otoole Documentation

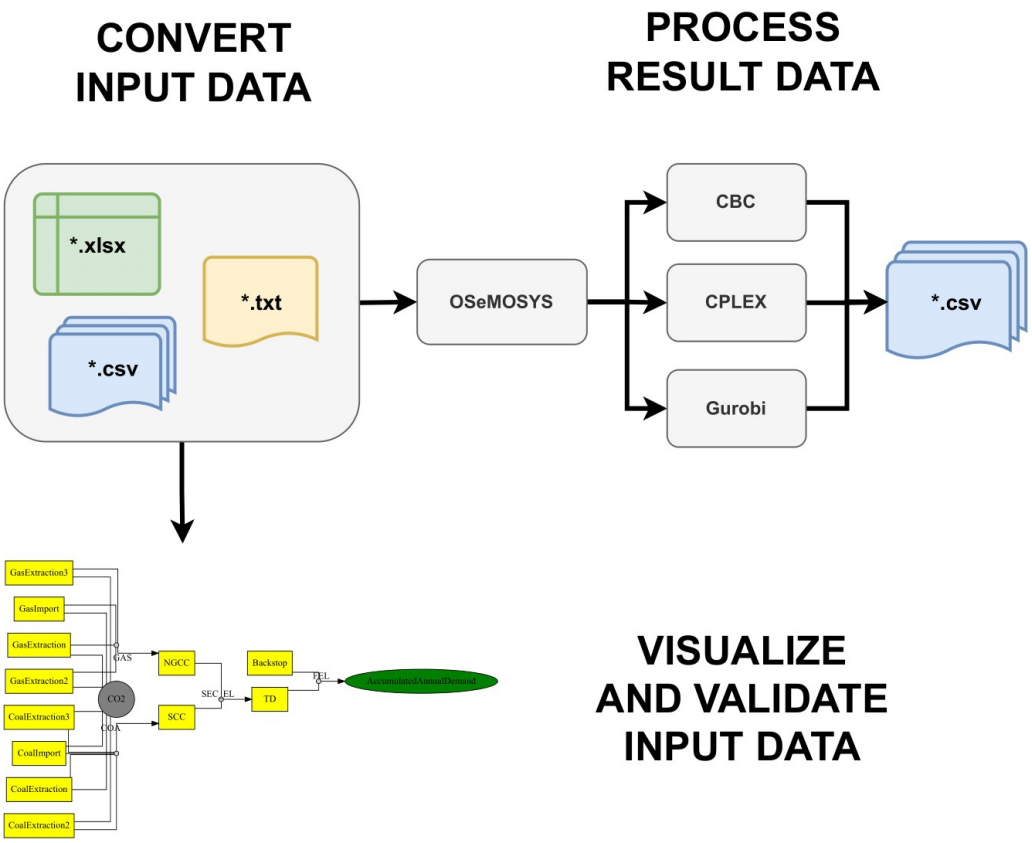
Release 1.1.2.post1.dev17+g8e103ed

Will Usher

Mar 31, 2024

CONTENTS

otoole, or **O**SeMOSYS **tool**s for **e**nergy work, is a Python package which provides a command-line interface and Python API for users of OSeMOSYS. The aim of the package is to provide commonly used pre- and post-processing steps when working with OSeMOSYS models. Specifically, `otoole` allows the user to convert between data formats, process solutions, and visualise the reference energy system.



Currently, `otoole` only supports the GNU MathProg version of OSeMOSYS. We hope to build support for other implementations of OSeMOSYS soon. In fact, this is one of the key aims of `otoole` - to better link the various OSeMOSYS communities and their implementations through common data standards and useful scripts and tools that we can work on together.

CHAPTER

ONE

CONTENTS

1.1 Getting Started

1.1.1 Installation

Install `otoole` using pip:

```
pip install otoole
```

Check the version installed:

```
~ otoole -V
1.0.0
```

To upgrade `otoole` using pip:

```
pip install otoole --upgrade
```

Tip: We recommend installing `otoole` in an isolated virtual environment, either through the use of venv or conda

1.1.2 Dependencies

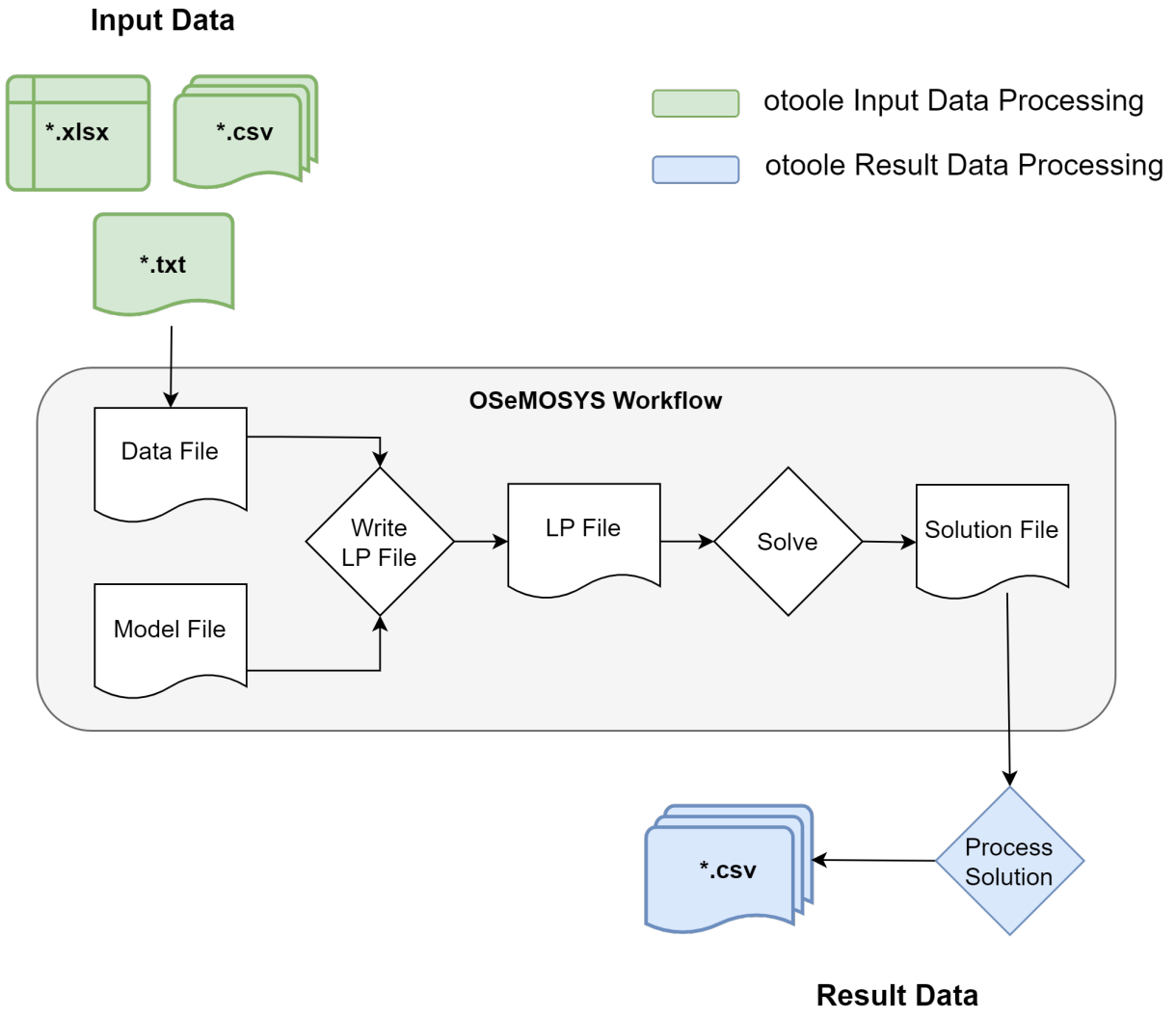
`otoole` relies on a number of dependencies. If the user wants to download the individual dependencies, the easiest way to do this is through miniconda.

1. Obtain the miniconda package
2. Add the `conda-forge` channel `conda config --add channels conda-forge`
3. Create a new Python environment `conda create -n otoole python>3.7 networkx pandas graphviz=2.46.1 xlrd pydantic`
4. Activate the new environment `conda activate otoole`
5. Use pip to install otoole `pip install otoole`

1.2 Core Functionality

`otoole`'s functionality includes converting between data formats, processing solution files, and visualising the model. These core functions, which provide a method to turbo-charge your modelling, are described on this page!

As shown in the diagram, `otoole` deals primarily with data before and after OSeMOSYS. Data work prior to the generation of a datafile, which is read in by GLPK, is called pre-processing. Anything that happens with the immediate outputs of a solver, such as the recommended open-source solvers GLPK, CBC, or the proprietary solvers CPLEX and Gurobi, is called results and post-processing.



Note: While `otoole` is targeted at OSeMOSYS users, the functionality can easily be extended to work with any workflow that involves the use of a MathProg file!

1.2.1 Data Conversion

Overview

`otoole` supports different data pre-processing conversions so as to ease the tasks of the OSeMOSYS modeller. The modeller can generate data in any one of the formats and convert it to another format through a `convert` command. `otoole` currently supports conversion between the following formats:

- Excel
- A folder of CSV files
- GNU MathProg datafile

`otoole convert`

The `otoole convert` command allows you to convert between various different input formats:

```
$ otoole convert --help
usage: otoole convert [-h] [--write_defaults] {csv,datafile,excel} {csv,datafile,excel}␣
↪from_path to_path config

positional arguments:
{csv,datafile,excel}  Input data format to convert from
{csv,datafile,excel}  Input data format to convert to
from_path             Path to file or folder to convert from
to_path               Path to file or folder to convert to
config                Path to config YAML file

optional arguments:
-h, --help            show this help message and exit
--write_defaults      Writes default values
--keep_whitespace     Keeps leading/trailing whitespace
```

Deprecated since version v1.0.0: The `datapackage` format is no longer supported

New in version v1.0.0: The `config` positional argument is now required

1.2.2 Result Processing

Overview

With small OSeMOSYS models, it is normally fine to use the free open-source GLPK solver. If you do, then OSeMOSYS will write out a full set of results as a folder of CSV files. As you progress to larger models, the performance constraints of GLPK quickly become apparent. CBC is an alternative open-source solver which offers better performance than GLPK and can handle much larger models. However, CBC has no way of knowing how to write out the CSV files you were used to dealing with when using GLPK. `otoole` to the rescue!

`otoole` currently supports using GLPK, CBC, CPLEX or Gurobi with all versions of GNU MathProg OSeMOSYS - the long, short and fast versions.

The long version includes all results as variables within the formulation, so the `otoole results` command parses the solution file, extracts the required variables, and produces a folder of CSV files containing the results in an identical format to if you had used GLPK.

The short and fast versions omit a large number of these calculated result variables so as to speed up the model matrix generation and solution times.

otoole results

The `results` command creates a folder of CSV result files from a GLPK, CBC, CLP, Gurobi or CPLEX solution file together with the input data:

```
$ otoole results --help
usage: otoole results [-h] [--glpk_model GLPK_MODEL] [--write_defaults]
                      {cbc,cplex,gurobi} {csv} from_path to_path {csv,datafile,excel}␣
↪input_path config

positional arguments:
{cbc,cplex,glpk,gurobi}  Result data format to convert from
{csv}                    Result data format to convert to
from_path                Path to file or folder to convert from
to_path                  Path to file or folder to convert to
{csv,datafile,excel}     Input data format
input_path               Path to input_data
config                   Path to config YAML file

optional arguments:
-h, --help                show this help message and exit
--glpk_model GLPK_MODEL   GLPK model file required for processing GLPK results
--write_defaults          Writes default values
```

New in version v1.0.0: The `config` positional argument is now required

New in version v1.1.0: The `input_data_format` and `input_path` positional arguments are now required supporting any supported format of input data for results processing.

Deprecated since version v1.0.0: The `--input_datapackage` flag is no longer supported

Deprecated since version v1.1.0: The `--input_datapackage` and `--input_datafile` flags have been replaced by new positional arguments `input_data_format` and `input_path`

1.2.3 Setup

The `setup` module in `otoole` allows you to generate template files to quickly get up and running.

otoole setup

The `setup` command allows you to generate a template user configuration file, useful for `conversion` and `result` commands, and template input `csv` data:

```
$ otoole setup --help

usage: otoole setup [-h] [--write_defaults] [--overwrite] {config,csv} data_path

positional arguments:
{config,csv}      Type of file to setup
data_path         Path to file or folder to save to
```

(continues on next page)

(continued from previous page)

```
optional arguments:
-h, --help        show this help message and exit
--write_defaults  Writes default values
--overwrite       Overwrites existing data
```

Warning: The template files are generated based on a specific version of OSeMOSYS, users will need to adapt the template data for their own needs

1.2.4 Visualization

Overview

The visualization module in `otoole` allows you to visualise the reference energy system. (with more visualisations to come!)

`otoole viz`

The `viz` command allows you to visualise aspects of the model. Currently, only visualising the reference energy system through the `vis res` command is supported:

```
$ otoole viz res --help

usage: otoole viz res [-h] {csv,datafile,excel} data_path resfile config

positional arguments:
{csv,datafile,excel}  Input data format
data_path             Input data path
resfile               Path to reference energy system
config                Path to config YAML file

optional arguments:
-h, --help            show this help message and exit
```

Note: The `resfile` command should include a file ending used for images, including `bmp`, `jpg`, `pdf`, `png` etc. The graphviz library used to layout the reference energy system will interpret the file ending.

1.2.5 Validation

The validation module in `otoole` checks technology and fuel names against a standard or user defined configuration file.

`otoole validate`

The `validate` command allows you to identify any incorrectly named technologies or fuels, by comparing against a user defined validation configuration file. Moreover, `otoole` will check if any technology or fuel are unconnected from the rest of the model:

```
$ otoole validate --help

usage: otoole validate [-h] [--validate_config VALIDATE_CONFIG] {csv,datafile,excel}␣
↪data_file user_config

positional arguments:
{csv,datafile,excel}  Input data format
data_file             Path to the OSeMOSYS data file to validate
user_config           Path to config YAML file

optional arguments:
-h, --help            show this help message and exit
--validate_config VALIDATE_CONFIG
                        Path to a user-defined validation-config file
```

1.3 Data Formats

This page explains the different data formatting options available in otoole. Firstly, the format of the user configuration file is explained. Following this, the different input data formats are explained.

See also:

See the Simplicity repository for a full example of these formats

1.3.1 User Configuration File

New in version v1.0.0: The user configuration file is now required for data conversion

Overview

Most commands in `otoole` require the user to specify a configuration file that describes the `parameters`, `sets`, and `results` in the model. This configuration file is written in `yaml` and is typically saved as `config.yaml`. This section will cover how to format the user configuration file. If the user incorrectly enters data, validation checks in `otoole` should catch this.

Information Required

The table below highlights what information is required for each `Set`, `Parameter` and `Result` definition in the configuration file. Required values are given by **X**, while optional values are given by **(X)**.

	Set	Parameter	Result
name	X	X	X
short_name	(X)	(X)	(X)
dtype	X	X	X
type	X	X	X
default		X	X

Deprecated since version v1.0.3: The `Calculated` keyword is no longer needed for Result definitions

Warning: Names longer than 31 characters require a `short_name` field. This is due to character limits on excel sheet names. `otoole` will raise an error if a `short_name` is not provided in these instances.

Sets Format

Sets are defined as follows:

```
SETNAME:
    short_name: SET (Optional)
    dtype: "int" or "string"
    type: set
```

Note: It's convention in OSeMOSYS to capitalize set names

Parameters Format

Parameters are defined as follows. When referencing set indices use the full name, **not** the `short_name`:

```
ParameterName:
    short_name: ParamName (Optional)
    indices: [SETNAME, SETNAME, ...]
    type: param
    dtype: "int" or "float"
    default: 0
```

Note: It's convention in OSeMOSYS to use Pascal case for parameter names

Results Format

Results are defined as follows. When referencing set indices use the full name, **not** the `short_name`:

```
AnnualEmissions:
    short_name: ParamName (Optional)
    indices: [SETNAME, SETNAME, ...]
    type: result
    dtype: "int" or "float"
    default: 0
```

Note: It's convention in OSeMOSYS to use Pascal case for result names

Examples

Below are examples of correctly formatted configuration file values. See the Simplicity repository for a complete example.

1. Set definition of `TECHNOLOGY`:

```
TECHNOLOGY:
    dtype: str
    type: set
```

2. Parameter definition of `AccumulatedAnnualDemand`:

```
AccumulatedAnnualDemand:
    short_name: AccAnnualDemand
    indices: [REGION,FUEL,YEAR]
    type: param
    dtype: float
    default: 0
```

3. Result definition of `AnnualEmissions`:

```
AnnualEmissions:
    indices: [REGION,EMISSION,YEAR]
    type: result
    dtype: float
    default: 0
```

Tip: See the *Examples* page to create a template configuration file

1.3.2 Input Data

Deprecated since version v1.0.0: The `datapackage` format is no longer supported

Overview

This section will describe how to format data for `excel`, `csv`, and `datafile` formats.

Excel

Interfacing with `otoole` through excel is a very user-friendly method to handle OSeMOSYS input data. In the excel workbook (an `*.xlsx` file), each sheet will correspond to a single parameter or set. Parameters that are indexed over years are pivoted on the `YEAR` index. This creates a wide formatted dataset, where each year is the column header, with the first columns holding the remaining indices.

For example, referencing the Simplicity model, the `AccumulatedAnnualDemand` parameter data will be under the `AccumulatedAnnualDemand` sheet and contain the data

REGION	TECHNOLOGY	2014	2015	2016	2017	2018	2019	2020
SIMPLICITY	BACKSTOP1	999999	999999	999999	999999	999999	999999	999999
SIMPLICITY	BACKSTOP2	999999	999999	999999	999999	999999	999999	999999
SIMPLICITY	ETHPLANT	25	25	25	25	25	25	25
SIMPLICITY	GRID_EXP	4000	4000	4000	4000	4000	4000	4000
SIMPLICITY	HYD1	4500	4500	4500	4500	4500	4500	4500
SIMPLICITY	HYD2	3500	3500	3500	3500	3500	3500	3500
…	…	…	…	…	…	…	…	…

Parameters that are not indexed over years will have an extra column titled `VALUE`. This column will hold the input value for that parameter. For example, the `OperationalLife` parameter in the Simplicity example will be formatted as shown

REGION	TECHNOLOGY	VALUE
SIMPLICITY	BACKSTOP1	1
SIMPLICITY	BACKSTOP2	1
SIMPLICITY	ETHPLANT	30
SIMPLICITY	GAS_EXTRACTION	1
SIMPLICITY	GAS_IMPORT	1
SIMPLICITY	GRID_EXP	50
SIMPLICITY	HYD1	80
SIMPLICITY	HYD2	80
…	…	…

Set definitions will have a single column, titled `VALUE`. For example, the set `TECHNOLOGY` will be formatted as shown

VALUE
BACKSTOP1
BACKSTOP2
ETHPLANT
GAS_EXTRACTION
GAS_IMPORT
GRID_EXP
HYD1
HYD2
…

CSV

Interfacing with `otoole` through a folder of CSV files is the most "computer friendly" way to handle input data. This is due to csv files being easy to read and write, and independent of the program, programming language, and operating system. This allows `otoole` to easily integrate into workflows.

When working with CSV data, all parameters and sets are saved under their name given in the configuration file, and nested in a single directory. CSV data will follow long formatting standards, where each column is the name of the index, and the final column is titled `VALUE`.

For example, the following data for `AccumulatedAnnualDemand` will be under the file `data/AccumulatedAnnualDemand.csv`

REGION	FUEL	YEAR	VALUE
SIMPLICITY	ETH	2014	1
SIMPLICITY	RAWSUG	2014	0.5
SIMPLICITY	ETH	2015	1.03
SIMPLICITY	RAWSUG	2015	0.51
SIMPLICITY	ETH	2016	1.061
SIMPLICITY	RAWSUG	2016	0.519
SIMPLICITY	ETH	2017	1.093
SIMPLICITY	RAWSUG	2017	0.529
SIMPLICITY	ETH	2018	1.126
…	…	…	…

While the `TECHNOLOGY` set data will be under the file `data/TECHNOLOGY.csv`` and formatted as shown with a single `VALUE` column.

VALUE
BACKSTOP1
BACKSTOP2
ETHPLANT
GAS_EXTRACTION
GAS_IMPORT
GRID_EXP
HYD1
HYD2
…

Datafile

Datafiles are the least user-friendly method of handling data, however, they are required for the OSeMOSYS GNU MathProg version of OSeMOSYS. Datafiles are written in MathProg, which shares syntax with the AMPL programming language.

Datafiles contain all model data in one file (often a `*.txt` file), and will follow a similar data standard to long formatted CSV data. However, the default value for the parameter is included in its declaration statement.

For example, in the file `data.txt`, the parameter `AccumulatedAnnualDemand` will be defined as follows:

```
param default 0.0 : AccumulatedAnnualDemand :=
    SIMPLICITY ETH 2014 1
    SIMPLICITY RAWSUG 2014 0.5
    SIMPLICITY ETH 2015 1.03
    SIMPLICITY RAWSUG 2015 0.51
    SIMPLICITY ETH 2016 1.061
    SIMPLICITY RAWSUG 2016 0.519
    SIMPLICITY ETH 2017 1.093
    SIMPLICITY RAWSUG 2017 0.529
    SIMPLICITY ETH 2018 1.126
    SIMPLICITY RAWSUG 2018 0.538
    SIMPLICITY ETH 2019 1.159
    SIMPLICITY RAWSUG 2019 0.548
    SIMPLICITY ETH 2020 1.194
    SIMPLICITY RAWSUG 2020 0.558
    ...
```

And in the same `data.txt` file, the set `TECHNOLOGY` will be defined as follows:

```
set TECHNOLOGY :=
    BACKSTOP1
    BACKSTOP2
    ETHPLANT
    GAS_EXTRACTION
    GAS_IMPORT
    GRID_EXP
    HYD1
    HYD2
    ...
```

See also:

For reading and writing between Python and AMPL, see the amply Python package.

1.4 Examples

This page will present examples to show the full functionality of `otoole`. It will walk through the `convert`, `results`, `setup`, `viz` and `validate` functionality in separate simple use cases.

Note: To follow these examples, clone the Simplicity repository and run all commands from the `simplicity/` directory:

```
git clone https://github.com/OSeMOSYS/simplicity.git
cd simplicity
```

1.4.1 Solver Setup

Objective

Install GLPK (required) and CBC (optional) to use in the otoole examples. While `otoole` does not require a solver, these examples will use the free and open source solvers GLPK and CBC.

1. Install GLPK

GLPK is a free and open-source linear program solver. Full install instructions can be found on the GLPK Website, however, the abbreviated instructions are shown below

To install GLPK on **Linux**, run the command:

```
$ sudo apt-get update
$ sudo apt-get install glpk glpk-utils
```

To install GLPK on **Mac**, run the command:

```
$ brew install glpk
```

To install GLPK on **Windows**, follow the instructions on the GLPK Website. Be sure to add GLPK to your environment variables after installation

Alternatively, if you use Anaconda to manage your Python packages, you can install GLPK via the command:

```
$ conda install -c conda-forge glpk
```

2. Test the GLPK install

Once installed, you should be able to call the `glpsol` command:

```
$ glpsol
GLPSOL: GLPK LP/MIP Solver, v4.65
No input problem file specified; try glpsol --help
```

Tip: See the GLPK Wiki for more information on the `glpsol` command

3. Install CBC

CBC is a free and open-source mixed integer linear programming solver. Full install instructions can be found on the CBC website, however, the abbreviated instructions are shown below

To install CBC on **Linux**, run the command:

```
$ sudo apt-get install coinor-cbc coinor-libcbc-dev
```

To install CBC on **Mac**, run the command:

```
$ brew install coin-or-tools/coinor/cbc
```

To install CBC on **Windows**, follow the install instruction on the CBC website.

Alternatively, if you use Anaconda to manage your Python packages, you can install CBC via the command:

```
$ conda install -c conda-forge coincbc
```

4. Test the CBC install

Once installed, you should be able to directly call CBC:

```
$ cbc
Welcome to the CBC MILP Solver
Version: 2.10.3
Build Date: Mar 24 2020

CoinSolver takes input from arguments ( - switches to stdin)
Enter ? for list of commands or help
Coin:
```

You can exit the solver by typing `quit`

1.4.2 Input Data Conversion

Objective

Convert input data between CSV, Excel, and GNU MathProg data formats.

1. Clone `Simplicity`

If not already done so, clone the Simplicity repository:

```
$ git clone https://github.com/OSeMOSYS/simplicity.git
$ cd simplicity
```

Note: Further information on the `config.yaml` file is in the *Template Setup* section

2. Convert CSV data into MathProg data

Convert the folder of Simplicity CSVs (`data/`) into an OSeMOSYS datafile called `simplicity.txt`:

```
$ otoole convert csv datafile data simplicity.txt config.yaml
```

3. Convert MathProg data into Excel Data

Convert the new Simplicity datafile (`simplicity.txt`) into Excel data called `simplicity.xlsx`:

```
$ otoole convert datafile excel simplicity.txt simplicity.xlsx config.yaml
```

Tip: Excel workbooks are an easy way for humans to interface with OSeMOSYS data!

4. Convert Excel Data into CSV data

Convert the new Simplicity excel data (`simplicity.xlsx`) into a folder of CSV data called `simplicity/`. Note that this data will be the exact same as the original CSV data folder (`data/`):

```
$ otoole convert excel csv simplicity.xlsx simplicity config.yaml
```

1.4.3 Process Solutions from Different Solvers

Objective

Process solutions from GLPK, CBC, Gurobi, and CPLEX. This example assumes you have an existing GNU MathProg datafile called `simplicity.txt` (from the previous example).

1. Process a solution from GLPK

Use GLPK to build the model, save the problem as `simplicity.glp`, solve the model, and save the solution as `simplicity.sol`. Use otoole to create a folder of CSV results called `results-glpk/`. When processing solutions from GLPK, the model file (`*.glp`) must also be passed:

```
$ glpsol -m OSeMOSYS.txt -d simplicity.txt --wglp simplicity.glp --write simplicity.sol

$ otoole results glpk csv simplicity.sol results-glpk datafile simplicity.txt config.
↪yaml --glpk_model simplicity.glp
```

Note: By default, MathProg OSeMOSYS models will write out folder of CSV results to a `results/` directory if solving via GLPK. However, using `otoole` allows the user to programmatically access results and control read/write locations

2. Process a solution from CBC

Use GLPK to build the model and save the problem as `simplicity.lp`. Use CBC to solve the model and save the solution as `simplicity.sol`. Use otoole to create a folder of CSV results called `results/` from the solution file:

```
$ glpsol -m OSeMOSYS.txt -d simplicity.txt --wlp simplicity.lp --check

$ cbc simplicity.lp solve -solu simplicity.sol

$ otoole results cbc csv simplicity.sol results csv data config.yaml
```

3. Process a solution from Gurobi

Use GLPK to build the model and save the problem as `simplicity.lp`. Use Gurobi to solve the model and save the solution as `simplicity.sol`. Use otoole to create a folder of CSV results called `results/` from the solution file:

```
$ glpsol -m OSeMOSYS.txt -d simplicity.txt --wlp simplicity.lp --check

$ gurobi_cl ResultFile=simplicity.sol simplicity.lp

$ otoole results gurobi csv simplicity.sol results csv data config.yaml
```

4. Process a solution from CPLEX

Use GLPK to build the model and save the problem as `simplicity.lp`. Use CPLEX to solve the model and save the solution as `simplicity.sol`. Use otoole to create a folder of CSV results called `results/` from the solution file:

```
$ glpsol -m OSeMOSYS.txt -d simplicity.txt --wlp simplicity.lp --check

$ cplex -c "read simplicity.lp" "optimize" "write simplicity.sol"

$ otoole results cplex csv simplicity.sol results csv data config.yaml
```

1.4.4 Model Visualization

Objective

Use `otoole` to visualize the reference energy system.

1. `otoole` Visualise

The visualization functionality of `otoole` will work with any supported input data format (`csv`, `datafile`, or `excel`). In this case, we will use the excel file, `simplicity.xlsx`, to generate the RES.

Run the following command, where the RES will be saved as the file `res.png`:

```
$ otoole viz res excel simplicity.xlsx res.png config.yaml
```

Warning: If you encounter a `graphviz` dependency error, install it on your system from the graphviz website (if on Windows) or via the command:

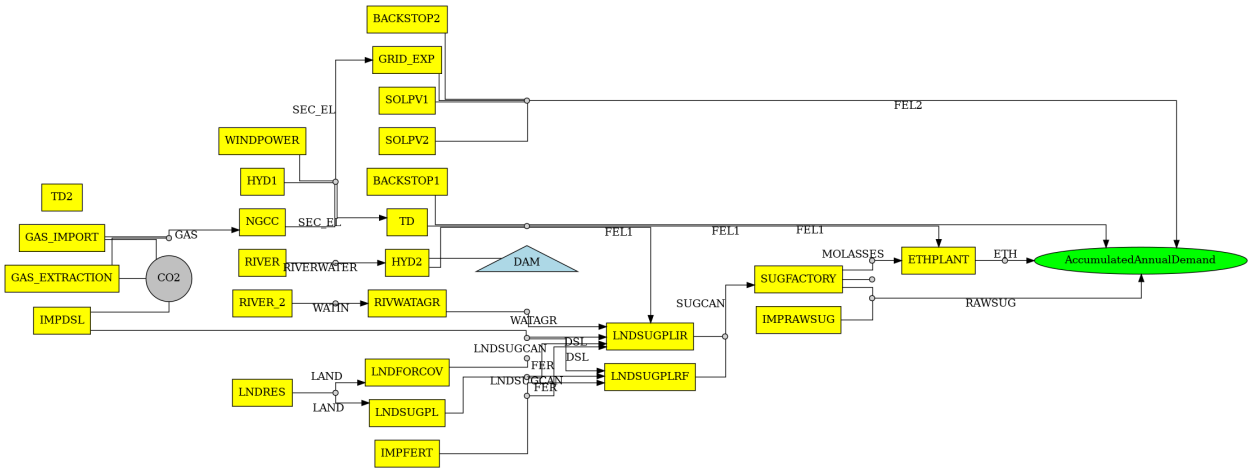
```
sudo apt install graphviz # if on Ubuntu
brew install graphviz # if on Mac
```

To check that `graphviz` installed correctly, run `dot -V` to check the version:

```
~$ dot -V
dot - graphviz version 2.43.0 (0)
```

1. View the RES

Open the newly created file, `res.png` and the following image should be displayed



1.4.5 Template Setup

Objective

Generate a template configuration file and excel input file to use with `otoole convert` commands

1. Create the Configuration File

Run the following command, to create a template configuration file called `config.yaml`:

```
$ otoole setup config template_config.yaml
```

2. Create the Template Data CSVs

`otoole` will only generate template CSV data, however, we want to input data in Excel format. Therefore, we will first generate CSV data and convert it to Excel format:

```
$ otoole setup csv template_data
```

3. Add Year Definitions

Open up the the file `template_data/YEARS.csv` and add all the years over the model horizon. For example, if the model horizon is from 2020 to 2050, the `template_data/YEARS.csv` file should be formatted as follows:

VALUE
2020
2021
2022
...
2050

Note: While this step in not technically required, by filling out the years in CSV format `otoole` will pivot all the Excel sheets on these years. This will save significant formatting time!

4. Convert the CSV Template Data

Convert the template CSV data into Excel formatted data:

```
$ otoole convert csv excel template_data template.xlsx template_config.yaml
```

5. Add Model Data

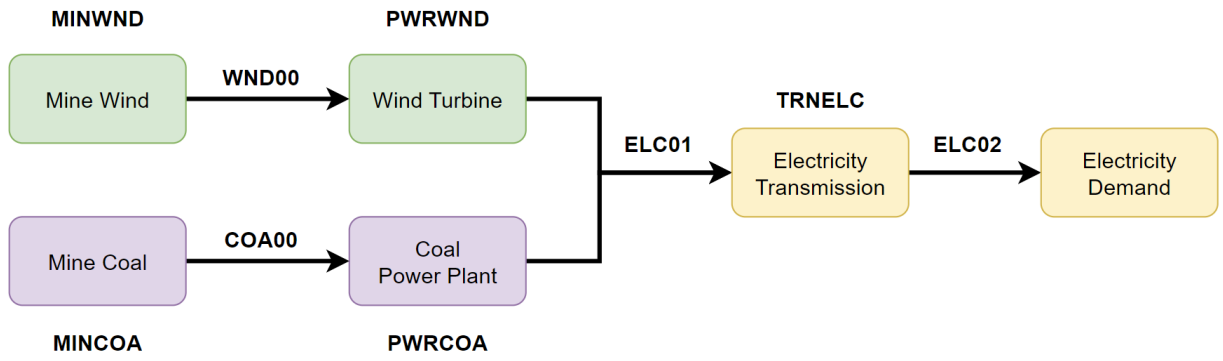
There should now be a file called `template.xlsx` that the user can open and add data to.

1.4.6 Model Validation

Note: In this example, we will use a very simple model instead of the Simplicity demonstration model. This way the user does not need to be familiar with the naming conventions of the model.

Objective

Use `otoole` to validate an input data file. The model we are going to validate is shown below, where the fuel and technology codes are shown in bold face.



1. Download the example datafile

The MathProg datafile describing this model can be found on the examples-validation page. Download the file and save it as `data.txt`

2. Create the Validation File

Create a configuration validation `yaml` file:

```
# on UNIX
$ touch validate.yaml

# on Windows
> type nul > validate.yaml
```

3. Create `FUEL` Codes

Create the fuel codes and descriptions in the validation configuration file:

```
codes:
  fuels:
    'WND': Wind
    'COA': Coal
    'ELC': Electricity
  identifiers:
    '00': Primary Resource
    '01': Intermediate
    '02': End Use
```

4. Create TECHNOLOGY Codes

Add the technology codes to the validation configuration file. Note that the powerplant types are the same codes as the fuels, so there is no need to redefine these codes:

```
codes:
  techs:
    'MIN': Mining
    'PWR': Generator
    'TRN': Transmission
```

5. Create FUEL Schema

Use the defined codes to create a schema for the fuel codes:

```
schema:
  FUEL:
  - name: fuel_name
    items:
    - name: type
      valid: fuels
      position: (1, 3)
    - name: identifier
      valid: identifiers
      position: (4, 5)
```

6. Create TECHNOLOGY Schema

Use the defined codes to create a schema for the technology codes:

```
schema:
  TECHNOLOGY:
  - name: technology_name
    items:
    - name: tech
      valid: techs
      position: (1, 3)
    - name: fuel
      valid: fuels
      position: (4, 6)
```

7. Save changes

The final validation configuration file for this example will look like:

```
codes:
  fuels:
    'WND': Wind
    'COA': Coal
    'ELC': Electricity
```

(continues on next page)

(continued from previous page)

```
  identifiers:
    '00': Primary Resource
    '01': Intermediate
    '02': End Use
  techs:
    'MIN': Mining
    'PWR': Generator
    'TRN': Transmission

schema:
  FUEL:
  - name: fuel_name
    items:
    - name: type
      valid: fuels
      position: (1, 3)
    - name: identifier
      valid: identifiers
      position: (4, 5)
  TECHNOLOGY:
  - name: technology_name
    items:
    - name: tech
      valid: techs
      position: (1, 3)
    - name: fuel
      valid: fuels
      position: (4, 6)
```

8. otoole validate

Use otoole to validate the input data (can be any of a `datafile`, `csv`, or `excel`) against the validation configuration file:

```
$ otoole validate datafile data.txt config.yaml --validate_config validate.yaml

***Beginning validation***

Validating FUEL with fuel_name

^(WND|COA|ELC)(00|01|02)
4 valid names:
WND00, COA00, ELC01, ELC02

Validating TECHNOLOGY with technology_name

^(MIN|PWR|TRN)(WND|COA|ELC)
5 valid names:
MINWND, MINCOA, PWRWND, PWRCOA, TRNELC
```

(continues on next page)

(continued from previous page)

```
***Checking graph structure***
```

Warning: Do not confuse the user configuration file (`config.yaml`) and the validation configuration file (`validate.yaml`). Both configuration files are required for validation functionality.

9. Use `otoole validate` to identify an issue

In the datafile create a new technology that does not follow the specified schema. For example, add the value `ELC03` to the `FUEL` set:

```
set FUEL :=
    WND00
    COA00
    ELC01
    ELC02
    ELC03
```

Running `otoole validate` again will flag this improperly named value. Moreover it will also flag it as an isolated fuel. This means the fuel is unconnected from the model:

```
$ otoole validate datafile data.txt config.yaml --validate_config validate.yaml

***Beginning validation***

Validating FUEL with fuel_name

^(WND|COA|ELC)(00|01|02)
1 invalid names:
ELC03

4 valid names:
WND00, COA00, ELC01, ELC02

Validating TECHNOLOGY with technology_name

^(MIN|PWR|TRN)(WND|COA|ELC)
5 valid names:
MINWND, MINCOA, PWRWND, PWRCOA, TRNELC


***Checking graph structure***

1 'fuel' nodes are isolated:
    ELC03
```

1.5 Python API

otoole also provides a Python API to access all the features available from the command line tool.

1.5.1 Converting between formats

`otoole` currently supports conversion between the following formats:

- Excel
- A folder of CSV files
- GNU MathProg datafile

```
>>> from otoole import convert
>>> convert('my_model.yaml', 'excel', 'csv', 'my_model.xlsx', 'my_model_csvs')
```

See *otoole.convert.convert()* for more details

1.5.2 Converting solver results to a folder of CSV files

The `convert_results` function creates a folder of CSV result files from a CBC, CLP, Gurobi or CPLEX solution file:

```
>>> from otoole import convert_results
>>> convert_results('my_model.yaml', 'cbc', 'csv', 'my_model.sol', 'my_model_csvs',
↪'datafile', 'my_model.dat')
```

See *otoole.convert.convert_results()* for more details

1.5.3 Reading solver results into a dict of Pandas DataFrames

The `read_results` function reads a CBC, CLP, Gurobi or CPLEX solution file into memory:

```
>>> from otoole import read_results
>>> read_results('my_model.yaml', 'cbc', 'csv', 'my_model.sol', 'my_model_csvs',
↪'datafile', 'my_model.dat')
```

See *otoole.convert.read_results()* for more details

1.5.4 Read in data from different Formats

You can use the *otoole.convert.read()* function to read data in from different formats to a Python object. This allows you to then use all the features offered by Python to manipulate the data.

```
>>> from otoole import read
>>> data, defaults = read('my_model.yaml', 'csv', 'my_model_csvs') # read from a folder␣
↪of csv files
>>> data, defaults = read('my_model.yaml', 'excel', 'my_model.xlsx') # read from an␣
↪Excel file
>>> data, defaults = read('my_model.yaml', 'datafile', 'my_model.dat') # read from a GNU␣
↪MathProg datafile
```

1.5.5 Write out data to different Formats

You can use the *otoole.convert.write()* function to write data out to different formats from a Python object.

```
>>> from otoole import read, write
>>> data, defaults = read('my_model.yaml', 'csv', 'my_model_csvs') # read from a folder␣
↪of csv files
>>> write('my_model.yaml', 'excel', 'my_model.xlsx', data, defaults) # write to an Excel␣
↪file
>>> write('my_model.yaml', 'datafile', 'my_model.dat', data, defaults) # write to a GNU␣
↪MathProg datafile
```

1.6 Contributing

Welcome to `otoole` contributor's guide!

This document focuses on getting any potential contributor familiarized with the development processes, but other kinds of contributions are also appreciated.

If you are new to using git or have never collaborated in a project previously, please have a look at contribution-guide.org. Other resources are also listed in the excellent guide created by FreeCodeCamp.

Please notice, all users and contributors are expected to be **open, considerate, reasonable, and respectful**. When in doubt, Python Software Foundation's Code of Conduct is a good reference in terms of behavior guidelines.

1.6.1 Issue Reports

If you experience bugs or general issues with `otoole`, please have a look on the issue tracker. If you don't see anything useful there, please feel free to fire an issue report.

Tip: Please don't forget to include the closed issues in your search. Sometimes a solution was already reported, and the problem is considered **solved**.

New issue reports should include information about your programming environment (e.g., operating system, Python version) and steps to reproduce the problem. Please try also to simplify the reproduction steps to a very minimal example that still illustrates the problem you are facing. By removing other factors, you help us to identify the root cause of the issue.

1.6.2 Documentation Improvements

You can help improve `otoole` docs by making them more readable and coherent, or by adding missing information and correcting mistakes.

`otoole` documentation uses Sphinx as its main documentation compiler. This means that the docs are kept in the same repository as the project code, and that any documentation update is done in the same way was a code contribution.

Our documentation is written in reStructuredText.

Tip: Please notice that the GitHub web interface provides a quick way of propose changes in `otoole`'s files. While this mechanism can be tricky for normal code contributions, it works perfectly fine for contributing to the docs, and can be quite handy.

If you are interested in trying this method out, please navigate to the `docs` folder in the source repository, find which file you would like to propose changes and click in the little pencil icon at the top, to open GitHub's code editor. Once you finish editing the file, please write a message in the form at the bottom of the page describing which changes have you made and what are the motivations behind them and submit your proposal.

When working on documentation changes in your local machine, you can compile them using `tox`:

```
tox -e docs
```

and use Python's built-in web server for a preview in your web browser (`http://localhost:8000`):

```
python3 -m http.server --directory 'docs/_build/html'
```

1.6.3 Code Contributions

`otoole` is built around a command line tool which is written using the Python argparse library. The `otoole.cli` module is a useful place to start when trying to understand how each command works.

The `otoole convert` and `otoole results` commands both use classes which inherit the `otoole.Strategy` class. An `otoole.ReadStrategy` implements functionality to read in data, while an `otoole.WriteStrategy` writes out the target file format. The internal datastore format in `otool` is a dictionary of `pandas.DataFrames`.

Comprehensive unit tests in the `tests` folder provide another way to understand what each of the components does.

Submit an issue

Before you work on any non-trivial code contribution it's best to first create a report in the issue tracker to start a discussion on the subject. This often provides additional considerations and avoids unnecessary work.

Create an environment

Before you start coding, we recommend creating an isolated virtual environment to avoid any problems with your installed Python packages. This can easily be done via either `virtualenv`:

```
virtualenv <PATH TO VENV>
source <PATH TO VENV>/bin/activate
```

or Miniconda:

```
conda create -n otoole python=3 six virtualenv pytest pytest-cov
conda activate otoole
```

Clone the repository

1. Create an user account on GitHub if you do not already have one.
2. Fork the project repository: click on the *Fork* button near the top of the page. This creates a copy of the code under your account on GitHub.
3. Clone this copy to your local disk:

```
git clone git@github.com:YourLogin/otoole.git
cd otoole
```

4. You should run:

```
pip install -U pip setuptools -e .
```

   to be able to import the package under development in the Python REPL.

5. Install `pre-commit`:

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

   `otoole` comes with a lot of hooks configured to automatically help the developer to check the code being written.

Implement your changes

1. Create a branch to hold your changes:

```
git checkout -b my-feature
```

   and start making changes. Never work on the main branch!

2. Start your work on this branch. Don't forget to add docstrings to new functions, modules and classes, especially if they are part of public APIs.
3. Add yourself to the list of contributors in `AUTHORS.rst`.
4. When you're done editing, do:

```
git add <MODIFIED FILES>
git commit
```

   to record your changes in git.

   Please make sure to see the validation messages from `pre-commit` and fix any eventual issues. This should automatically use flake8/black to check/fix the code style in a way that is compatible with the project.

Important: Don't forget to add unit tests and documentation in case your contribution adds an additional feature and is not just a bugfix.

Moreover, writing a descriptive commit message is highly recommended. In case of doubt, you can check the commit history with:

```
git log --graph --decorate --pretty=oneline --abbrev-commit --all
```

to look for recurring communication patterns.

---

5. Please check that your changes don't break any unit tests with:

```
tox
```

   (after having installed `tox` with `pip install tox` or `pipx`).

   You can also use `tox` to run several other pre-configured tasks in the repository. Try `tox -av` to see a list of the available checks.

Submit your contribution

1. If everything works fine, push your local branch to GitHub with:

```
git push -u origin my-feature
```

2. Go to the web page of your fork and click "Create pull request" to send your changes for review.

Find more detailed information in creating a PR. You might also want to open the PR as a draft first and mark it as ready for review after the feedbacks from the continuous integration (CI) system or any required fixes.

We track test coverage using coveralls. You can check the coverage of your PR by clicking on the "details" link in the "Coverage" section of the pull request checks. Try to ensure that your pull requests always increase test coverage.

Troubleshooting

The following tips can be used when facing problems to build or test the package:

1. Make sure to fetch all the tags from the upstream repository. The command `git describe --abbrev=0 --tags` should return the version you are expecting. If you are trying to run CI scripts in a fork repository, make sure to push all the tags. You can also try to remove all the egg files or the complete egg folder, i.e., `.eggs`, as well as the `*.egg-info` folders in the `src` folder or potentially in the root of your project.

2. Sometimes `tox` misses out when new dependencies are added, especially to `setup.cfg` and `docs/requirements.txt`. If you find any problems with missing dependencies when running a command with `tox`, try to recreate the `tox` environment using the `-r` flag. For example, instead of:

```
tox -e docs
```

   Try running:

```
tox -r -e docs
```

3. Make sure to have a reliable `tox` installation that uses the correct Python version (e.g., 3.8+). When in doubt you can run:

```
tox --version
# OR
which tox
```

   If you have trouble and are seeing weird errors upon running `tox`, you can also try to create a dedicated virtual environment with a `tox` binary freshly installed. For example:

```
virtualenv .venv
source .venv/bin/activate
.venv/bin/pip install tox
.venv/bin/tox -e all
```

4. Pytest can drop you in an interactive session in the case an error occurs. In order to do that you need to pass a `--pdb` option (for example by running `tox -- -k <NAME OF THE FALLING TEST> --pdb`). You can also setup breakpoints manually instead of using the `--pdb` option.

1.6.4 Maintainer tasks

Releases

If you are part of the group of maintainers and have correct user permissions on PyPI, the following steps can be used to release a new version for `otoole`:

1. Make sure all unit tests are successful.
2. Tag the current commit on the main branch with a release tag, e.g., `v1.2.3`.
3. Push the new tag to the upstream repository, e.g., `git push upstream v1.2.3`
4. Clean up the `dist` and `build` folders with `tox -e clean` (or `rm -rf dist build`) to avoid confusion with old builds and Sphinx docs.
5. Run `tox -e build` and check that the files in `dist` have the correct version (no `.dirty` or git hash) according to the git tag. Also check the sizes of the distributions, if they are too big (e.g., > 500KB), unwanted clutter may have been accidentally included.
6. Run `tox -e publish -- --repository pypi` and check that everything was uploaded to PyPI correctly.

1.7 License

1.8 Contributors

- Will Usher <https://github.com/willu47>
- Hauke Henke <https://github.com/HauHe>
- Christoph Muschner <https://github.com/chrwm>
- Trevor Barnes <https://github.com/trevorb1>

1.9 Changelog

1.9.1 (Development) Version 1.1.3

- Lock pandas to 2.1.4 or later
- Capital Investment result calculation fixed

1.9.2 Version 1.1.2

- Update zenodo metadata for JOSS

1.9.3 Version 1.1.1

- Fixes CPLEX result processing docs
- Added joss status badge to readme
- Fix Tests on Windows
- Update graphviz install instructions

1.9.4 Version 1.1.0

- Public Python API added to call otoole directly in Python files
- ReadCplex directly reads in CPLEX solution files. Drops the need to transform and sort solution files
- ReadGlpk class added to process GLPK solution files
- Update to Pydantic v2.0
- ReadResultsCbc renamed to ReadWideResults
- Model validation instructions updated in documentation
- The `--input_datafile` argument is deprecated, and the user now must supply the input data to process results
- Locks pandas to <2.1

1.9.5 Version 1.0.4

- Fixed issue with pydantic v2.0.0
- Dropped support for Python 3.8. Otoole now requires Python 3.9 or later

1.9.6 Version 1.0.3

- Improved error message for multiple names mismatches
- Fix for excel pivoting bug (issue 171)
- Fix data type casting issue for floats to ints (issue 167)
- Deprecates calculated field for Result definitions in config file (issue 173)
- Minor documentation updates

1.9.7 Version 1.0.2

- Fix of pandas version in setup.cfg

1.9.8 Version 1.0.1

- Updates to citation file
- Relink to coveralls
- Upgrade to pandas 2.0
- Add issue and PR templates
- Add reading checks between config file and input data

1.9.9 Version 1.0.0

- Requires explicit provision of a user-defined configuration file for otoole to workbook
- Deprecates datapackage fuctionality
- Adds setup command to generate template config.yaml and csv files
- Documentation update
- Bumped pyscaffold to 4.2
- Otoole now requires Python 3.8 or later

1.9.10 Version 0.11.0

- Foundation for user defined configuration
- Fix for issue #101
- Better writing of floating point numbers as text

1.9.11 Version 0.10.0

- Adds support for OSeMOSYS v1.0, Gurobi and CBC

1.9.12 Version 0.9

- Adds support for processing Gurobi solution files
- Better handling of datatypes when converting datapackages
- Fixing bugs on Windows where empty lines were written out during conversion to datapackage

1.9.13 Version 0.8

- Behind-the-scenes reorganisation of code to use *ReadStrategies* and *WriteStrategies* pattern. This enables much cleaner structuring of the code and more reusability of modular blocks.
- Updates to documentation with clearer explaination of how to perform conversions
- Otoole now requires Python 3.7 or later
- Harmonisation of results and pre-processing using the strategies mentioned above
- Bugfixes for #61, #63, #65, #70

1.9.14 Version 0.7

- Adds results processing and conversion of results
- CBC results are transformed into a folder of CSV files
- Missing intermediate results parameters are automatically generated
- Adds command `otoole results cbc csv simplicity.sol ./results --input_datafile simplicity.txt`
- Removes dependency upon PuLP now that amply is available separately on PyPi
- Fixed bug with parameter names >31 characters in converting to Excel and fixed round trip
- Added conversions from Excel to datafile and datapackage to avoid intermediate commands so `otoole convert excel datapackage` <> <> and `otoole convert excel datafile` <> <> are both now legal

1.9.15 Version 0.6

- Fixes bug in writing to datafile where any values that matched the default were ignored
- Adds CLI command to convert to Excel from datapackage e.g. `otoole convert datapackage excel <datapackage.json> <to.xlsx>`
- Uses black code style and uses mypy and black for syntax checking and formatting

1.9.16 Version 0.5

- Add validation of names and fuels in datapackage - Adds `validate` command to the command-line interface - Define a validation config as a YAML file for names

1.9.17 Version 0.4

- Tidy up the command line interface
- Convert to/from SQLite database from/to datapackage
- Remove rotten pygraphviz dependency

1.9.18 Version 0.3

- Create a Tabular Data Package from an OSeMOSYS datafile

1.9.19 Version 0.2

- Visualise a reference energy system from a Tabular Data Package

1.9.20 Version 0.1

- Add CPLEX to csv or CBC solution file conversion script
- Create CSV files in a folder from an excel workbook
- Create a Tabular Data Package from a folder of CSVs
- Create an OSeMOSYS datafile from a Tabular Data Package
- Adds a command line interface to access these tools

1.10 otoole

1.10.1 otoole package

Subpackages

otoole.preprocess package

Submodules

otoole.preprocess.longify_data module

Read in a folder of irregular wide-format csv files and write them out as narrow csvs

otoole.preprocess.longify_data.**check_datatypes**(*df: DataFrame*, *config_details: Dict*, *parameter: str*) → DataFrame

Checks a parameters datatypes

Parameters

- **df** (*pandas.DataFrame*) – The parameter data
- **config_details** (*dict*) – The configuration dictionary
- **parameter** (*str*) – The name of the parameter

otoole.preprocess.longify_data.**check_set_datatype**(*narrow: DataFrame*, *config_details: Dict*, *set_name: str*) → DataFrame

Checks the datatypes of a set_name dataframe

Parameters

- **narrow** (*pandas.DataFrame*) – The set data
- **config_details** (*dict*) – The configuration dictionary
- **set_name** (*str*) – The name of the set

otoole.preprocess.setup module

Module to create template data

otoole.preprocess.setup.**get_config_setup_data**() → Dict[str, Any]

Reads in template config data

otoole.preprocess.setup.**get_csv_setup_data**(*config: Dict[str, Any]*) → Tuple[Dict[str, DataFrame], Dict[str, Any]]

Gets template dataframe data to write as csvs

Parameters

config (*Dict[str,Any]*) – Configuration data to get CSV data to match against

Returns

- **Dict[str** (*pd.DataFrame]*) – Parameters with empty template dataframes
- **Dict[str** (*Any]*) – Default values extracted from config file

otoole.preprocess.validate_config module

Validation methods for the user configuration file.

class otoole.preprocess.validate_config.**UserDefinedParameter**(*, *name: str*, *type: str*, *dtype: str*, *defined_sets: Optional[List[str]] = None*, *indices: Optional[List[str]] = None*, *default: Optional[Union[int, float]] = None*, *calculated: Optional[bool] = None*, *short_name: Optional[str] = None*)

Bases: *UserDefinedValue*

Represents a parameter

classmethod check_dtype(*value*, *info*)

check_dtype_default()

check_index_in_set()

classmethod check_required_inputs(*values*)

model_computed_fields: ClassVar[dict[str, ComputedFieldInfo]] = {}

A dictionary of computed field names and their corresponding *ComputedFieldInfo* objects.

model_config: ClassVar[ConfigDict] = {'extra': 'forbid'}

Configuration for the model, should be a dictionary conforming to [*ConfigDict*][pydantic.config.ConfigDict].

model_fields: ClassVar[dict[str, FieldInfo]] = {'calculated': FieldInfo(annotation=Union[bool, NoneType], required=False), 'default': FieldInfo(annotation=Union[int, float, NoneType], required=False), 'defined_sets': FieldInfo(annotation=Union[List[str], NoneType], required=False), 'dtype': FieldInfo(annotation=str, required=True), 'indices': FieldInfo(annotation=Union[List[str], NoneType], required=False), 'name': FieldInfo(annotation=str, required=True), 'short_name': FieldInfo(annotation=Union[str, NoneType], required=False), 'type': FieldInfo(annotation=str, required=True)}

Metadata about the fields defined on the model, mapping of field names to [*FieldInfo*][pydantic.fields.FieldInfo].

This replaces *Model.__fields__* from Pydantic V1.

class otoole.preprocess.validate_config.**UserDefinedResult**(*, *name: str*, *type: str*, *dtype: str*, *defined_sets: Optional[List[str]] = None*, *indices: Optional[List[str]] = None*, *default: Optional[Union[int, float]] = None*, *calculated: Optional[bool] = None*, *short_name: Optional[str] = None*)

Bases: *UserDefinedValue*

Represents a result

classmethod check_deprecated_values(*values*)

classmethod check_dtype(*value*, *info*)

check_dtype_default()

check_index_in_set()

classmethod check_required_inputs(*values*)

model_computed_fields: ClassVar[dict[str, ComputedFieldInfo]] = {}

A dictionary of computed field names and their corresponding *ComputedFieldInfo* objects.

model_config: ClassVar[ConfigDict] = {'extra': 'forbid'}

Configuration for the model, should be a dictionary conforming to [*ConfigDict*][pydantic.config.ConfigDict].

model_fields: ClassVar[dict[str, FieldInfo]] = {'calculated': FieldInfo(annotation=Union[bool, NoneType], required=False), 'default': FieldInfo(annotation=Union[int, float, NoneType], required=False), 'defined_sets': FieldInfo(annotation=Union[List[str], NoneType], required=False), 'dtype': FieldInfo(annotation=str, required=True), 'indices': FieldInfo(annotation=Union[List[str], NoneType], required=False), 'name': FieldInfo(annotation=str, required=True), 'short_name': FieldInfo(annotation=Union[str, NoneType], required=False), 'type': FieldInfo(annotation=str, required=True)}

Metadata about the fields defined on the model, mapping of field names to [*FieldInfo*][pydantic.fields.FieldInfo].

This replaces *Model.__fields__* from Pydantic V1.

class otoole.preprocess.validate_config.**UserDefinedSet**(*, *name: str*, *type: str*, *dtype: str*, *defined_sets: Optional[List[str]] = None*, *indices: Optional[List[str]] = None*, *default: Optional[Union[int, float]] = None*, *calculated: Optional[bool] = None*, *short_name: Optional[str] = None*)

Bases: *UserDefinedValue*

Represents a set

classmethod check_dtype(*value*, *info*)

model_computed_fields: ClassVar[dict[str, ComputedFieldInfo]] = {}

A dictionary of computed field names and their corresponding *ComputedFieldInfo* objects.

model_config: ClassVar[ConfigDict] = {'extra': 'forbid'}

Configuration for the model, should be a dictionary conforming to [*ConfigDict*][pydantic.config.ConfigDict].

model_fields: ClassVar[dict[str, FieldInfo]] = {'calculated': FieldInfo(annotation=Union[bool, NoneType], required=False), 'default': FieldInfo(annotation=Union[int, float, NoneType], required=False), 'defined_sets': FieldInfo(annotation=Union[List[str], NoneType], required=False), 'dtype': FieldInfo(annotation=str, required=True), 'indices': FieldInfo(annotation=Union[List[str], NoneType], required=False), 'name': FieldInfo(annotation=str, required=True), 'short_name': FieldInfo(annotation=Union[str, NoneType], required=False), 'type': FieldInfo(annotation=str, required=True)}

Metadata about the fields defined on the model, mapping of field names to [*FieldInfo*][pydantic.fields.FieldInfo].

This replaces *Model.__fields__* from Pydantic V1.

class otoole.preprocess.validate_config.**UserDefinedValue**(*, *name: str*, *type: str*, *dtype: str*, *defined_sets: Optional[List[str]] = None*, *indices: Optional[List[str]] = None*, *default: Optional[Union[int, float]] = None*, *calculated: Optional[bool] = None*, *short_name: Optional[str] = None*)

Bases: `BaseModel`

Represents any user defined value

calculated: `Optional[bool]`

classmethod check_name_for_numbers(*value*)

classmethod check_name_for_spaces(*value*)

classmethod check_name_for_special_chars(*value*)

classmethod check_name_length(*values*)

classmethod check_param_type(*value*, *info*)

default: `Optional[Union[int, float]]`

defined_sets: `Optional[List[str]]`

dtype: `str`

indices: `Optional[List[str]]`

model_computed_fields: `ClassVar[dict[str, ComputedFieldInfo]] = {}`

A dictionary of computed field names and their corresponding *ComputedFieldInfo* objects.

model_config: `ClassVar[ConfigDict] = {'extra': 'forbid'}`

Configuration for the model, should be a dictionary conforming to [*ConfigDict*][pydantic.config.ConfigDict].

model_fields: `ClassVar[dict[str, FieldInfo]] = {'calculated': FieldInfo(annotation=Union[bool, NoneType], required=False), 'default': FieldInfo(annotation=Union[int, float, NoneType], required=False), 'defined_sets': FieldInfo(annotation=Union[List[str], NoneType], required=False), 'dtype': FieldInfo(annotation=str, required=True), 'indices': FieldInfo(annotation=Union[List[str], NoneType], required=False), 'name': FieldInfo(annotation=str, required=True), 'short_name': FieldInfo(annotation=Union[str, NoneType], required=False), 'type': FieldInfo(annotation=str, required=True)}`

Metadata about the fields defined on the model, mapping of field names to [*FieldInfo*][pydantic.fields.FieldInfo].

This replaces *Model.__fields__* from Pydantic V1.

name: `str`

short_name: Optional[str]

type: str

Module contents

otoole.results package

Submodules

otoole.results.result_package module

class otoole.results.result_package.**ResultsPackage**(*data: Dict[str, DataFrame]*, *input_data: Optional[Dict[str, DataFrame]] = None*)

Bases: Mapping

A package of OSeMOSYS results

Internal data structure is a dictionary of pandas DataFrames

A crude caching function is implemented whereby calculated results are stored in the internal data structure so that each result is only calculated once.

Parameters

- **data** (*dict*) – A dictionary of results data
- **input_data** (*dict, default=None*) – Dictionary of input data

accumulated_new_capacity() → DataFrame

AccumulatedNewCapacity

Parameters

- **operational_life** (*pandas.DataFrame*) –
- **new_capacity** (*pandas.DataFrame*) –
- **year** (*pandas.Index*) –

Notes

From the formulation:

```
r~REGION, t~TECHNOLOGY, y~YEAR,
sum{yy in YEAR: y-yy < OperationalLife[r,t] && y-yy>=0}
    NewCapacity[r,t,yy] ~VALUE;
```

annual_emissions() → DataFrame

Calculates the annual emissions

Annual Emission are found by multiplying the emission activity ratio (emissions per unit activity) by the rate of activity and the yearsplit

Notes

From the formulation:

```
sum{t in TECHNOLOGY, l in TIMESLICE, m in MODE_OF_OPERATION:
        EmissionActivityRatio[r,t,e,m,y]<>0}
    RateOfActivity[r,l,t,m,y] * EmissionActivityRatio[r,t,e,m,y]
        * YearSplit[l,y]~VALUE;
```

annual_fixed_operating_cost() → DataFrame

Compute AnnualFixedOperatingCost result

Notes

From the formulation:

```
r~REGION, t~TECHNOLOGY, y~YEAR,
FixedCost[r,t,y] *
((sum{yy in YEAR: y-yy < OperationalLife[r,t] && y-yy>=0}
    NewCapacity[r,t,yy]) + ResidualCapacity[r,t,y]) ~VALUE;
```

annual_technology_emission_by_mode() → DataFrame

AnnualTechnologyEmissionByMode

Notes

From the formulation:

```
r~REGION, t~TECHNOLOGY, e~EMISSION, m~MODE_OF_OPERATION, y~YEAR,
sum{l in TIMESLICE: EmissionActivityRatio[r,t,e,m,y] <> 0}
    EmissionActivityRatio[r,t,e,m,y] * RateOfActivity[r,l,t,m,y]
        * YearSplit[l,y]
```

annual_technology_emissions() → DataFrame

Calculates results `AnnualTechnologyEmission`

Notes

From the formulation:

```
REGION, TECHNOLOGY, EMISSION, YEAR,
sum{l in TIMESLICE, m in MODE_OF_OPERATION:
    EmissionActivityRatio[r,t,e,m,y]<>0}
EmissionActivityRatio[r,t,e,m,y] * RateOfActivity[r,l,t,m,y]
    * YearSplit[l,y];
```

annual_variable_operating_cost() → DataFrame

AnnualVariableOperatingCost

Notes

From the formulation:

```
r~REGION, t~TECHNOLOGY, y~YEAR,
sum{m in MODE_OF_OPERATION, l in TIMESLICE}
    RateOfActivity[r,l,t,m,y]
    * YearSplit[l,y]
    * VariableCost[r,t,m,y] ~VALUE;
```

capital_investment() → DataFrame

CapitalInvestment

Notes

From the formulation:

```
r~REGION, t~TECHNOLOGY, y~YEAR,
CapitalCost[r,t,y] * NewCapacity[r,t,y] * CapitalRecoveryFactor[r,t] *
PvAnnuity[r,t] ~VALUE;
```

property data: Dict[str, DataFrame]

View the results dictionary

demand() → DataFrame

Demand

Notes

From the formulation:

```
r~REGION, l~TIMESLICE, f~FUEL, y~YEAR,
SpecifiedAnnualDemand[r,f,y] * SpecifiedDemandProfile[r,f,l,y] ~VALUE;
```

discounted_tech_emis_pen() → DataFrame

DiscountedTechnologyEmissionsPenalty

Notes

From the formulation:

```
DiscountedTechnologyEmissionsPenalty[r,t,y] :=
EmissionActivityRatio[r,t,e,m,y] * RateOfActivity[r,l,t,m,y] *
YearSplit[l,y] * EmissionsPenalty[r,e,y] / DiscountFactorMid[r,y]
```

get_unique_values_from_index(*dataframes: List*, *name: str*) → List

Utility function to extract list of unique values

Extract unique values from the same index of the passed dataframes

production_by_technology() → DataFrame

ProductionByTechnology

Notes

From the formulation:

```
r~REGION, l~TIMESLICE, t~TECHNOLOGY, f~FUEL, y~YEAR,
sum{m in MODE_OF_OPERATION: OutputActivityRatio[r,t,f,m,y] <> 0}
    RateOfActivity[r,l,t,m,y] * OutputActivityRatio[r,t,f,m,y]
    * YearSplit[l,y] ~VALUE;
```

production_by_technology_annual() → DataFrame

Aggregates production by technology to the annual level

rate_of_product_technology() → DataFrame

Sums up mode of operation for rate of production

Notes

From the formulation:

```
r~REGION, l~TIMESLICE, t~TECHNOLOGY, f~FUEL, y~YEAR,
sum{m in MODE_OF_OPERATION: OutputActivityRatio[r,t,f,m,y] <> 0}
    RateOfActivity[r,l,t,m,y] * OutputActivityRatio[r,t,f,m,y]~VALUE;
```

rate_of_production_tech_mode() → DataFrame

RateOfProductionByTechnologyByMode

Notes

From the formulation:

```
r~REGION, l~TIMESLICE, t~TECHNOLOGY, m~MODE_OF_OPERATION, f~FUEL, y~YEAR,
RateOfActivity[r,l,t,m,y] * OutputActivityRatio[r,t,f,m,y]~VALUE;
```

rate_of_use_by_technology() → DataFrame

RateOfUseByTechnology

Notes

From the formulation:

```
r~REGION, l~TIMESLICE, t~TECHNOLOGY, f~FUEL, y~YEAR,
sum{m in MODE_OF_OPERATION: InputActivityRatio[r,t,f,m,y]<>0}
    RateOfActivity[r,l,t,m,y] * InputActivityRatio[r,t,f,m,y]~VALUE;
```

rate_of_use_by_technology_by_mode() → DataFrame

RateOfUseByTechnologyByMode

Notes

From the formulation:

```
r~REGION, l~TIMESLICE, t~TECHNOLOGY, m~MODE_OF_OPERATION, f~FUEL, y~YEAR,
RateOfActivity[r,l,t,m,y] * InputActivityRatio[r,t,f,m,y]~VALUE;
```

property result_cache: Dict[str, DataFrame]

property result_mapper: Dict[str, DataFrame]

total_annual_tech_activity_mode() → DataFrame

TotalAnnualTechnologyActivityByMode

Notes

From the formulation:

```
r~REGION, t~TECHNOLOGY, m~MODE_OF_OPERATION, y~YEAR,
sum{l in TIMESLICE}
    RateOfActivity[r,l,t,m,y] * YearSplit[l,y]~VALUE;
```

total_capacity_annual() → DataFrame

TotalCapacityAnnual

Notes

From the formulation:

```
r~REGION, t~TECHNOLOGY, y~YEAR,
ResidualCapacity[r,t,y] +
(sum{yy in YEAR: y-yy < OperationalLife[r,t] && y-yy>=0}
    NewCapacity[r,t,yy])~VALUE;
```

total_discounted_cost() → DataFrame

TotalDiscountedCost

Notes

From the formulation:

```
r~REGION, y~YEAR,
sum{t in TECHNOLOGY}
(
    (
        (
            (
                sum{yy in YEAR: y-yy < OperationalLife[r,t] && y-yy>=0}
                NewCapacity[r,t,yy]
            )
            + ResidualCapacity[r,t,y]
        )
```

(continues on next page)

(continued from previous page)

```
        * FixedCost[r,t,y]
        + sum{l in TIMESLICE, m in MODEperTECHNOLOGY[t]}
        RateOfActivity[r,l,t,m,y] * YearSplit[l,y] * VariableCost[r,t,m,y]
    )
    / (DiscountFactorMid[r,y])
    + CapitalCost[r,t,y] * NewCapacity[r,t,y] * CapitalRecoveryFactor[r,t] *␣
↪PvAnnuity[r,t] / (DiscountFactor[r,y])
    + DiscountedTechnologyEmissionsPenalty[r,t,y] - DiscountedSalvageValue[r,t,
↪y])
    + sum{s in STORAGE}
    (
        CapitalCostStorage[r,s,y] * NewStorageCapacity[r,s,y] /␣
↪(DiscountFactorStorage[r,s,y])
        - CapitalCostStorage[r,s,y] * NewStorageCapacity[r,s,y] /␣
↪(DiscountFactorStorage[r,s,y]
    )
) ~VALUE;
```

total_tech_model_period_activity() → DataFrame

TotalTechnologyModelPeriodActivity

Notes

From the formulation:

```
ResultsPath & "/TotalTechnologyModelPeriodActivity.csv":
r~REGION, t~TECHNOLOGY,
sum{l in TIMESLICE, m in MODE_OF_OPERATION, y in YEAR}
    RateOfActivity[r,l,t,m,y]*YearSplit[l,y]~VALUE;
```

total_technology_annual_activity() → DataFrame

TotalTechnologyAnnualActivity

Notes

From the formulation:

```
ResultsPath & "/TotalTechnologyAnnualActivity.csv":
r~REGION, t~TECHNOLOGY, y~YEAR,
sum{l in TIMESLICE, m in MODE_OF_OPERATION}
    RateOfActivity[r,l,t,m,y] * YearSplit[l,y]~VALUE;
```

use_by_technology() → DataFrame

UseByTechnology

Notes

From the formulation:

```
r~REGION, l~TIMESLICE, t~TECHNOLOGY, f~FUEL, y~YEAR,
sum{m in MODE_OF_OPERATION}
    RateOfActivity[r,l,t,m,y]
    * InputActivityRatio[r,t,f,m,y]
    * YearSplit[l,y]~VALUE;
```

otoole.results.result_package.**capital_recovery_factor**(*regions: List*, *technologies: List*, *discount_rate_idv: DataFrame*, *operational_life: DataFrame*) → DataFrame

Calculates the capital recovery factor

Parameters

- **regions** (*list*) –
- **technologies** (*list*) –
- **discount_rate_idv** (*pd.DataFrame*) –
- **operational_life** (*pd.DataFrame*) –

Notes

From the formulation:

```
param CapitalRecoveryFactor{r in REGION, t in TECHNOLOGY} :=
        (1 - (1 + DiscountRateIdv[r,t])^(-1))/(1 - (1 + DiscountRateIdv[r,t])^(-
↪(OperationalLife[r,t])));
```

otoole.results.result_package.**discount_factor**(*regions: List*, *years: List*, *discount_rate: DataFrame*, *adj: float = 0.0*) → DataFrame

DiscountFactor

Parameters

- **regions** (*list*) –
- **years** (*list*) –
- **discount_rate** (*pd.DataFrame*) –
- **adj** (*float, default=0.0*) – Adjust to beginning of the year (default), mid year (0.5) or end year (1.0)

Notes

From the formulation:

```
param DiscountFactor{r in REGION, y in YEAR} :=
        (1 + DiscountRate[r]) ^ (y - min{yy in YEAR} min(yy) + 0.0);

param DiscountFactorMid{r in REGION, y in YEAR} :=
        (1 + DiscountRate[r]) ^ (y - min{yy in YEAR} min(yy) + 0.5);
```

otoole.results.result_package.**discount_factor_storage**(*regions: List*, *storages: List*, *years: List*, *discount_rate_storage: DataFrame*, *adj: float = 0.0*) → DataFrame

DiscountFactorStorage

Parameters

- **regions** (*list*) –
- **storages** (*list*) –
- **years** (*list*) –
- **discount_rate_storage** (*pd.DataFrame*) –
- **adj** (*float, default=0.0*) – Adjust to beginning of the year (default), mid year (0.5) or end year (1.0)

Notes

From the formulation:

```
param DiscountFactorStorage{r in REGION, s in STORAGE, y in YEAR} :=
        (1 + DiscountRateStorage[r,s]) ^ (y - min{yy in YEAR} min(yy) + 0.0);
```

otoole.results.result_package.**pv_annuity**(*regions: List*, *technologies: List*, *discount_rate: DataFrame*, *operational_life: DataFrame*) → DataFrame

Calculates the present value of an annuity

Parameters

- **regions** (*list*) –
- **technologies** (*list*) –
- **discount_rate** (*pd.DataFrame*) –
- **operational_life** (*pd.DataFrame*) –

Notes

From the formulation:

```
param PvAnnuity{r in REGION, t in TECHNOLOGY} :=
        (1 - (1 + DiscountRate[r])^(-(OperationalLife[r,t]))) * (1 +␣
↪DiscountRate[r]) / DiscountRate[r];
```

otoole.results.results module

class otoole.results.results.**ReadCbc**(*user_config: Dict[str, Dict]*)

Bases: *ReadWideResults*

Read a CBC solution file into memory

Parameters

- **user_config** (*Dict[str, Dict]*) –
- **results_config** (*Dict[str, Dict]*) –

class otoole.results.results.**ReadCplex**(*user_config: Dict[str, Dict]*)

Bases: *ReadWideResults*

Read a CPLEX solution file into memeory

class otoole.results.results.**ReadGlpk**(*user_config: Dict[str, Dict]*, *glpk_model: Union[str, TextIO]*)

Bases: *ReadWideResults*

Reads a GLPK Solution file into memory

Parameters

- **user_config** (*Dict[str, Dict]*) –
- **glpk_model** (*Union[str, TextIO]*) – Path to GLPK model file. Can be created using the *–wglp* flag.

read_model(*file_path: Union[str, TextIO]*) → DataFrame

Reads in a GLPK Model File

Parameters

file_path (*Union[str, TextIO]*) – Path to GLPK model file. Can be created using the *–wglp* flag.

Returns

- *pd.DataFrame* – ID NUM NAME INDEX
- *0 i 1 CAa4_Constraint_Capacity "SIMPLICITY,ID,BACKSTOP1,2015"*
- *1 j 2 NewCapacity "SIMPLICITY,WINDPOWER,2039"*

Notes

-> GENERAL LAYOUT OF SOLUTION FILE

n p NAME # p = problem instance n z NAME # z = objective function n i ROW NAME # i = constraint name, ROW is the row ordinal number n j COL NAME # j = variable name, COL is the column ordinal number

read_solution(*file_path: Union[str, TextIO]*) → Tuple[Dict[str, Union[str, float]], DataFrame]

Reads a GLPK solution file

Parameters

file_path (*Union[str, TextIO]*) – Path to GLPK solution file. Can be created using the *–write* flag

Returns

- *Tuple[Dict[str,Union[str, float]], pd.DataFrame]* – Dict[str,Union[str, float]] -> Problem name, status, and objective value pd.DataFrame -> Variables and constraints
- **{"name"** (*"osemosys", "status":"OPTIMAL", "objective":4497.31976}*) – ID NUM STATUS PRIM DUAL
- *0 i 1 b 5 0*
- *1 j 2 l 0 2*

Notes

-> ROWS IN SOLUTION FILE

i ROW ST PRIM DUAL

ROW is the ordinal number of the row ST is one of: - b = inactive constraint; - l = inequality constraint active on its lower bound; - u = inequality constraint active on its upper bound; - f = active free (unounded) row; - s = active equality constraint. PRIM specifies the row primal value (float) DUAL specifies the row dual value (float)

-> COLUMNS IN SOLUTION FILE

j COL ST PRIM DUAL

COL specifies the column ordinal number ST contains one of the following lower-case letters that specifies the column status in the basic solution: - b = basic variable - l = non-basic variable having its lower bound active - u = non-basic variable having its upper bound active - f = non-basic free (unbounded) variable - s = non-basic fixed variable. PRIM field contains column primal value (float) DUAL field contains the column dual value (float)

class otoole.results.results.**ReadGurobi**(*user_config: Dict[str, Dict]*)

Bases: *ReadWideResults*

Read a Gurobi solution file into memory

class otoole.results.results.**ReadResults**(*user_config: Dict[str, Dict]*)

Bases: *ReadStrategy*

calculate_results(*available_results: Dict[str, DataFrame]*, *input_data: Dict[str, DataFrame]*) → Dict[str, DataFrame]

Populates the results with calculated values using input data

abstract get_results_from_file(*filepath*, *input_data*)

read(*filepath: Union[str, TextIO]*, ***kwargs*) → Tuple[Dict[str, DataFrame], Dict[str, Any]]

Read a solution file from `filepath` and process using `input_data`

Parameters

- **filepath** (*str, TextIO*) – A path name or file buffer pointing to the solution file
- **input_data** (*dict, default=None*) – dict of dataframes

Returns
A tuple containing dict of pandas.DataFrames and a dict of default_values

Return type
tuple

class otoole.results.results.**ReadWideResults**(*user_config: Dict[str, Dict]*)

Bases: *ReadResults*

get_results_from_file(*filepath*, *input_data*)

otoole.results.results.**check_duplicate_index**(*df: DataFrame*, *columns: List*, *index: List*) → DataFrame

Catches pandas error when there are duplicate column indices

otoole.results.results.**check_for_duplicates**(*index: List*) → bool

otoole.results.results.**identify_duplicate**(*index: List*) → Union[int, bool]

otoole.results.results.**rename_duplicate_column**(*index: List*) → List

Module contents

otoole.visualise package

Submodules

otoole.visualise.res module

Visualise the reference energy system

otoole.visualise.res.**add_fuel**(*package_rows: List[List]*) → List[Tuple[str, Dict]]

Add fuel nodes

Parameters
package_rows (*list of dict*) –

Returns
A list of node names along with a dict of node attributes

Return type
list of tuple

otoole.visualise.res.**build_graph**(*nodes: List[Tuple[str, Dict]]*, *edges: List[Tuple[str, str, Dict]]*) → DiGraph

Builds the graph using networkx

Parameters

- **nodes** (*list*) – A list of node tuples
- **edges** (*list*) – A list of edge tuples

Returns
A directed graph representing the reference energy system

Return type
networkx.DiGraph

otoole.visualise.res.**create_graph**(*input_data: Dict[str, DataFrame]*)

Creates a graph of technologies and fuels

Parameters
input_data (*Dict[str, pd.DataFrame]*) – Internal datastore for otoole input data

Returns
networkx.DiGraph

Return type
graph

otoole.visualise.res.**create_res**(*input_data: Dict[str, DataFrame]*, *path_to_resfile: str*)

Create a reference energy system diagram

Parameters

- **input_data** (*Dict[str, pd.DataFrame]*) – Internal datastore for otoole input data
- **path_to_resfile** (*str*) – The path to the image file to be created

otoole.visualise.res.**draw_graph**(*graph*, *path_to_resfile*)

Layout the graph and write it to disk

Uses pygraphviz to set some graph attributes, layout the graph and write it to disk

Parameters
path_to_resfile (*str*) – The file path of the PNG image file that will be created

otoole.visualise.res.**extract_edges**(*package_rows: List[Dict]*, *from_column: str*, *to_column: str*, *parameter_name: str*, *directed: bool = True*) → List[Tuple[str, str, Dict]]

Add edges from a Tabular Data Table

Parameters

- **package_rows** (*list of dict*) –
- **from_column** (*str*) – The name of the column to use as source of the edge
- **to_column** (*str*) – The name of the column to use as a destination of the edge
- **parameter_name** (*str*) – The name of the parameter
- **directed** (*bool, default=True*) – Specifies whether the edge should have an arrow or not

Returns
A list of edges with from/to nodes names and edge attributes

Return type
list of tuple

otoole.visualise.res.**extract_nodes**(*package_rows: List[List]*, *node_type='technology'*, *color='red'*, *shape='circle'*) → List[Tuple[str, Dict]]

Add nodes from a Tabular Data Table

Parameters

- **package_rows** (*list of dict*) –
- **node_type** (*str, default='technology'*) –
- **color** (*str, default='red'*) –
- **shape** (*str, default='circle'*) –

Returns
A list of nodes with attributes

Return type
list of tuple

Module contents

Visualise different aspects of an OSeMOSYS model and data

Provides the following commands:

```
otoole viz res <input_data_format> <path_to_input_data> <path_to_res_image> <path_to_
↪user_config>
```

`otoole viz res` generates an image of the reference energy system

otoole.visualise.**create_res**(*input_data: Dict[str, DataFrame]*, *path_to_resfile: str*)

Create a reference energy system diagram

Parameters

- **input_data** (*Dict[str, pd.DataFrame]*) – Internal datastore for otoole input data
- **path_to_resfile** (*str*) – The path to the image file to be created

Submodules

otoole.cli module

Provides a command line interface to the `otoole` package

The key functions are **convert**, **cplex** and **viz**.

The `convert` command allows convertion of multiple different OSeMOSYS input formats including from/to csv, an AMPL format datafile, a folder of CSVs, an Excel workbook with one tab per parameter, an SQLite database

The `cplex` command provides access to scripts which transform and process a CPLEX solution file into a format which is more readily processed - either to CBC or CSV format.

The `validate` command checks the technology and fuel names against a standard or user defined configuration file.

The **viz** command allows you to produce a Reference Energy System diagram

Example

Ask for help on the command line:

```
>>> $ otoole --help
usage: otoole [-h] [--verbose] [--version] {convert,cplex,validate,viz} ...

otoole: Python toolkit of OSeMOSYS users

positional arguments:
{convert,cplex,validate,viz}
    convert             Convert from one input format to another
    cplex               Process a CPLEX solution file
    validate            Validate an OSeMOSYS model
    viz                 Visualise the model

optional arguments:
-h, --help            show this help message and exit
--verbose, -v         Enable debug mode
--version, -V         The version of otoole
```

class otoole.cli.**DeprecateAction**(*option_strings*, *dest*, *nargs=None*, *const=None*, *default=None*, *type=None*, *choices=None*, *required=False*, *help=None*, *metavar=None*)

Bases: Action

otoole.cli.**data2res**(*args*)

Get input data and call res creation.

otoole.cli.**get_parser**()

otoole.cli.**main**()

otoole.cli.**setup**(*args*)

Creates template data

otoole.cli.**validate_model**(*args*)

otoole.convert module

This module implements the public API of the otoole package

Use the otoole `convert` function to convert between different file formats. Import the convert function from the otoole package:

```
>>> from otoole import convert
>>> convert('config.yaml', 'excel', 'datafile', 'input.xlsx', 'output.dat')
```

otoole.convert.**convert**(*config*, *from_format*, *to_format*, *from_path*, *to_path*, *write_defaults=False*, *keep_whitespace=False*) → bool

Convert OSeMOSYS data from/to datafile, csv and Excel formats

Parameters

- **config** (*str*) – Path to config file
- **from_format** (*str*) – Available options are 'datafile', 'datapackage', 'csv' and 'excel'

- **to_format** (*str*) – Available options are 'datafile', 'datapackage', 'csv' and 'excel'
- **from_path** (*str*) – Path to destination file (if datafile or excel) or folder (csv or datapackage)
- **write_defaults** (*bool, default: False*) – Write default values to CSVs
- **keep_whitespace** (*bool, default: False*) – Keep whitespace in CSVs

Returns
True if conversion was successful, False otherwise

Return type
bool

otoole.convert.**convert_results**(*config: str*, *from_format: str*, *to_format: str*, *from_path: str*, *to_path: str*, *input_format: str*, *input_path: str*, *write_defaults: bool = False*, *glpk_model: Optional[str] = None*) → bool

Post-process results from a CBC, CPLEX, Gurobi, or GLPK solution file into CSV format

Parameters

- **config** (*str*) – Path to config file
- **from_format** (*str*) – Available options are 'cbc', 'cplex' and 'gurobi'
- **to_format** (*str*) – Available options are 'csv'
- **from_path** (*str*) – Path to cbc, cplex or gurobi solution file
- **to_path** (*str*) – Path to destination folder
- **input_format** (*str*) – Format of input data. Available options are 'datafile', 'csv' and 'excel'
- **input_path** (*str*) – Path to input data
- **write_defaults** (*bool*) – Write default values to CSVs
- **glpk_model** (*str*) – Path to `*.glp` model file

Returns
True if conversion was successful, False otherwise

Return type
bool

otoole.convert.**read**(*config: str*, *from_format: str*, *from_path: str*, *keep_whitespace: bool = False*) → Tuple[Dict[str, DataFrame], Dict[str, float]]

Read OSeMOSYS data from datafile, csv or Excel formats

Parameters

- **config** (*str*) – Path to config file
- **from_format** (*str*) – Available options are 'datafile', 'csv', 'excel' and 'datapackage' [deprecated]
- **from_path** (*str*) – Path to source file (if datafile or excel) or folder (csv)
- **keep_whitespace** (*bool, default: False*) – Keep whitespace in source files

Returns
Dictionary of parameter and set data and dictionary of default values

Return type
Tuple[dict[str, pd.DataFrame], dict[str, float]]

otoole.convert.**read_results**(*config: str*, *from_format: str*, *from_path: str*, *input_format: str*, *input_path: str*, *glpk_model: Optional[str] = None*) → Tuple[Dict[str, DataFrame], Dict[str, float]]

Read OSeMOSYS results from CBC, GLPK, Gurobi, or CPLEX results files

Parameters

- **config** (*str*) – Path to config file
- **from_format** (*str*) – Available options are 'cbc', 'gurobi', 'cplex', and 'glpk'
- **from_path** (*str*) – Path to source file (if datafile or excel) or folder (csv)
- **input_format** (*str*) – Format of input data. Available options are 'datafile', 'csv' and 'excel'
- **input_path** (*str*) – Path to input data
- **glpk_model** (*str*) – Path to `*.glp` model file

Returns
Dictionary of parameter and set data and dictionary of default values

Return type
Tuple[dict[str, pd.DataFrame], dict[str, float]]

otoole.convert.**write**(*config: str*, *to_format: str*, *to_path: str*, *inputs*, *default_values: Optional[Dict[str, float]] = None*) → bool

Write OSeMOSYS data to datafile, csv or Excel formats

Parameters

- **config** (*str*) – Path to config file
- **to_format** (*str*) – Available options are 'datafile', 'csv', 'excel' and 'datapackage' [deprecated],
- **to_path** (*str*) – Path to destination file (if datafile or excel) or folder (csv))
- **inputs** (*dict[str, pd.DataFrame]*) – Dictionary of pandas data frames to write
- **default_values** (*dict[str, float], default: None*) – Dictionary of default values to write to datafile

otoole.exceptions module

exception otoole.exceptions.**OtooleConfigFileError**(*message: str*)

Bases: *OtooleException*

Config file validation error

Parameters
message (*str*) – Message to display to users

exception otoole.exceptions.**OtooleDeprecationError**(*resource*, *message*)

Bases: *OtooleException*

New version of otoole does drops this feature support

Parameters

- **resource** (*str*) – Name of the resource which is invalid
- **message** (*str*) – Error message

exception otoole.exceptions.**OtooleError**(*resource*, *message*)

Bases: *OtooleException*

General purpose error

Parameters

- **resource** (*str*) – Name of the resource which is invalid
- **message** (*str*) – Error message

exception otoole.exceptions.**OtooleExcelNameLengthError**(*name: str*, *message: str = 'Parameter name must be less than 31 characters when writing to Excel'*)

Bases: *OtooleException*

Invalid tab name for writing to Excel.

exception otoole.exceptions.**OtooleException**

Bases: Exception

Base class for all otoole exceptions.

exception otoole.exceptions.**OtooleIndexError**(*resource*, *config_indices*, *data_indices*)

Bases: *OtooleException*

Index data not consistent between data and config file

Parameters

- **resource** (*str*) – Name of the resource which is invalid
- **config_indices** (*List[str]*) – Indices from config file
- **data_indices** (*List[str]*) – Indices from input data

exception otoole.exceptions.**OtooleNameMismatchError**(*name: Union[List, str]*)

Bases: *OtooleException*

Names not consistent between read in data and config file

exception otoole.exceptions.**OtooleRelationError**(*resource*, *foreign_resource*, *message*)

Bases: *OtooleException*

Relations between input data is not correct

Parameters

- **resource** (*str*) – Name of the resource which is invalid
- **foreign_resource** (*str*) – Name of the resource which is invalid
- **message** (*str*) – Error message

exception otoole.exceptions.**OtooleSetupError**(*resource*, *message='Data already exists. Delete file/directory or pass the --overwrite flag'*)

Bases: *OtooleException*

Setup data already exists

Parameters

- **resource** (*str*) – Name of the resource which is invalid
- **message** (*str*) – Error message

exception otoole.exceptions.**OtooleValidationError**(*resource*, *message*)

Bases: *OtooleException*

Input data is invalid

Parameters

- **resource** (*str*) – Name of the resource which is invalid
- **message** (*str*) – Error message

otoole.input module

The `input` module allows you to access the conversion routines programmatically

To use the routines, you need to instantiate a `ReadStrategy` and a `WriteStrategy` relevant for the format of the input and output data. You then pass these to a `Context`.

Example

Convert an in-memory dictionary of pandas DataFrames containing OSeMOSYS parameters to an Excel spreadsheet:

```
>>> from otoole import ReadMemory
>>> from otoole import WriteExcel
>>> from otoole import Context
>>> reader = ReadMemory(parameters)
>>> writer = WriteExcel()
>>> converter = Context(read_strategy=reader, write_strategy=writer)
>>> converter.convert('.', 'osemosys_to_excel.xlsx')
```

Convert a GNUMathProg datafile to a folder of CSV files:

```
>>> from otoole import ReadDataFile
>>> from otoole import WriteCsv
>>> from otoole import Context
>>> reader = ReadDataFile()
>>> writer = WriteCsv()
>>> converter = Context(read_strategy=reader, write_strategy=writer)
>>> converter.convert('my_datafile.txt', 'folder_of_csv_files')
```

class otoole.input.**Context**(*read_strategy: ReadStrategy*, *write_strategy: WriteStrategy*)

Bases: `object`

The Context defines the interface of interest to clients.

convert(*input_filepath: str*, *output_filepath: str*, ***kwargs: Dict*)

Converts from file `input_filepath` to file `output_filepath`

Parameters

- **input_filepath** (*str*) –
- **output_filepath** (*str*) –

property read_strategy: *ReadStrategy*

The Context maintains a reference to one of the Strategy objects. The Context does not know the concrete class of a strategy. It should work with all strategies via the Strategy interface.

property write_strategy: *WriteStrategy*

The Context maintains a reference to one of the Strategy objects. The Context does not know the concrete class of a strategy. It should work with all strategies via the Strategy interface.

class otoole.input.**ReadStrategy**(*user_config: Dict[str, Dict]*)

Bases: *Strategy*

The Strategy interface declares operations common to all reading formats.

The Context uses this interface to call the algorithm defined by Concrete Strategies.

abstract read(*filepath: Union[str, TextIO]*, ***kwargs*) → Tuple[Dict[str, DataFrame], Dict[str, Any]]

Reads in data from file

Parameters
filepath (*Union[str, TextIO]*) –

Returns
tuple of input_data as a dictionary of pandas DataFrames and dictionary of default values

Return type
Tuple[Dict[str, pd.DataFrame], Dict[str, Any]]

class otoole.input.**Strategy**(*user_config: Dict[str, Dict]*)

Bases: ABC

Parameters
user_config (*dict, default=None*) – A user configuration for the input parameters and sets

property user_config: Dict

class otoole.input.**WriteStrategy**(*user_config: Dict*, *filepath: Optional[str] = None*, *default_values: Optional[Dict] = None*, *write_defaults: bool = False*, *input_data: Optional[Dict[str, DataFrame]] = None*)

Bases: *Strategy*

The WriteStrategy interface declares operations common to all writing formats

The Context uses this interface to call the algorithm defined by Concrete Strategies.

Parameters

- **user_config** (*dict, default=None*) –
- **filepath** (*str, default=None*) –
- **default_values** (*dict, default=None*) –
- **write_defaults** (*bool, default=False*) –
- **input_data** (*dict, default=None*) –

write(*inputs: Dict[str, DataFrame]*, *filepath: str*, *default_values: Dict[str, float]*, ***kwargs*)

Perform the conversion from dict of dataframes to destination format

otoole.read_strategies module

class otoole.read_strategies.**ReadCsv**(*user_config: Dict[str, Dict]*, *keep_whitespace: bool = False*)

Bases: _ReadTabular

Read in a folder of CSV files to a dict of Pandas DataFrames

Parameters

- **user_config** (*Dict[str, Dict]*) – User configuration
- **keep_whitespace** (*bool*) – Whether to keep whitespace in the dataframes

read(*filepath*, ***kwargs*) → Tuple[Dict[str, DataFrame], Dict[str, Any]]

Reads in data from file

Parameters
filepath (*Union[str, TextIO]*) –

Returns
tuple of input_data as a dictionary of pandas DataFrames and dictionary of default values

Return type
Tuple[Dict[str, pd.DataFrame], Dict[str, Any]]

class otoole.read_strategies.**ReadDatafile**(*user_config: Dict[str, Dict]*)

Bases: *ReadStrategy*

Read in a datafile to a dict of Pandas DataFrames

Parameters

- **user_config** (*Dict[str, Dict]*) – User configuration
- **keep_whitespace** (*bool*) – Whether to keep whitespace in the dataframes

extract_param(*config*, *name*, *datafile_parser*, *dict_of_dataframes*) → DataFrame

extract_set(*datafile_parser*, *name*, *config*, *dict_of_dataframes*) → DataFrame

read(*filepath*, ***kwargs*) → Tuple[Dict[str, DataFrame], Dict[str, Any]]

Reads in data from file

Parameters
filepath (*Union[str, TextIO]*) –

Returns
tuple of input_data as a dictionary of pandas DataFrames and dictionary of default values

Return type
Tuple[Dict[str, pd.DataFrame], Dict[str, Any]]

read_in_datafile(*path_to_datafile: str*, *config: Dict*) → Amply

Read in a datafile using the Amply parsing class

Parameters

- **path_to_datafile** (*str*) –
- **config** (*Dict*) –

class otoole.read_strategies.**ReadExcel**(*user_config: Dict[str, Dict]*, *keep_whitespace: bool = False*)

Bases: _ReadTabular

Read in an Excel spreadsheet in wide format to a dict of Pandas DataFrames

Parameters

- **user_config** (*Dict[str, Dict]*) – User configuration
- **keep_whitespace** (*bool*) – Whether to keep whitespace in the dataframes

read(*filepath: Union[str, TextIO]*, ***kwargs*) → Tuple[Dict[str, DataFrame], Dict[str, Any]]

Reads in data from file

Parameters
filepath (*Union[str, TextIO]*) –

Returns
tuple of input_data as a dictionary of pandas DataFrames and dictionary of default values

Return type
Tuple[Dict[str, pd.DataFrame], Dict[str, Any]]

class otoole.read_strategies.**ReadMemory**(*parameters: Dict[str, DataFrame]*, *user_config: Dict[str, Dict]*)

Bases: *ReadStrategy*

Read a dict of OSeMOSYS parameters from memory

Parameters

- **parameters** (*Dict[str, pd.DataFrame]*) – Dictionary of OSeMOSYS parameters
- **user_config** (*Dict[str, Dict]*) – User configuration

read(*filepath: Optional[Union[str, TextIO]] = None*, ***kwargs*) → Tuple[Dict[str, DataFrame], Dict[str, Any]]

Reads in data from file

Parameters
filepath (*Union[str, TextIO]*) –

Returns
tuple of input_data as a dictionary of pandas DataFrames and dictionary of default values

Return type
Tuple[Dict[str, pd.DataFrame], Dict[str, Any]]

otoole.utils module

class otoole.utils.**UniqueKeyLoader**(*stream*)

Bases: SafeLoader

YALM Loader to find duplicate keys

This loader will treat lowercase and uppercase keys as the same. Meaning, the keys "SampleKey" and "SAMPLEKEY" are considered the same.

Raises

- **ValueError** – If a key is defined more than once.
- **Adapted from:** –

- **https://stackoverflow.com/a/63215043/14961492** –

construct_mapping(*node*, *deep=False*)

otoole.utils.**create_name_mappings**(*config: Dict[str, Dict[str, Union[str, List]]]*, *map_full_to_short: bool = True*) → Dict

Creates name mapping between full name and short name.

Parameters

- **config** (*Dict[str, Dict[str, Union[str, List]]]*) – Parsed user configuration file
- **map_full_to_short** (*bool*) – Map full name to short name if true, else map short name to full name

Returns

Mapping of full name to shortened name

Return type

csv_to_excel Dict[str, str]

otoole.utils.**extract_config**(*schema: Dict*, *default_values: Dict*) → Dict[str, Dict[str, Union[str, List[str]]]]

otoole.utils.**format_config_for_validation**(*config_in: Dict*) → List

Formats config for validation function.

Flattens dictionary to a list

Parameters

config_in (*Dict*) – Read in user config yaml file

Returns

config_out

Return type

List

Example

```
>>> config_in
>>> AccumulatedAnnualDemand:
      indices: [REGION,FUEL,YEAR]
      type: param
      dtype: float
      default: 0
```

```
>>> config_out
>>> [{
    name: AccumulatedAnnualDemand
    indices: [REGION,FUEL,YEAR]
    type: param
    dtype: float
    default: 0
    }, ... ]
```

otoole.utils.**get_all_sets**(*config: Dict*) → List

Extracts user defined sets

otoole.utils.**get_packaged_resource**(*input_data: Dict[str, DataFrame]*, *param: str*) → List[Dict[str, Any]]

Gets input parameter data and formats it as a dictionary

Parameters

- **input_data** (*Dict[str, pd.DataFrame]*) – Internal datastore for otoole input data
- **param** (*str*) – Name of OSeMOSYS parameter

Returns

List of all rows in the df, where each dictionary is the column name, followed by the value in that row

Return type

List[Dict[str,any]]

Example

```
>>> get_packaged_resource(input_data, "InputActivityRatio")
>>> [{'REGION': 'SIMPLICITY',
    'TECHNOLOGY': 'RIVWATAGR',
    'FUEL': 'WATIN',
    'MODE_OF_OPERATION': 1,
    'YEAR': 2020,
    'VALUE': 1.0}]
```

otoole.utils.**read_deprecated_datapackage**(*datapackage: str*) → str

Checks filepath for CSVs if a datapackage is provided

Parameters

datapackage (*str*) – Location of input datapackge

Returns

input_csvs – Location of input csv files

Return type

str

Raises

OtooleDeprecationError – If no 'data/' directory is found

otoole.utils.**read_packaged_file**(*filename: str*, *module_name: Optional[str] = None*)

otoole.utils.**validate_config**(*config: Dict*) → None

Validates user input data

Parameters

config (*Dict*) – Read in user config yaml file

Raises

ValidationError – If the user inputs are not valid

otoole.validate module

Ensures that technology and fuel names match the convention

For example, to validate the following list of names, you would use the config shown below:

```
theseUNIQUE_ENTRY1
are__UNIQUE_ENTRY2
all__UNIQUE_ENTRY1
validUNIQUE_ENTRY2
entryUNIQUE_ENTRY1
in__UNIQUE_ENTRY2
a____UNIQUE_ENTRY1
list_UNIQUE_ENTRY2
```

Create a yaml validation config with the following format:

```
codes:
  some_valid_codes:
    UNIQUE_ENTRY1: Description of unique entry 1
    UNIQUE_ENTRY2: Description of unique entry 2
schema:
  schema_name:
  - name: first_entry_in_schema
    valid: ['these', 'are__', 'all__', 'valid', 'entry', 'in__', 'a____', 'list_']
    position: (1, 5) # a tuple representing the start and end position
  - name: second_entry_in_schema
    valid: some_valid_codes  # references an entry in the codes section of the
    config
    position: (6, 19) # a tuple representing the start and end position
```

otoole.validate.**check_for_duplicates**(*codes: Sequence*) → bool

otoole.validate.**compose_expression**(*schema: List*) → str

Generates a regular expression from a schema

Return type
str

otoole.validate.**compose_multi_expression**(*resource: List*) → str

Concatenates multiple expressions using an OR operator

Use to validate elements using an OR operation e.g. the elements must match this expression OR the expression

otoole.validate.**create_schema**(*config: Optional[Dict[str, Dict]] = None*) → Dict

Populate the dict of schema with codes from the validation config

Parameters
config (*dict, default=None*) – A configuration dictionary containing `codes` and `schema` keys

otoole.validate.**identify_orphaned_fuels_techs**(*input_data: Dict[str, DataFrame]*) → Dict[str, str]

Returns a list of fuels and technologies which are unconnected

Return type
dict

otoole.validate.**main**(*input_data: Dict[str, DataFrame]*, *config=None*)

otoole.validate.**read_validation_config**()

otoole.validate.**validate**(*expression: str*, *name: str*) → bool

Determine if `name` matches the `expression`

Parameters

- **expression** (*str*) –
- **name** (*str*) –

Return type
bool

otoole.validate.**validate_resource**(*input_data: Dict[str, DataFrame]*, *resource: str*, *schemas: List[Dict]*) → None

Validates a single resource against the validation config.

Parameters

- **input_data** (*dict[str,pd.DataFrame]*) – otoole internal datastore
- **resource** (*str*) –
- **schemas** (*List[Dict]*) – The schema from which to create a validation expression

otoole.write_strategies module

class otoole.write_strategies.**WriteCsv**(*user_config: Dict*, *filepath: Optional[str] = None*, *default_values: Optional[Dict] = None*, *write_defaults: bool = False*, *input_data: Optional[Dict[str, DataFrame]] = None*)

Bases: *WriteStrategy*

Write parameters to comma-separated value files

Parameters

- **filepath** (*str, default=None*) – The path to write a folder of csv files
- **default_values** (*dict, default=None*) –
- **user_config** (*dict, default=None*) –

class otoole.write_strategies.**WriteDatafile**(*user_config: Dict*, *filepath: Optional[str] = None*, *default_values: Optional[Dict] = None*, *write_defaults: bool = False*, *input_data: Optional[Dict[str, DataFrame]] = None*)

Bases: *WriteStrategy*

class otoole.write_strategies.**WriteExcel**(*user_config: Dict*, *filepath: Optional[str] = None*, *default_values: Optional[Dict] = None*, *write_defaults: bool = False*, *input_data: Optional[Dict[str, DataFrame]] = None*)

Bases: *WriteStrategy*

Module contents

otoole.**read**(*config: str*, *from_format: str*, *from_path: str*, *keep_whitespace: bool = False*) → Tuple[Dict[str, DataFrame], Dict[str, float]]

Read OSeMOSYS data from datafile, csv or Excel formats

Parameters

- **config** (*str*) – Path to config file
- **from_format** (*str*) – Available options are 'datafile', 'csv', 'excel' and 'datapackage' [deprecated]
- **from_path** (*str*) – Path to source file (if datafile or excel) or folder (csv)
- **keep_whitespace** (*bool, default: False*) – Keep whitespace in source files

Returns
Dictionary of parameter and set data and dictionary of default values

Return type
Tuple[dict[str, pd.DataFrame], dict[str, float]]

otoole.**read_results**(*config: str*, *from_format: str*, *from_path: str*, *input_format: str*, *input_path: str*, *glpk_model: Optional[str] = None*) → Tuple[Dict[str, DataFrame], Dict[str, float]]

Read OSeMOSYS results from CBC, GLPK, Gurobi, or CPLEX results files

Parameters

- **config** (*str*) – Path to config file
- **from_format** (*str*) – Available options are 'cbc', 'gurobi', 'cplex', and 'glpk'
- **from_path** (*str*) – Path to source file (if datafile or excel) or folder (csv)
- **input_format** (*str*) – Format of input data. Available options are 'datafile', 'csv' and 'excel'
- **input_path** (*str*) – Path to input data
- **glpk_model** (*str*) – Path to `*.glp` model file

Returns
Dictionary of parameter and set data and dictionary of default values

Return type
Tuple[dict[str, pd.DataFrame], dict[str, float]]

otoole.**write**(*config: str*, *to_format: str*, *to_path: str*, *inputs*, *default_values: Optional[Dict[str, float]] = None*) → bool

Write OSeMOSYS data to datafile, csv or Excel formats

Parameters

- **config** (*str*) – Path to config file
- **to_format** (*str*) – Available options are 'datafile', 'csv', 'excel' and 'datapackage' [deprecated],
- **to_path** (*str*) – Path to destination file (if datafile or excel) or folder (csv))
- **inputs** (*dict[str, pd.DataFrame]*) – Dictionary of pandas data frames to write
- **default_values** (*dict[str, float], default: None*) – Dictionary of default values to write to datafile

CHAPTER

TWO

INDICES AND TABLES

- genindex
- modindex
- search

PYTHON MODULE INDEX

O

INDEX

D

E

F

G

I

M

N

O

P

R

S

T

U

V

W