, Codecov.io for our test coverage, and ReadTheDocs for our documentation. Detailed information about our test suite and how to run tests locally can be found in our *Testing Reference*.

CHAPTER

FIFTEEN

LICENSE

The **SPSS Converter** is made available under an *MIT License*.

PYTHON MODULE INDEX

s

t

INDEX

T

V