oemof.tabular

Release 0.0.4dev

Feb 15, 2023

Contents

CHAPTER 1

Overview

Load oemof energy systems from tabular data sources.

- Free software: BSD 3-Clause License

1.1 Installation

Simpy run:

```
pip install oemof.tabular
```

1.2 Documentation

https://oemof-tabular.readthedocs.io/

1.3 Development

Please activate pre-commit hooks in order to follow our coding styles:

```
pip install pre-commit
pre-commit install
```

To run the all tests run:

```
pytest
```

CHAPTER 2

Installation

At the command line:

```
pip install oemof.tabular
```

CHAPTER 3

Usage

To use oemof.tabular in a project:

```
import oemof.tabular
```

3.1 Background

The underlying concept of **oemof-tabular** is the oemof solph package. The Open Energy Modelling Framework (oemof) is based on a graph structure at its core. In addition it provides an optimization model generator to construct individual dispatch and investment models. The internal logic, used terminology and software architecture is abstract and rather designed for model developers and experienced modellers.

Oemof users / developers can model energy systems with different degrees of freedom:

1. Modelling based using existing classes
2. Add own classes
3. Add own constraints based on the underlying algebraic modelling library

However, in some cases complexity of this internal logic and full functionality is neither necessary nor suitable for model users. Therefore we provide so called **facade classes** that provide an energy specific and reduced access to the underlying oemof.solph functionality. More importantly theses classes provide an interface to tabular data sources from that models can be created easily.

Note: To see the implemented facades check out the *facades* module.

3.1.1 Facades

Modelling energy systems based on these classes is straightforward. Parametrization of an energy system can either be done via python scripting or by using the datapackage structure described below. The documentation for the facades

can be found *facades*. In addition you can check out the jupyter notebook from the tutorials and the examples directory.

Currently we provide the following facades:

- `Dispatchable`
- `Volatile`
- `Storage`
- `Reservoir`
- `BackpressureTurbine`
- `ExtractionTurbine`
- `Commodity`
- `Conversion`
- `Load`.
- `Link`
- `Excess`

These can be mixed with all oemof solph classes if your are scripting.

3.1.2 Datamodel and Naming Conventions

Facades require specific attributes. For all facades the attribute *carrier*, 'tech' and 'type' need to be set. The type of the attribute is string, therefore you can choose string for these. However, if you want to leverage full postprocessing functionality we recommend using one of the types listed below

Carriers

- solar, wind, biomass, coal, lignite, uranium, oil, gas, hydro, waste, electricity, heat, other

Tech types

- st, ocgt, ccgt, ce, pv, onshore, offshore, ror, rsv, phs, ext, bp, battery

We recommend use the following naming convention for your facade names *bus-carrier-tech-number*. For example: *DE-gas-ocgt-1*. This allows you to also take advantage of the color map from *facades* module.

```
from oemof.facades import TECH_COLOR_MAP, CARRIER_COLER_MAP

biomass_color = CARRIER_COLER_MAP["biomass"]
pv_color = TECH_COLOR_MAP["pv"]
```

3.2 Datapackage

To construct a model based on the datapackage the following 2 steps are required:

1. Add the topology of the energy system based on the components and their **exogenous model variables** to csv-files in the datapackage format.
2. Create a python script to construct the energy system and the model from that data.

We recommend a specific workflow to allow to publish your scenario (input data, assumptions, model and results) altogether in one consistent block based on the datapackage standard (see: Reproducible Workflows).

3.2.1 How to create a Datapackage

We adhere to the frictionless (tabular) datapackage standard. On top of that structure we add our own logic. We require at least two things:

1. A directory named *data* containing at least one sub-folder called *elements* (optionally it may contain a directory *sequences*, *geometries* and/or *constraints*. Of course you may add any other directory, data or other information.)
2. A valid meta-data *.json* file for the datapackage

Note: You **MUST** provide one file with the buses called *bus.csv*!

The resulting tree of the datapackage could for example look like this:

```
|-- datapackage
    |-- data
        |-- elements
            |-- demand.csv
            |-- generator.csv
            |-- storage.csv
            |-- bus.csv
        |-- sequences
    |-- scripts
    |-- datapackage.json
```

Inside the datapackage, data is stored in so called resources. For a tabular-datapackage, these resources are CSV files. Columns of such resources are referred to as *fields*. In this sense field names of the resources are equivalent to parameters of the energy system elements and sequences.

To distinguish elements and sequences these two are stored in sub-directories of the data directory. In addition, geometrical information can be stored under *data/geometries* in a *.geojson* format. An optional subdirectory *data/constraints* can hold data describing global constraints. To simplifiy the process of creating and processing a datapackage you may also use the funtionalities of the `datapackage`

You can use functions to read and write resources (pandas.DataFrames in python). This can also be done for sequences and geometries.

```
from oemof.tabular.datapackage import building
...

building.read_elements('volatile.csv')

# manipulate data ...

building.write_elements('volatile.csv')
```

To create meta-data *json* file you can use the following code:

```
from datapackage_utilities import building

building.infer_metadata(
        package_name="my-datapackage",
        foreign_keys={
                        "bus": [
                                "volatile",
                                "dispatchable",
```

(continues on next page)

(continued from previous page)

```
                                "storage",
                                "heat_storage",
                                "load",
                                "ror",
                                "reservoir",
                                "phs",
                                "excess",
                                "boiler",
                                "commodity",
                            ],
                            "profile": ["load", "volatile", "heat_load", "ror", "reservoir
↪"],
                            "from_to_bus": ["link", "conversion", "line"],
                            "chp": ["backpressure", "extraction"],
        },
        path="/home/user/datpackages/my-datapackage"
)
```

3.2.2 Elements

We recommend using one tabular data resource (i.e. one csv-file) for each type you want to model. The fields (i.e. column names) match the attribute names specified in the description of the facade classes.

Example for **Load**:

```
| name       | type    | tech   |amount  | profile          | bus              |
|------------|---------|--------|--------|------------------|------------------|
| el-demand  | load    | load   | 2000   | demand-profile1  | electricity-bus  |
| ...        |  ...    | ....   | ...    |      ...         |      ...         |
```

The corresponding meta data *schema* of the resource would look as follows:

```
"schema": {
    "fields": [
        {
            "name": "name",
            "type": "string",
        },
        {
            "name": "type",
            "type": "string",
        },
        {
            "name": "tech",
            "type": "string",
        },
        {
            "name": "amount",
            "type": "number",
        },
        {
            "name": "profile",
            "type": "string",
        },
        {
```

(continues on next page)

(continued from previous page)

```
            "name": "bus",
            "type": "string",
        }
    ],
    "foreignKeys": [
          {
              "fields": "bus",
              "reference": {
                  "fields": "name",
                  "resource": "bus"
              }
          },
          {
              "fields": "profile",
              "reference": {
                  "resource": "load_profile"
              }
          }
    ],
}
```

Example for **Dispatchable**:

```
| name  | type         | capacity | capacity_cost   | bus             | marginal_cost␣
↪|
|-------|--------------|----------|-----------------|-----------------|--------------
↪|
| gen   | dispatchable | null     | 800             | electricity-bus | 75           ␣
↪|
| ...   |     ...      |    ...   |     ...         |     ...         |  ...         ␣
↪|
```

3.2.3 Sequences

A resource stored under */sequences* should at leat contain the field *timeindex* with the following standard format ISO 8601, i.e. *YYYY-MM-DDTHH:MM:SS*.

Example:

```
| timeindex        |  load-profile1   |  load-profile2   |
|------------------|------------------|------------------|
| 2016-01-01T00:00 |     0.1          |      0.05        |
| 2016-01-01T01:00 |     0.2          |      0.1         |
```

The schema for resource *load_profile* stored under *sequences/load_profile.csv* would be described as follows:

```
"schema": {
    "fields": [
        {
            "name": "timeindex",
            "type": "datetime",
        },
        {
            "name": "load-profile1",
            "type": "number",
```

(continues on next page)

(continued from previous page)

```
        },
        {
            "name": "load-profile2",
            "type": "number",
        }
    ]
}
```

3.3 Foreign Keys

Parameter types are specified in the (json) meta-data file corresponding to the data. In addition foreign keys can be specified to link elements entries to elements stored in other resources (for example buses or sequences).

To reference the *name* field of a resource with the bus elements (bus.csv, resource name: bus) the following FK should be set in the element resource:

```
"foreignKeys": [
  {
    "fields": "bus",
    "reference": {
      "fields": "name",
      "resource": "bus"
    }
  }
]
```

This structure can also be used to reference sequences, i.e. for the field *profile* of a resource, the reference can be set like this:

```
"foreignKeys": [
  {
    "fields": "profile",
    "reference": {
      "resource": "generator_profile"
    }
  }
]
```

In contrast to the above example, where the foreign keys points to a special field, in this case references are resolved by looking at the field names in the generators-profile resource.

> **Note:** This usage breaks with the datapackage standard and creates non-valid resources.**

3.4 Scripting

Currently the only way to construct a model and compute it is by using the *oemof.solph* library. As described above, you can simply use the command line tool on your created datapackage. However, you may also use the *facades.py* module and write your on application.

Just read the *.json* file to create an *solph.EnergySystem* object from the datapackage. Based on this you can create the model, compute it and process the results.

```
from oemof.solph import EnergySystem, Model
from renpass.facades import Load, Dispatchable, Bus

es = EnergySystem.from_datapackage(
    'datapackage.json',
    attributemap={
        Demand: {"demand-profiles": "profile"}},
    typemap={
        'load': Load,
        'dispatchable': Dispatchable,
        'bus': Bus})

m = Model(es)
m.solve()
```

Note: You may use the *attributemap* to map your your field names to facade class attributes. In addition you may also use different names for types in your datapackage and map those to the facade classes (use *typemap* attribute for this)

3.4.1 Write results

For writing results you either use the *oemof.outputlib* functionalities or / and the oemof tabular specific postprocessing functionalities of this package.

3.5 Reproducible Workflows

To get reproducible results we recommend setting up a folder strucutre as follows:

```
|-- model
        |-- environment
                |--requirements.txt
        |-- raw-data
        |-- scenarios
                |--scenario1.toml
                |--scenatio2.toml
                |-- ...
        |-- scripts
                |--create_input_data.py
                |--compute.py
                |-- ...
        |-- results
                |--scenario1
                        |--input
                        |--output
                 |-- scenario2
                        |--input
                        |--ouput
```

The *raw-data* directory contains all input data files required to build the input datapckages for your modelling. This data can also be downloaded from an additional repository which adheres to FAIR principles, like zenodo. If you provide raw data, make sure the license is compatiple with other data in your repository. The *scenarios* directory allows you to specify different scenarios and describe them in a basic way via config files. The *toml* standard is

used by oemof-tabular, howerver you may also use *yaml*, *json*, etc.. The scripts inside the *scripts* directory will build input data for your scenarios from the *.toml* files and the raw-data. This data will be in the format that oemof-tabular datapackage reader can understand. In addition the script to compute the models and postprocess results are stored there.

Of course the structure may be adapted to your needs. However you should provide all this data when publishing results.

3.6 Debugging

Debugging can sometimes be tricky, here are some things you might want to consider:

3.6.1 Components do not end up in the model

- Does the data resource (i.e. csv-file) for your components exist in the *datapackage.json* file
- Did you set the *attributemap* and *typemap* arguments of the *EnergySystem.from_datapackge()* method correctly? Make sure all classes with their types are present.

3.6.2 Errors when reading a datapackage

- Does the column order match the order of fields in the (tabular) data resource?
- Does the type match the types in of the columns (i.e. for integer, obviously only integer values should be in the respective column)

If you encounter this error message when reading a datapackage, you most likely provided *output_parameters* that are of type object for a tabular resource. However, there will be emtpy entries in the field of your *output_parameters*.

```
...
TypeError: type object argument after ** must be a mapping, not NoneType
```

Note: If your column / field in a tabular resource is of a specific type, make sure every entry in thies column has this type! For example numeric and empty entries in combination will yield string as a type and not numeric!

3.6.3 OEMOF related errors

If you encounter errors from oemof, the objects are not instantiated correctly which may happen if something of the following is wrong in your metadata file.

- Errors regarding the non-int type like this one:

  ```
  ...
  self.flows[o, i].nominal_value)
  TypeError: can't multiply sequence by non-int of type 'float'
  ```

  Check your type(s) in the *datapackage.json* file. If meta-data are inferred types might be string instead of number or integer which most likely causes such an error.

- Profiles for volatile and load components

```
...
ValueError: Cannot fix flow value to None.
Please set the actual_value attribute of the flow
```

This error is likely to occur if your foreign keys are set correctly but the name in the field *profile* of your *volatilel.csv* resource does not match any name inside the *volatile_profile.csv* file, i.e. the profile is not found where it is looked for.

Another possible source of error might be the missing values in your sequences files. Check these files for NaNs.

3.6.4 Solver and pyomo related errors

If you encounter an error for writing a lp-file, you might want to check if your foreign-keys are set correctly. In particular for resources with fk's for sequences. If this is missing, you will get unsupported operation string and numeric. This will unfortunately only happen on the pyomo level currently.

Also the following error might occure:

```
...
File "/home/admin/projects/oemof-tabular/venv/lib/python3.6/site-packages/
↪pyomo/repn/plugins/cpxlp.py", line 849, in _print_model_LP
% (_no_negative_zero(vardata_ub)))
TypeError: must be real number, not str
```

This message may indicate that fields in your datapackage that should be numeric are actually of type string. While pyomo seems sometimes still to be fine with this, solvers are not. Here also check your meta data types and the data. Most likely this happens if meta data is inferred from the data and fields with numeric values are left empty which will yield a string type for this field.

CHAPTER 4

Facade attributes overview

`BackpressureTurbine`

name	type	default
fuel_bus	Bus	
heat_bus	Bus	
electricity_bus	Bus	
carrier	str	
tech	str	
electric_efficiency	typing.Union[float, typing.Sequence[float]]	
thermal_efficiency	typing.Union[float, typing.Sequence[float]]	
capacity	float	
capacity_cost	float	
carrier_cost	float	0
marginal_cost	float	0
expandable	bool	False
input_parameters	dict	

`Commodity`

name	type	default
bus	Bus	
carrier	str	
amount	float	
marginal_cost	float	0.0
output_parameters	dict	

`Conversion`

name	type	default
from_bus	Bus	
to_bus	Bus	
carrier	str	
tech	str	
capacity	float	
efficiency	float	1
marginal_cost	float	0
carrier_cost	float	0
capacity_cost	float	
expandable	bool	False
capacity_potential	float	inf
capacity_minimum	float	
input_parameters	dict	
output_parameters	dict	

`Dispatchable`

name	type	default
bus	Bus	
carrier	str	
tech	str	
profile	typing.Union[float, typing.Sequence[float]]	1
capacity	float	
capacity_potential	float	inf
marginal_cost	float	0
capacity_cost	float	
capacity_minimum	float	
expandable	bool	False
output_parameters	dict	

`Excess`

name	type	default
bus	Bus	
marginal_cost	float	0
capacity	float	
capacity_potential	float	inf
capacity_cost	float	
capacity_minimum	float	
expandable	bool	False
input_parameters	dict	

`ExtractionTurbine`

name	type	default
carrier	str	
tech	str	
electricity_bus	Bus	
heat_bus	Bus	
fuel_bus	Bus	
condensing_efficiency	typing.Union[float, typing.Sequence[float]]	
electric_efficiency	typing.Union[float, typing.Sequence[float]]	
thermal_efficiency	typing.Union[float, typing.Sequence[float]]	
capacity	float	
carrier_cost	float	0
marginal_cost	float	0
capacity_cost	float	
expandable	bool	False
input_parameters	dict	
conversion_factor_full_condensation	dict	

`Generator`

name	type	default
bus	Bus	
carrier	str	
tech	str	
profile	typing.Union[float, typing.Sequence[float]]	1
capacity	float	
capacity_potential	float	inf
marginal_cost	float	0
capacity_cost	float	
capacity_minimum	float	
expandable	bool	False
output_parameters	dict	

`HeatPump`

name	type	default
electricity_bus	Bus	
high_temperature_bus	Bus	
low_temperature_bus	Bus	
carrier	str	
tech	str	
cop	float	
capacity	float	
marginal_cost	float	0
carrier_cost	float	0
capacity_cost	float	
expandable	bool	False
capacity_potential	float	inf
low_temperature_parameters	dict	
high_temperature_parameters	dict	
input_parameters	dict	

`Link`

name	type	default
from_bus	Bus	
to_bus	Bus	
from_to_capacity	float	
to_from_capacity	float	
loss	float	0
capacity_cost	float	
marginal_cost	float	0
expandable	bool	False
limit_direction	bool	False

`Load`

name	type	default
bus	Bus	
amount	float	
profile	typing.Union[float, typing.Sequence[float]]	
marginal_utility	float	0.0
input_parameters	dict	

`Reservoir`

name	type	default
bus	Bus	
carrier	str	
tech	str	
efficiency	float	
profile	typing.Union[float, typing.Sequence[float]]	
storage_capacity	float	
capacity	float	
output_parameters	dict	
expandable	bool	False

`Shortage`

name	type	default
bus	Bus	
carrier	str	
tech	str	
profile	typing.Union[float, typing.Sequence[float]]	1
capacity	float	
capacity_potential	float	inf
marginal_cost	float	0
capacity_cost	float	
capacity_minimum	float	
expandable	bool	False
output_parameters	dict	

`Storage`

name	type	default
bus	Bus	
carrier	str	
tech	str	
storage_capacity	float	0
capacity	float	0
capacity_cost	float	0
storage_capacity_cost	float	
storage_capacity_potential	float	inf
capacity_potential	float	inf
expandable	bool	False
marginal_cost	float	0
efficiency	float	1
input_parameters	dict	
output_parameters	dict	

`Volatile`

name	type	default
bus	Bus	
carrier	str	
tech	str	
profile	typing.Union[float, typing.Sequence[float]]	
capacity	float	
capacity_potential	float	inf
capacity_minimum	float	
expandable	bool	False
marginal_cost	float	0
capacity_cost	float	
output_parameters	dict	

CHAPTER 5

API Reference

5.1 oemof.tabular.datapackage package

5.1.1 Submodules

5.1.2 oemof.tabular.datapackage.aggregation module

Module used for aggregation sequences and elements.

oemof.tabular.datapackage.aggregation.**temporal_clustering**(*datapackage*, *n*, *path='/tmp'*, *how='daily'*)

Creates a new datapackage by aggregating sequences inside the *sequence* folder of the specified datapackage by clustering *n* timesteps

Parameters

- **datapackage** (*string*) – String of meta data file datapackage.json
- **n** (*integer*) – Number of clusters
- **path** (*string*) – Path to directory where the aggregated datapackage is stored
- **how** (*string*) – How to cluster 'daily' or 'hourly'

oemof.tabular.datapackage.aggregation.**temporal_skip**(*datapackage*, *n*, *path='/tmp'*, *name=None*, **args*)

Creates a new datapackage by aggregating sequences inside the *sequence* folder of the specified datapackage by skipping *n* timesteps

Parameters

- **datapackage** (*string*) – String of meta data file datapackage.json
- **n** (*integer*) – Number of timesteps to skip
- **path** (*string*) – Path to directory where the aggregated datapackage is stored

- **name** (*string*) – Name of the new, aggregated datapackage. If not specified a name will be given

5.1.3 oemof.tabular.datapackage.building module

oemof.tabular.datapackage.building.**download_data**(*url*, *directory='cache'*, *unzip_file=None*, ***kwargs*)

Downloads data and stores it in specified directory

Parameters

- **url** (*str*) – Url of file to be downloaded.
- **directory** (*str*) – Name of directory where to store the downloaded data. Default is 'cache'-
- **unzip_file** (*str*) – Regular or directory file name to be extracted from zip source.
- **kwargs** – Additional keyword arguments.

oemof.tabular.datapackage.building.**infer_metadata**(*package_name='default-name'*, *keep_resources=False*, *foreign_keys=None*, *path=None*, *metadata_filename='datapackage.json'*)

Add basic meta data for a datapackage

Parameters

- **package_name** (*string*) – Name of the data package
- **keep_resources** (*boolean*) – Flag indicating of the resources meta data json-files should be kept after main datapackage.json is created. The resource meta data will be stored in the *resources* directory.
- **foreign_keys** (*dict*) – Dictionary with foreign key specification. Keys for dictionary are: 'bus', 'profile', 'from_to_bus'. Values are list with strings with the name of the resources
- **path** (*string*) – Absolute path to root-folder of the datapackage
- **metadata_filename** (*basestring*) – Name of the inferred metadata string.

oemof.tabular.datapackage.building.**infer_resources**(*directory='data/elements'*)

Method looks at all files in *directory* and creates datapackage.Resource object that will be stored

Parameters **directory** (*string*) – Path to directory from where resources are inferred

oemof.tabular.datapackage.building.**initialize**(*config*, *directory='.'*)

Initialize datapackage by reading config file and creating required directories (data/elements, data/sequences etc.) if directories are not specified in the config file, the default directory setup up will be used.

oemof.tabular.datapackage.building.**input_filepath**(*file*, *directory='archive/'*)

oemof.tabular.datapackage.building.**package_from_resources**(*resource_path*, *output_path*, *clean=True*)

Collects resource descriptors and merges them in a datapackage.json

Parameters

- **resource_path** (*string*) – Path to directory with resources (in .json format)
- **output_path** (*string*) – Root path of datapackage where the newly created datapckage.json is stored
- **clean** (*boolean*) – If true, resources will be deleted

oemof.tabular.datapackage.building.**read_build_config**(*file='build.toml'*)
Read config build file in toml format

Parameters **file** (*string*) – String with name of config file

oemof.tabular.datapackage.building.**read_elements**(*filename*, *directory='data/elements'*)
Reads element resources from the datapackage

Parameters

- **filename** (*string*) – Name of the elements to be read, for example *load.csv*
- **directory** (*string*) – Directory where the file is located. Default: *data/elements*

Returns *pd.DataFrame*

oemof.tabular.datapackage.building.**read_sequences**(*filename*, *directory='data/sequences'*)
Reads sequence resources from the datapackage

Parameters

- **filename** (*string*) – Name of the sequences to be read, for example *load_profile.csv*
- **directory** (*string*) – Directory from where the file should be read. Default: *data/sequences*

oemof.tabular.datapackage.building.**timeindex**(*year*, *periods=8760*, *freq='H'*)
Create pandas datetimeindex.

Parameters

- **year** (*string*) – Year of the index
- **periods** (*string*) – Number of periods, default: 8760
- **freq** (*string*) – Freq of the datetimeindex, default: 'H'

oemof.tabular.datapackage.building.**update_package_descriptor**()

oemof.tabular.datapackage.building.**write_elements**(*filename*, *elements*, *directory='data/elements'*, *replace=False*, *overwrite=False*, *create_dir=True*)
Writes elements to filesystem.

Parameters

- **filename** (*string*) – Name of the elements to be read, for example *reservoir.csv*
- **elements** (*pd.DataFrame*) – Elements to be stored in data frame. Index: *name*
- **directory** (*string*) – Directory where the file is stored. Default: *data/elements*
- **replace** (*boolean*) – If set, existing data will be overwritten. Otherwise integrity of data (unique indices) will be checked
- **overwrite** (*boolean*) – If set, existing elements will be overwritten
- **create_dir** (*boolean*) – Create the directory if not exists

Returns **path** (*string*) – Returns the path where the file has been stored.

oemof.tabular.datapackage.building.**write_sequences**(*filename*, *sequences*, *directory='data/sequences'*, *replace=False*, *create_dir=True*)
Writes sequences to filesystem.

Parameters

- **filename** (*string*) – Name of the sequences to be read, for example *load_profile.csv*
- **sequences** (*pd.DataFrame*) – Sequences to be stored in data frame. Index: *datetimeindex* with format %Y-%m-%dT%H:%M:%SZ
- **directory** (*string*) – Directory where the file is stored. Default: *data/elements*
- **replace** (*boolean*) – If set, existing data will be overwritten. Otherwise integrity of data (unique indices) will be checked
- **create_dir** (*boolean*) – Create the directory if not exists

Returns **path** (*string*) – Returns the path where the file has been stored.

5.1.4 oemof.tabular.datapackage.processing module

oemof.tabular.datapackage.processing.**clean**(*path=None, directories=['data', 'cache', 'resources']*)

Parameters

- **path** (*str*) – Path to root directory of the datapackage, if no path is passed the current directory is to be assumed the root.
- **directories** (*list (optional)*) – List of directory names inside the root directory to clean (remove).

oemof.tabular.datapackage.processing.**copy_datapackage**(*source, destination, subset=None*)

Parameters

- **source** (*str*) – datapackage.json
- **destination** (*str*) – Destination of copied datapackage
- **name (optional)** (*str*) – Name of datapackage
- **only_data** (*str*) – Name of directory to only copy subset of datapackage (for example only the 'data' directory)

oemof.tabular.datapackage.processing.**to_dict**(*value*)
Convert value from e.g. csv-reader to valid json / dict

5.1.5 oemof.tabular.datapackage.reading module

Tools to deserialize energy systems from datapackages.

WARNING

This is work in progress and still pretty volatile, so use it at your own risk. The datapackage format and conventions we use are still a bit in flux. This is also why we don't have documentation or tests yet. Once things are stabilized a bit more, the way in which we extend the datapackage spec will be documented along with how to use the functions in this module.

oemof.tabular.datapackage.reading.**deserialize_constraints**(*model, path, constraint_type_map=None*)

oemof.tabular.datapackage.reading.**deserialize_energy_system**(*cls, path, typemap={}, attributemap={}*)

oemof.tabular.datapackage.reading.**read_facade**(*facade*, *facades*, *create*, *typemap*, *data*, *objects*, *sequence_names*, *fks*, *resources*)

Parse the resource *r* as a facade.

oemof.tabular.datapackage.reading.**sequences**(*r*, *timeindices=None*)

Parses the resource *r* as a sequence.

5.2 oemof.tabular.tools package

oemof.tabular's kitchen sink module.

Contains all the general tools needed by other tools dealing with specific tabular data sources.

class oemof.tabular.tools.**HSN**

Bases: types.SimpleNamespace

A hashable variant of *types.Simplenamespace*.

By making it hashable, we can use the instances as dictionary keys, which is necessary, as this is the default type for flows.

oemof.tabular.tools.**raisestatement**(*exception*, *message=''*)

A version of *raise* that can be used as a statement.

oemof.tabular.tools.**remap**(*mapping*, *renamings*, *selection*)

Change *mapping*'s keys according to the *selection* in *renamings*.

The *renaming* found under *selection* in *renamings* is used to rename the keys found in *mapping*. I.e., return a copy of *mapping* with every *key* of *mapping* that is also found in *renaming* replaced with *renaming[key]*.

If key doesn't have a renaming, it's returned as is. If *selection* doesn't appear as a key in *renamings*, *mapping* is returned unchanged.

Example

```
>>> renamings = {'R1': {'zero': 'nada'}, 'R2': {'foo': 'bar'}}
>>> mapping = {'zero': 0, 'foo': 'foobar'}
>>> remap(mapping, renamings, 'R1') == {'nada': 0, 'foo': 'foobar'}
True
>>> remap(mapping, renamings, 'R2') == {'zero': 0, 'bar': 'foobar'}
True
```

As a special case, if *selection* is a *class*, not only *selection* is considered to select a renaming, but the classes in *selection*'s *mro* are considered too. The first class in *selection*'s *mro* which is also found to be a key in *renamings* is used to determine which renaming to use. The search starts at *selection*.

Parameters

- **mapping** (*Mapping*) – The *Mapping* whose keys should be renamed.
- **renamings** (*Mapping* of *Mappings* <*collections.abc.Mapping*>)
- **selection** (*Hashable*) – Key specifying which entry in *renamings* is used to determine the new keys in the copy of *mapping*. If *selection* is a *class*, the first entry of *selection*'s *mro* which is found in *renamings* is used to determine the new keys.

5.2.1 Submodules

5.2.2 oemof.tabular.tools.geometry module

The code in this module is partly based on third party code which has been licensed under GNU-GPL3. The following functions are copied and adapted from:

https://github.com/FRESNA/vresutils, Copyright 2015-2017 Frankfurt Institute for Advanced Studies

- _shape2poly()
- simplify_poly()
- nuts()

oemof.tabular.tools.geometry.**Shapes2Shapes**(*orig*, *dest*, *normed=True*, *equalarea=False*, *prep_first=True*, ***kwargs*)

Notes

Copied from: https://github.com/FRESNA/vresutils, Copyright 2015-2017 Frankfurt Institute for Advanced Studies

oemof.tabular.tools.geometry.**intersects**(*geom*, *labels*, *geometries*)

oemof.tabular.tools.geometry.**nuts**(*filepath=None*, *nuts=0*, *subset=None*, *tolerance=0.03*, *minarea=1.0*)

Reads shapefile with nuts regions and converts to polygons

Returns *OrderedDict* – Country keys as keys of dict and shapely polygons as corresponding values

Notes

Copied from: https://github.com/FRESNA/vresutils, Copyright 2015-2017 Frankfurt Institute for Advanced Studies

oemof.tabular.tools.geometry.**read_geometries**(*filename*, *directory='data/geometries'*)

Reads geometry resources from the datapackage. Data may either be stored in geojson format or as WKT representation in CSV-files.

Parameters

- **filename** (*string*) – Name of the elements to be read, for example *buses.geojson*
- **directory** (*string*) – Directory where the file is located. Default: *data/geometries*

Returns *pd.Series*

oemof.tabular.tools.geometry.**reproject**(*geom*, *fr=<sphinx.ext.autodoc.importer._MockObject object>*, *to=<sphinx.ext.autodoc.importer._MockObject object>*)

Notes

Copied and adapted from: https://github.com/FRESNA/vresutils, Copyright 2015-2017 Frankfurt Institute for Advanced Studies

oemof.tabular.tools.geometry.**simplify_poly**(*poly*, *tolerance*)

Notes

Copied from: https://github.com/FRESNA/vresutils, Copyright 2015-2017 Frankfurt Institute for Advanced Studies

oemof.tabular.tools.geometry.**write_geometries**(*filename*, *geometries*, *directory='data/geometries'*)

Writes geometries to filesystem.

Parameters

- **filename** (*string*) – Name of the geometries stored, for example *buses.geojson*
- **geometries** (*pd.Series*) – Index entries become name fields in GeoJSON properties.
- **directory** (*string*) – Directory where the file is stored. Default: *data/geometries*

Returns **path** (*string*) – Returns the path where the file has been stored.

5.3 oemof.tabular.facades module

CHAPTER 6

Contributing

Contributions are welcome, and they are greatly appreciated! Every little bit helps, and credit will always be given.

6.1 Bug reports

When reporting a bug please include:

- Your operating system name and version.
- Any details about your local setup that might be helpful in troubleshooting.
- Detailed steps to reproduce the bug.

6.2 Documentation improvements

oemof.tabular could always use more documentation, whether as part of the official oemof.tabular docs, in docstrings, or even on the web in blog posts, articles, and such.

6.3 Feature requests and feedback

The best way to send feedback is to file an issue at https://github.com/oemof/oemof-tabular/issues.

If you are proposing a feature:

- Explain in detail how it would work.
- Keep the scope as narrow as possible, to make it easier to implement.
- Remember that this is a volunteer-driven project, and that code contributions are welcome :)

6.4 Development

To set up *oemof-tabular* for local development:

1. Fork oemof-tabular (look for the "Fork" button).
2. Clone your fork locally:

```
git clone git@github.com:your_name_here/oemof-tabular.git
```

3. Create a branch for local development:

```
git checkout -b name-of-your-bugfix-or-feature
```

   Now you can make your changes locally.

4. When you're done making changes, run all the checks, doc builder and spell checker with tox one command:

```
tox
```

5. Commit your changes and push your branch to GitHub:

```
git add .
git commit -m "Your detailed description of your changes."
git push origin name-of-your-bugfix-or-feature
```

6. Submit a pull request through the GitHub website.

6.4.1 Pull Request Guidelines

If you need some code review or feedback while you're developing the code just make the pull request.

For merging, you should:

1. Include passing tests (run `tox`)[1].
2. Update documentation when there's new API, functionality etc.
3. Add a note to `CHANGELOG.rst` about the changes.
4. Add yourself to `AUTHORS.rst`.

6.4.2 Tips

To run a subset of tests:

```
tox -e envname -- pytest -k test_myfeature
```

To run all the test environments in *parallel* (you need to `pip install detox`):

```
detox
```

[1] If you don't have all the necessary python versions available locally you can rely on Travis - it will run the tests for each change you add in the pull request.
It will be slower though ...

CHAPTER 7

Authors

- Stephan Günther - https://oemof.org
- Simon Hilpert
- Martin Söthe
- Cord Kaldemeyer
- Jann Launer
- Hendrik Huyskens
- Monika Orlowski
- Francesco Witte
- Sarah Berendes
- Marie-Claire Gering

CHAPTER 8

Changelog

8.1 0.0.3 (2022-01-26)

8.1.1 Fixes

- Fix link by not setting constraints that limit direction #38
- Fix storage investment #33
- Link investment #28
- Variable cost #24
- Marginal cost #23

8.1.2 Features

- Adjust to new oemof.solph structure #21
- Allow to define custom foreign keys #39
- Add constraint tests for most facades #35, #42
- Reduce number of imported packages #32, #49
- Cleaned up the badges in README #59
- Move most CI services to github actions #37

8.2 0.0.2 (2019-07-08)

8.3 0.0.1 (2018-12-12)

- Moved the datapackage reader from core *oemof* into this package. That means the basic functionality of deserializing energy systems from datapackages has finally arrived.
- Moved *Facade* classes from *renpass* into this package. The *Facade* classes are designed to complement the datapackage reader, by enabling easy construction of energy system components from simple tabular data sources.
- Also moved the example datapackages from *renpass* into this package. These datapackages provide a good way of at least testing, that the datapackage reader doesn't throw errors.

8.4 0.0.0 (2018-11-23)

- First release on PyPI. Pretty much non functional because it only consists of the package boilerplate and nothing else. But this is what a version zero is for, IMHO.

CHAPTER 9

Indices and tables

- genindex
- modindex
- search

Python Module Index

o

Index

U

W