
PyEHub Documentation

Ralph Evins

Mar 24, 2024

PYEHUB

1	BatchRunExcel	3
2	ci	5
3	epsilon_constraint	7
4	excel_to_request_format	9
5	logger	11
6	multiple_hubs	13
7	network_to_request_format	15
8	outputter	17
9	run	21
10	ehub_model	23
11	input_data	27
12	param_var	31
13	utils	33
14	response_format	35
15	capacity	37
16	converter	39
17	network	41
18	request_format	43
19	storage	45
20	stream	47
21	time_series	49
22	constraint	51

23	expression	53
24	problem	55
25	tutorial	57
26	variable	59
27	multi_run_period	61
28	Indices and tables	63
	Python Module Index	65
	Index	67

To view the Besos documentation go to: <https://besos.readthedocs.io>

BATCHRUNEXCEL

A script used to solve a batch series of energy hubs in a combined PyEHub model.

This functionality is now also provided by the multiple hubs module.

`BatchRunExcel.get_output_fields(list, str)`

Gets the required outputs from the excel files passed in the list of filenames.

Parameters

- **filenames** (*list*) – A list of the excel files to find outputs for.
- **directory** (*str*) – The optional directory that these filenames are located at. for use only if the filenames do not include their directory.

Returns

A list with every output requested from every file (duplicates are deleted.)

`BatchRunExcel.output_to_excel(dict, openpyxl.sheet, dict)`

Takes the output dictionary produced by multiple hubs, and writes specific fields to the passed excel sheet based on the output dictionary passed.

Parameters

- **outputs** (*dict*) – The dictionary produced by multiple_hubs.
- **sheet** (*openpyxl.sheet*) – The excel sheet to be written to.
- **output_dict** (*dict*) – A dictionary that links keys found in the outputs dict, and the column that they should appear in, within the sheet.

A script for running as part of a CI system.

It currently only runs Pylint on any *.py* files in the directory. Pylint checks for style and some common errors. See the *.pylintrc* file for Pylint config options. If you have an objection with the style, make a PR to change it.

To run the script, just do:

```
$ python ci.py
```

in the same directory as *ci.py*.

returns

0 for success, non-zero for failure

rtype

A standard error code

exception `ci.CheckError`

The CI process detected an error.

`ci.get_files_to_check(directory: str) → Iterable[str]`

Iterate over the files to check in the directory.

Parameters

directory – The directory to check files in

Returns

An iterator over the Python files

`ci.lint(file: str) → None`

Call pylint on a file.

Parameters

file – The relative path of the file to *ci.py*

`ci.main() → None`

The main function that runs the script.

`ci.run_doctests()`

Run *doctest* on all the files.

Returns

The return code of running all the tests.

`ci.run_linting() → int`

Run pylint on all the files.

Returns

The return code

`ci.run_system_test()`

Run all the system tests.

Returns

The return code

EPSILON_CONSTRAINT

EXCEL_TO_REQUEST_FORMAT

A script that converts an Excel file into the request format.

The Excel file has to be in one of the supported formats, which examples can be found in the `data_formats` directory.

To run the script on a supported Excel file, do:

```
$ python excel_to_request_format.py <Excel file path> <output file path>
```

This prints out the request format of the excel file into *<output file path>*.

To see all the possible arguments, run:

```
$ python excel_to_request_format.py -help
```

class `excel_to_request_format.Converter(excel_file)`

The SuperClass for all Excel-to-request-format converters.

All a subclass needs to do is implement all the abstract methods and the superclass does the rest of the work.

convert()

Convert the file into the request format.

Returns

The request format

exception `excel_to_request_format.FormatUnsupportedError`

The Excel format is not supported.

`excel_to_request_format.convert(excel_file)`

Convert the excel file into the request format.

Parameters

excel_file – The path to the excel file

Returns

The request format as a Python dict

`excel_to_request_format.main()`

The function that runs the script.

`excel_to_request_format.parse_args()`

Parses the command-line arguments.

LOGGER

This script is an interface for using the logging module to log the running of energy hub model.

`logger.add_console_handler()` → None

Add a logging handler for stderr.

`logger.add_file_handler(filename: str)` → None

Add a logging handler for a log file.

Parameters

filename – The name of the log file

`logger.create_logger(filename: str)` → None

Add logging to the application.

Parameters

filename – The name of the log file

`logger.get_default_formatter()` → Formatter

Return a default formatter for logging.

The format is:

Hours:Minutes:Second.Milliseconds - Location - [level]: message

Returns

A logging.Formatter for the default format

MULTIPLE_HUBS

Solving energy hub model for n number of hubs with any network of connections wanted between the hubs.

To run for multiple hubs:

```
$ python multiple_hubs.py -n NUMBER OF HUBS
```

If n (NUMBER OF HUBS) is not inputted the code will not run.

Naming input excel files:

file names should start with “hub” followed by increasing numbers starting from 0. The files should be in the ‘network’ folder

To set the connections between the hubs (the links):

All the links between the hubs should be set in a separate excel file in the ‘network’ folder. It should be called “network.xlsx”

The default set up is one directional connections. To set bidirectional connections, the connections should be defined per each direction, i.e:

one direction: link 0: start_id: 0 end_id: 1

bidirectional: link 0: start_id: 0 end_id: 1,
link 1: start_id: 1 end_id: 0

The link ids should start from 0 and increase by 1. The node ids (start_id and end_id) correspond to hubs numbering in the names of the hubs excel files.

Note: Do not name constraints specific names in the EHubModel class -> will not be able to construct constraints in multiple_hubs

```
class multiple_hubs.NetworkModel(*, excel=None, request=None, name=None, network=None,  
                                network_request=None, hub_id=None, max_carbon=None)
```

A subclass that allows connections between hubs.

calc_network_investment_cost()

Calculating investment cost for the links the hub has. Cost split between 2 hubs that are connected with the same link

calc_total_cost()

Modifying total cost constraint. Adding network cost to the total cost.

link_is_installed(link)

Set binary to 1 if capacity of link is > 0. :param link: A link

link_is_installed_2(link)

Set binary to 1 if capacity of link is > 0. :param link: A link

`multiple_hubs.linear_power_flow_constraint(power, angle_from, angle_to, time, reactance)`

Constraint for linear powerflows

`multiple_hubs.link_capacity_constraint(link, hub, i)`

Constraint for the flow in the links.

`multiple_hubs.multiple_hubs(minimize_carbon=False, output_filename=None, input_files=None, network_excel=None, network_request=None, max_carbon=None, n=0, solver='glpk')`

Core function for solving of multiple PyEHub models.

`multiple_hubs.network_constraint(hub, link_end, link_start)`

Yields the constraints that allow a network connection between two hubs.

Parameters

- **hub** – The hub
- **link_end** – all the links that the hub ends at
- **link_start** – all the links that the hub start at

Yields

A network energy balanced constraints for each hub

NETWORK_TO_REQUEST_FORMAT

A script that converts a network Excel file into the request format.

The Excel file has to be in one of the supported formats, which examples can be found in the `data_formats` directory.

OUTPUTTER

A script for outputting the results of a PyEHub model.

Outputs can be stored to an excel spreadsheet or printed to the console.

`outputter.output_excel(solution: dict, file_name: str, time_steps: int | None = None, sheets: list | None = None)`

Output the solution dictionary to an Excel file.

It outputs the time series data in their own sheet with the rest being put into another sheet.

Parameters

- **solution** – A dictionary of the solution part of the response format. This contains the variables and parameters of the solved model.
- **file_name** – The name of the Excel file to write to
- **time_steps** – Optional. The number of time steps to classify a dataframe as holding time series data.
- **sheets** – A list of all the sheets to be contained in the excel file.

`outputter.place_in_dict(t, model, dictionary: dict, key: str, value)`

Appends an entry in the list corresponding to key if key exists in dictionary. Otherwise creates a single entry in the dictionary as {key: [value]}

`outputter.plot_energy_balance(model, results: dict, **kwargs) → None`

Visualization of energy balance for all the streams, i.e., Plots the energy interactions of all the streams with loads, converters, storages, imports and exports. Plots are dashed and marked with shape markers for visibility in case of overlap. Adjust the properties of the plot by passing arguments from below.

Parameters

- **model** – The object of EHubModel class (or any child class thereof); the energy hub model.
- **results** – dictionary; returned by solve() method.
- **size** – tuple; (width, height) of the plot [default is (9,5)].
- **lw** – float; linewidth of the plots [default is 2].
- **dl** – float; length of the dashes constituting the dashed plots [default is 3].

`outputter.plot_storages(results: dict, **kwargs) → None`

Plots various variables related to storages: storage state: The level of energy remaining in the storage. gross charge(charge from stream): Energy sent to storage from stream for charging. gross discharge(discharge to stream): Energy going into stream after discharging. net charge: Energy actually reaching storage(after loss of gross charge due to charging efficiency). net discharge: Energy actually leaving storage(after loss from this due to discharging efficiency, it becomes gross

discharge).

decay loss(standing loss): Energy dissipating due to standing losses. Note: All of the above are plotted as default. You can change this by passing respective arguments.

Parameters

- **results** – dictionary; returned by solve() method.
- **pl_state** – boolean value; whether to plot ‘storage state’ or not.
- **pl_gross_ch** – boolean value; whether to plot ‘gross charge’ or not.
- **pl_gross_dch** – boolean value; whether to plot ‘gross discharge’ or not.
- **pl_net_ch** – boolean value; whether to plot ‘net charge’ or not.
- **pl_net_dch** – boolean value; whether to plot ‘net discharge’ or not.
- **pl_decay** – boolean value; whether to plot ‘decay loss’ or not.
- **size** – tuple; (width, height) of the plot [default is (10,5)].
- **percentage** – boolean; y-axis units in % or in kWh [default].

outputter.**pretty_print**(*results: dict*) → None

Print the results in a prettier format.

Parameters

results – The results dictionary to print

outputter.**print_capacities**(*results*)

Prints the capacities of each tech and storage at the end

Parameters

results – the solved model

outputter.**print_section**(*section_name: str, solution_section: dict*) → None

Print all the attributes with a heading.

Parameters

- **section_name** – The heading
- **solution_section** – The dictionary with all the attributes

outputter.**print_warning**(*results*)

Prints an error if the model burns energy, i.e there is energy from storage and energy to storage at same time step.

Parameters

results – the solved model

outputter.**sort_dict**(*mapping: dict | OrderedDict*) → OrderedDict

Sorts a dictionary and all its sub-dictionaries as well.

Examples

```
>>> sort_dict({1: 'a', 3: 'c', 2: 'b'})
OrderedDict([(1, 'a'), (2, 'b'), (3, 'c')])
>>> sort_dict({1: {3: 'c', 2: 'b', 1: 'a'}})
OrderedDict([(1, OrderedDict([(1, 'a'), (2, 'b'), (3, 'c')]))])
```

Parameters

mapping – The dictionary to be sorted

Returns

The sorted dictionary as an OrderedDict

`outputter.stream_info(results, output_file)`

New output format with the information separated in different sheets for different streams.

Parameters

- **results** – the solved model
- **output_file** – the output excel file

`outputter.to_dataframe(name: str, value: dict) → DataFrame`

Convert a dictionary into a dataframe.

Parameters

- **name** – The name for the dataframe
- **value** – The dictionary to be converted

Returns

A Pandas DataFrame.

`outputter.to_dataframes(frames: dict) → OrderedDict`

Convert the values of a dictionary into dataframes if they can be converted.

Parameters

frames – The dictionary to be converted

Returns

An ordered dictionary with dataframes as values (if they can be converted).

CHAPTER

NINE

RUN

EHUB_MODEL

Provides a class for encapsulating an energy hub model.

class `energy_hub.ehub_model.EHubModel(*, excel=None, request=None, big_m=99999, max_carbon=None)`

Represents a black-box Energy Hub.

calc_investment_cost()

A constraint for calculating the investment cost.

calc_maintenance_cost()

A constraint for calculating the maintenance cost.

calc_operating_cost()

A constraint for calculating the total operating cost.

calc_total_carbon()

A constraint for calculating the total carbon produced.

calc_total_cost()

A constraint for calculating the total cost.

capacity_bounds()

Ensure the capacities are within their given bounds.

compile()

Build all the constraints and variables of the model.

property constraints

The list of constraints on the model.

energy_balance(*t, stream*)

Ensure the loads and exported energy is below the produced energy.

Parameters

- **t** – A time step
- **stream** – An output stream

max_carbon_level()

Constraint to set a max carbon cap.

property objective

The objective “function” of the model.

recompile()

Clear all constraints and variables then compile again.

solve(*solver_settings*: dict | None = None, *is_verbose*: bool = False)

Solve the model.

Parameters

- **solver_settings** – The config options for the solver
- **is_verbose** – Makes it so the solver prints everything

Returns

The results

storage_balance(*t*, *storage*)

Calculate the current storage level from the previous level.

Parameters

- **t** – A time step
- **storage** – A storage

storage_charge_rate(*t*, *storage*)

Ensure the charge rate of a storage is below it's maximum rate.

Parameters

- **t** – A time step
- **storage** – A storage

storage_discharge_rate(*t*, *storage*)

Ensure the discharge rate of a storage is below it's maximum rate.

Parameters

- **t** – A time step
- **storage** – A storage

storage_input_positive(*t*, *storage*)

Energy to storage should be positive.

storage_is_installed(*storage*)

Set binary to 1 if capacity of storage is > 0. :param storage: A storage

storage_is_installed_2(*storage*)

Set binary to 1 if capacity of storage is > 0. :param storage: A storage

storage_level_above_minimum(*t*, *storage*)

Ensure the storage level is above it's minimum level.

Parameters

- **t** – A time step
- **storage** – A storage

storage_level_below_capacity(*t*, *storage*)

Ensure the storage's level is below the storage's capacity.

Parameters

- **t** – A time step
- **storage** – A storage

storage_level_positive(*t, storage*)

Storages' levels should be above zero.

storage_looping(*storage*)

Ensure that the storage level at the beginning is equal to it's end level.

storage_output_positive(*t, storage*)

Energy from the storages should be positive.

tech_export_positive(*t, stream*)

Energy exported should be positive.

tech_import_positive(*t, stream*)

Energy imported should be positive.

tech_input_below_capacity(*t, tech*)

Ensure the energy input by a tech is less than its capacity.

Parameters

- **t** – A time step
- **tech** – A converter

tech_input_positive(*t, tech*)

Energy input to tech should be positive.

tech_is_installed(*tech*)

Set binary to 1 if capacity of tech is > 0. :param tech: A converter

tech_is_installed_2(*tech*)

Set binary to 1 if capacity of tech is > 0. :param tech: A converter

tech_is_on(*t, disp, out*)

Set binary to on if tech is active.

Parameters

- **t** – A time step
- **disp** – A dispatch tech
- **out** – An output energy stream

tech_is_on_2(*t, disp, out*)

Set binary to on if tech is active.

Parameters

- **t** – A time step
- **disp** – A dispatch tech
- **out** – An output energy stream

tech_part_loads(*t, disp, out*)

Parameters

- **t** – A time step
- **disp** – A dispatch tech
- **out** – An output energy stream

exception `energy_hub.ehub_model.InfeasibleConstraintError`(*constraint_name: str | None = None*,
arguments: tuple | None = None)

A constraint will always be false.

INPUT_DATA

Provides functionality for handling the request format for using in the EHubModel.

class `energy_hub.input_data.InputData`(*request: dict*)

Provides convenient access to needed data to implement an energy hub model.

property `c_matrix`: `Dict[str, Dict[str, float]]`

Return a dictionary-format for the C matrix.

The format is like {converter name: {stream name: ...}, ...}

property `capacities`: `List[Capacity]`

The list of capacities.

property `carbon_credits`: `Dict[str, float | str]`

The carbon credit of each stream.

property `carbon_factors`: `Dict[str, float | str]`

The carbon factor of each stream.

property `converter_names`: `List[str]`

Return the names of the converters.

property `converters`: `List[Converter]`

The list of converters.

property `converters_capacity`: `Dict[str, float]`

Return the capacities of the converters.

property `demand_stream_names`

The sorted list of demand streams names.

property `demands`: `Iterable[TimeSeries]`

Return the TimeSeries that are demands.

property `export_streams`: `List[str]`

The names of streams that are exportable.

property `feed_in`: `Dict[str, float | str]`

The export price of each output stream.

property `fixed_capital_costs`: `Dict[str, float]`

Return the fixed capital cost for each converter.

property `fuel_price`: `Dict[str, float | str]`

Return the price of each fuel.

property import_streams: List[str]

The names of streams that are importable.

property interest_rate: float

The interest rate.

property linear_cost: Dict[str, float]

Return the linear cost for each tech.

property link_capacity: Dict[int, float]

Return the capacities of the converters.

property link_end: Dict[int, int]

Return the id of the end node

property link_length: Dict[int, float]

Returns the length of each link

property link_reactance: [<class 'int'>, <class 'float'>]

Return the reactance of each link

property link_start: Dict[int, int]

Return the id of the start node

property link_thermal_loss: [<class 'int'>, <class 'float'>]

Return the thermal loss of each link

property link_type: [<class 'int'>, <class 'str'>]

Return the type of each link

property links_ids: List[int]

Returns a list of id of all the links

property loads: Dict[str, Dict[int, float]]

The data for all demands as a dictionary that is indexed by (demand time series ID, time).

property output_stream_names

The sorted list of output streams names.

property part_load: Dict[str, Dict[str, float]]

Return the part load for each tech and each of its outputs.

property part_load_techs: List[str]

The names of the converters that have a part load.

property source: Iterable[TimeSeries]

Return the TimeSeries that are sources.

property source_stream_names

The sorted list of source streams names.

property storage_annual_maintenance_cost: Dict[str, float]

Returns annual maintenance cost of each storage

property storage_capacity

The capacity of each storage.

property storage_charge: Dict[str, float]
The maximum charge of each storage.

property storage_discharge: Dict[str, float]
The maximum discharge of each storage.

property storage_ef_ch: Dict[str, float]
The charging efficiency of each storage.

property storage_ef_disch: Dict[str, float]
The discharging efficiency of each storage.

property storage_fixed_capital_cost: Dict[str, float]
Returns fixed capital cost of each storage

property storage_lin_cost: Dict[str, float]
The linear cost of each storage.

property storage_loss: Dict[str, float]
The decay of each storage.

property storage_min_soc: Dict[str, float]
The minimum state of charge of each storage.

property storage_names: List[str]
Return the names of the storages.

property storage_npv: Dict[str, float]
The net present value of each storage.

property storages: List[[Storage](#)]
The list of storages.

property stream_names: List[str]
Return the names of the streams.

property stream_timeseries: Dict[str, float]
Return the streams that have an availability timeseries.

property streams: List[[Stream](#)]
The list of streams.

property tech_npv: Dict[str, float]
The net present value of each converter.

property time: List[int]
Return the time steps of the model.

property time_series_data: Dict[str, Dict[int, float]]
The data for the availability time series as a dictionary that is indexed by time.

property time_series_list: List[[TimeSeries](#)]
The list of time series.

property variable_maintenance_cost: Dict[str, float]
The variable maintenance cost of each converter.

PARAM_VAR

Provides a class that can either be a reference to a value or to a variable.

class `energy_hub.param_var.ConstantOrVar`(*indices: *List*, model=*None*, values: *Dict* | *None* = *None*)

Provides access to data that can either be a constant or a variable.

The values can either be a constant (like a float) or can be a str, which is a reference to a variable in the model.

property values

Return the values.

UTILS

Provides some utility functions for all code in `energy_hub`.

`energy_hub.utils.cached_property(func)`

Return a property that caches results.

Parameters

func – The function to decorated

Returns

The decorated cached property

`energy_hub.utils.constraint(*args, enabled=True)`

Mark a function as a constraint of the model.

The function that adds these constraints to the model is `_add_indexed_constraints`.

Parameters

- ***args** – The indices that the constraint is indexed by. Each element of each index is passed to the method.
- **enabled** – Is the constraint enabled? Defaults to True.

Returns

The decorated function

`energy_hub.utils.constraint_list(*, enabled=True)`

Mark a function as a ConstraintList of the model.

The function has to return a generator. ie: must use a yield in the method body.

Parameters

enabled – Is the constraint enabled? Defaults to True.

Returns

The decorated function

RESPONSE_FORMAT

Provides functionality for handling the response format.

Examples

If you want to validate a dictionary against the response format:

```
>>> from data_formats import response_format
>>> example = {}
>>> response_format.validate(example)
```

Traceback (most recent call last):

```
...
data_formats.response_format.ResponseValidationError
exception data_formats.response_format.ResponseValidationError
    The instance failed to validate against the SCHEMA.
data_formats.response_format.create_response(status: Status, model: Dict) → Dict[str, Any]
    Create a new response format dictionary.
```

Parameters

- **status** – The status of the problem
- **model** – A dictionary of the variables and their values that were used in the model

```
data_formats.response_format.validate(instance: dict) → None
```

Validate the instance against the schema.

Parameters

instance – The potential instance of the schema

Raises

ValidationError – the instance does not match the schema

CAPACITY

Provides functionality for handling a request format capacity.

class `data_formats.request_format.capacity.Capacity(capacity_request: dict)`

A wrapper for a request format capacity.

property domain

The domain of the capacity.

property lower_bound: float | None

The lower bound of the capacity as a float.

property name: str

The name of the capacity.

property upper_bound: float | None

The lower bound of the capacity as a float.

CONVERTER

Provides functionality for handling a request format's converter.

```
class data_formats.request_format.converter.Converter(converter_request: dict, capacity_converter:  
dict)
```

A wrapper for a request format converter.

property area: float | None

Return the area of this roof tech if this tech is a roof tech.

property capacity: float | str

Return the capacity of the converter.

property capital_cost: float

Return the capital cost of the converter.

property efficiency: float

The efficiency of the converter.

property fixed_capital_cost: float

Return the fixed capital cost of the converter.

Returns

The fixed capital cost if it has one or 0.

property has_part_load: bool

Does the converter have a part load?

property inputs: List[str]

The names of the input streams for the converter.

property is_chp: bool

Does the converter use and/or output heat and power?

property is_dispatch: bool

Is this a dispatch converter?

property is_grid: bool

Is this converter the grid?

property is_roof_tech: bool

Is this converter on the roof?

property is_solar: bool

Is the converter solar?

property lifetime: float

The lifetime in years of the tech.

property max_capacity: float

The maximum capacity of the converter.

property min_load: float | None

Return the minimum load of the tech if it has one.

property name: str

The name of the converter.

property output_ratios: Dict[str, float]

The output ratios for each output stream.

property outputs: List[str]

The names of the output streams for the converter.

property usage_maintenance_cost: float

The usage maintenance cost of the converter.

NETWORK

Provides functionality for handling a request format network_links.

```
class data_formats.request_format.network.Network_links(network_links_request: dict, capacity_link:  
dict)
```

A wrapper for a request format Network_links.

```
property end_id: int
```

return end node id

```
property length: float
```

return the length of the link

```
property link_capacity: float | str
```

return the capacity of the connection

```
property link_cost: float
```

return the cost of the connection

```
property link_id: int
```

return network link id

```
property link_reactance: float
```

return the reactance across the connection

```
property link_type: str
```

return the type of the connection

```
property start_id: int
```

return start node id

```
property total_pressure_loss: float
```

return the pressure loss of the connection

```
property total_thermal_loss: float
```

return the thermal loss of the connection

REQUEST_FORMAT

Provides functionality for handling the request format.

Examples

If you want to validate a dictionary against the request format:

```
>>> from data_formats import request_format
>>> example = {}
>>> request_format.validate(example)
```

Traceback (most recent call last):

```
...
data_formats.request_format.request_format.RequestValidationError
exception data_formats.request_format.request_format.RequestValidationError
    The request format instance failed to validate against the SCHEMA.
data_formats.request_format.request_format.validate(instance: dict) → None
    Validate the instance against the schema.
```

Parameters

instance – The potential instance of the schema

Raises

ValidationError – the instance does not match the schema

STORAGE

Provides functionality for handling a request format's storage.

class `data_formats.request_format.storage.Storage`(*storage_request: dict, capacity_request: dict*)

A wrapper for a request format storage.

property `annual_maintenance_cost: float`

The annual maintenance cost of the storage

property `capacity: float | str`

Return the capacity of the storage.

property `charge_efficiency: float`

The charge efficiency of the storage.

property `cost: float`

The cost of the storage.

property `decay: float`

The decay of the storage.

property `discharge_efficiency: float`

The discharge efficiency of the storage.

property `fixed_capital_cost: float`

The fixed capital cost of the storage

property `lifetime: float`

The life time in years of the storage.

property `max_charge: float`

The maximum charge of the storage.

property `max_discharge: float`

The maximum discharge of the storage.

property `min_state: float`

The minimum state of charge of the storage.

property `name: str`

Return the name of the storage.

property `stream: str`

The stream that this storage holds.

STREAM

Provides functionality for handling the streams in the request format.

```
class data_formats.request_format.stream.Stream(stream_request: dict, request: dict)
```

A wrapper for a request format stream.

```
property co2: float
```

The CO2 factor of the stream.

```
property co2_credit: float
```

The CO2 factor of the stream.

```
property export_price: float
```

The export price of the stream.

```
property exportable: bool
```

If the stream is exportable.

```
property importable: bool
```

If the stream is importable.

```
property is_output: bool
```

Is this an output stream?

```
property name: str
```

The name of the stream.

```
property price: float
```

The price of the stream.

```
property timeseries: str
```

The availability of the stream.

TIME_SERIES

Provides functionality for handling a request format's time series.

class `data_formats.request_format.time_series.TimeSeries`(*time_series_request: dict*)

A wrapper for a request format time series.

property data: `Dict[int, float]`

A dictionary from time to the values of the series.

property is_demand: `bool`

Is this time series a demand?

property is_price: `bool`

Is this time series a price

property is_solar: `bool`

Is this time series for solar data?

property is_source: `bool`

Is this time series a source?

property name: `str`

The name of the time series.

property stream: `str`

Return the name of the stream.

CONSTRAINT

Contains a class for a linear programming constraint.

Exports:

```
>>> from pylp.constraint import Constraint
```

class `pylp.constraint.Constraint`(*lhs, operator: str, rhs*)

Represents a linear programming constraint.

construct()

Build the constraint for use in the solver.

EXPRESSION

Contains functionality for dealing with expressions.

Exports:

```
>>> from pylp.expression import Expression
```

class pylp.expression.**Expression**(operator: str, operands: list)

Represents an expression in a linear programming problem.

Notes

A list of operands is given in order to reduce the height of the expression tree.

Normally, an operator operates on two operands. Thus the expression:

5 + 5 + 5

can be represented as a tree:

•

/ 5 +
 / 5 5

But as the expression gets longer, a naive approach would result in a very tall tree. In order to evaluate the tree, we would have to traverse all the way to its leafs, which at its deepest part would probably result in a stack overflow (maximum recursion error).

But treating the operands as a list, results in a nicer tree:

•

/ | 5 5 5

And this tree stays at the same height the more operands it holds.

This prevents a stack overflow from occurring. This also results in better performance as well.

construct()

Build the expression for use in the solver.

evaluate()

Evaluate the expression.

is_same_type(other: Expression) → bool

Return True if the other expression is of the same type.

PROBLEM

Contains functionality for dealing with a linear programming model.

```
class pylp.problem.Status(status, time)
```

status

Alias for field number 0

time

Alias for field number 1

```
pylp.problem.solve(*, objective=None, constraints: Iterable[Constraint] | None = None, minimize: bool =  
    False, solver: str = 'glpk', verbose: bool = False, options: list | None = None, solver_path:  
    str | None = None, **kwargs) → Status
```

Solve the linear programming problem.

Parameters

- **objective** – The objective function
- **constraints** – The collection of constraints
- **minimize** – True for minimizing; False for maximizing
- **solver** – The solver to use. Current supports ‘glpk’, ‘theo-cluster’ and ‘cplex’.
- **verbose** – If True, output the results of the solver
- **list** (*options*) – add options to the (glpk) solver
- ****kwargs** – is used to set the cluster path

Returns

Optimal, Unbounded, etc.) and the elapsed time

Return type

A tuple of the status (eg

solver: theo-cluster

This is a specific version of the code to do cluster submission.

TUTORIAL

This file serves as a tutorial/example of using PyLP.

To run the file from the root directory of this project, do

```
python3.6 -m pylp.tutorial
```

This is because this tutorial is inside the package it is using.

`pylp.tutorial.main()`

The main function of the tutorial.

For this tutorial, we will be finding a selection of candy that minimizes our costs but we get at least 400 grams of sugar.

VARIABLE

Contains classes for linear programming variables.

class `pulp.variable.BinaryVariable`(*name: str | None = None*)

A variable that represents a binary variable.

ie: it only has two value: 0 or 1.

class `pulp.variable.IntegerVariable`(*name: str | None = None*)

A variable that represents an integer number.

class `pulp.variable.RealVariable`(*name: str | None = None*)

A variable that represents a real number.

class `pulp.variable.Variable`(**, name: str | None = None, category: str = 'Continuous'*)

Represents a linear programming variable.

This is the parent class of all variable classes that you should be using. ie. do NOT use this class directly.

construct() → `LpVariable`

Build the variable for use in the solver.

evaluate() → `float`

The value of the variable found by the solver.

classmethod `pulp_name`() → `str`

Return a unique name for a new variable.

MULTI_RUN_PERIOD

`multi_run_period.merge_hubs(hubs)`

Compiles and combines the constraints of each subproblem into one list. :param hubs: List of EHub. :return: The list of constraints for each hub combined with the capacities constraint to ensure the same converter capacities across all EHub.

`multi_run_period.multi_run_output(hubs, factor)`

Function for collecting the correct info from the multiple EHub models and combining them into one set of results :param hubs: the EHub models now containing their solutions :param factor: If the models are samples for larger time scales

`multi_run_period.run_split_period(excel=None, request=None, output_filename=None, max_carbon=None, num_periods=1, len_periods=24, num_periods_in_sample_period=1, sample_period_position=0, solver='glpk')`

Core function for splitting a PyEHub model into smaller problems to solve together. :param excel: Input excel file if the hub to be split is in excel. Converted into request format before being split into subproblems. :param request: Input in request JSON format if the hub to be split is in JSON. :param output_filename: Name for file to right the output to if an output file is being used. :param max_carbon: Max carbon value if using a capped carbon value. :param num_periods: Number of sub problem EHub to be solved together. :param len_periods: Number of time steps per sub problem EHub to be solved. :param num_periods_in_sample_period: Number of periods being grouped together to be represented by 1 sub problem EHub. Example: One week representing a whole month would be ~four periods in a sample period. :param sample_period_position: Which period in the grouped sample to use as the representative EHub. Example the second week of every month would be two. :param solver: Which MILP solver to use. :return: The status of pyIp's solving, the list of hubs (each with their solution), and the absolute cost (cost of all the subproblems added together with the correct factor)

`multi_run_period.same_converter_constraint(converter, hubs)`

Constraint to ensure the capacities are kept constant across all the subproblems.

`multi_run_period.same_storage_constraint(storage, hubs)`

Constraint to ensure the capacities are kept constant across all the subproblems.

`multi_run_period.split_hubs(excel=None, request=None, max_carbon=None, num_periods=1, len_periods=24, num_periods_in_sample_period=1, sample_period_position=0)`

Splits a PyEHub into a series of smaller hubs with a given period.

INDICES AND TABLES

- `genindex`
- `modindex`
- `search`

PYTHON MODULE INDEX

b

`BatchRunExcel`, 3

c

`ci`, 5

d

`data_formats.request_format.capacity`, 37

`data_formats.request_format.converter`, 39

`data_formats.request_format.network`, 41

`data_formats.request_format.request_format`,
43

`data_formats.request_format.storage`, 45

`data_formats.request_format.stream`, 47

`data_formats.request_format.time_series`, 49

`data_formats.response_format`, 35

e

`energy_hub.ehub_model`, 23

`energy_hub.input_data`, 27

`energy_hub.param_var`, 31

`energy_hub.utils`, 33

`excel_to_request_format`, 9

l

`logger`, 11

m

`multi_run_period`, 61

`multiple_hubs`, 13

n

`network_to_request_format`, 15

o

`outputter`, 17

p

`pulp.constraint`, 51

`pulp.expression`, 53

`pulp.problem`, 55

`pulp.tutorial`, 57

`pulp.variable`, 59

A

add_console_handler() (in module *logger*), 11
 add_file_handler() (in module *logger*), 11
 annual_maintenance_cost
 (*data_formats.request_format.storage.Storage*
 property), 45
 area (*data_formats.request_format.converter.Converter*
 property), 39

B

BatchRunExcel
 module, 3
 BinaryVariable (class in *pylp.variable*), 59

C

c_matrix (*energy_hub.input_data.InputData* *property*),
 27
 cached_property() (in module *energy_hub.utils*), 33
 calc_investment_cost() (*energy_hub.ehub_model.EHubModel*
 method), 23
 calc_maintenance_cost() (*energy_hub.ehub_model.EHubModel*
 method), 23
 calc_network_investment_cost() (multiple_hubs.*NetworkModel*
 method), 13
 calc_operating_cost() (*energy_hub.ehub_model.EHubModel*
 method), 23
 calc_total_carbon() (*energy_hub.ehub_model.EHubModel*
 method), 23
 calc_total_cost() (*energy_hub.ehub_model.EHubModel*
 method), 23
 calc_total_cost() (multiple_hubs.*NetworkModel*
 method), 13
 capacities (*energy_hub.input_data.InputData* *prop-*
 erty), 27
 Capacity (class in *data_formats.request_format.capacity*),
 37

capacity (*data_formats.request_format.converter.Converter*
 property), 39
 capacity (*data_formats.request_format.storage.Storage*
 property), 45
 capacity_bounds() (*energy_hub.ehub_model.EHubModel*
 method), 23
 capital_cost (*data_formats.request_format.converter.Converter*
 property), 39
 carbon_credits (*energy_hub.input_data.InputData*
 property), 27
 carbon_factors (*energy_hub.input_data.InputData*
 property), 27
 charge_efficiency (*data_formats.request_format.storage.Storage*
 property), 45
 CheckError, 5
 ci
 module, 5
 co2 (*data_formats.request_format.stream.Stream* *prop-*
 erty), 47
 co2_credit (*data_formats.request_format.stream.Stream*
 property), 47
 compile() (*energy_hub.ehub_model.EHubModel*
 method), 23
 ConstantOrVar (class in *energy_hub.param_var*), 31
 Constraint (class in *pylp.constraint*), 51
 constraint() (in module *energy_hub.utils*), 33
 constraint_list() (in module *energy_hub.utils*), 33
 constraints (*energy_hub.ehub_model.EHubModel*
 property), 23
 construct() (*pylp.constraint.Constraint* *method*), 51
 construct() (*pylp.expression.Expression* *method*), 53
 construct() (*pylp.variable.Variable* *method*), 59
 convert() (*excel_to_request_format.Converter*
 method), 9
 convert() (in module *excel_to_request_format*), 9
 Converter (class in *data_formats.request_format.converter*),
 39
 Converter (class in *excel_to_request_format*), 9
 converter_names (*energy_hub.input_data.InputData*
 property), 27
 converters (*energy_hub.input_data.InputData* *prop-*

erty), 27

converters_capacity (energy_hub.input_data.InputData property), 27

cost (data_formats.request_format.storage.Storage property), 45

create_logger() (in module logger), 11

create_response() (in module data_formats.response_format), 35

D

data (data_formats.request_format.time_series.TimeSeries property), 49

data_formats.request_format.capacity module, 37

data_formats.request_format.converter module, 39

data_formats.request_format.network module, 41

data_formats.request_format.request_format module, 43

data_formats.request_format.storage module, 45

data_formats.request_format.stream module, 47

data_formats.request_format.time_series module, 49

data_formats.response_format module, 35

decay (data_formats.request_format.storage.Storage property), 45

demand_stream_names (energy_hub.input_data.InputData property), 27

demands (energy_hub.input_data.InputData property), 27

discharge_efficiency (data_formats.request_format.storage.Storage property), 45

domain (data_formats.request_format.capacity.Capacity property), 37

E

efficiency (data_formats.request_format.converter.Converter property), 39

EHubModel (class in energy_hub.ehub_model), 23

end_id (data_formats.request_format.network.NetworkLink property), 41

energy_balance() (energy_hub.ehub_model.EHubModel method), 23

energy_hub.ehub_model module, 23

energy_hub.input_data module, 27

energy_hub.param_var module, 31

energy_hub.utils module, 33

evaluate() (pylp.expression.Expression method), 53

evaluate() (pylp.variable.Variable method), 59

excel_to_request_format module, 9

export_price (data_formats.request_format.stream.Stream property), 47

export_streams (energy_hub.input_data.InputData property), 27

exportable (data_formats.request_format.stream.Stream property), 47

Expression (class in pyp.expression), 53

F

feed_in (energy_hub.input_data.InputData property), 27

fixed_capital_cost (data_formats.request_format.converter.Converter property), 39

fixed_capital_cost (data_formats.request_format.storage.Storage property), 45

fixed_capital_costs (energy_hub.input_data.InputData property), 27

FormatUnsupportedError, 9

fuel_price (energy_hub.input_data.InputData property), 27

G

get_default_formatter() (in module logger), 11

get_files_to_check() (in module ci), 5

get_output_fields() (in module BatchRunExcel), 3

H

has_part_load (data_formats.request_format.converter.Converter property), 39

I

import_streams (energy_hub.input_data.InputData property), 27

importable (data_formats.request_format.stream.Stream property), 47

InfeasibleConstraintError, 25

InputData (class in energy_hub.input_data), 27

inputs (data_formats.request_format.converter.Converter property), 39

IntegerVariable (class in pyp.variable), 59

interest_rate (energy_hub.input_data.InputData property), 28

is_chp (data_formats.request_format.converter.Converter property), 39

[is_demand](#) ([data_formats.request_format.time_series.TimeSeries](#) property), 49
[is_dispatch](#) ([data_formats.request_format.converter.Converter](#) property), 39
[is_grid](#) ([data_formats.request_format.converter.Converter](#) property), 39
[is_output](#) ([data_formats.request_format.stream.Stream](#) property), 47
[is_price](#) ([data_formats.request_format.time_series.TimeSeries](#) property), 49
[is_roof_tech](#) ([data_formats.request_format.converter.Converter](#) property), 39
[is_same_type](#) ([pylp.expression.Expression](#) method), 53
[is_solar](#) ([data_formats.request_format.converter.Converter](#) property), 39
[is_solar](#) ([data_formats.request_format.time_series.TimeSeries](#) property), 49
[is_source](#) ([data_formats.request_format.time_series.TimeSeries](#) property), 49
L
[length](#) ([data_formats.request_format.network.Network_links](#) property), 41
[lifetime](#) ([data_formats.request_format.converter.Converter](#) property), 39
[lifetime](#) ([data_formats.request_format.storage.Storage](#) property), 45
[linear_cost](#) ([energy_hub.input_data.InputData](#) property), 28
[linear_power_flow_constraint](#) () (in module [multiple_hubs](#)), 13
[link_capacity](#) ([data_formats.request_format.network.Network_links](#) property), 41
[link_capacity](#) ([energy_hub.input_data.InputData](#) property), 28
[link_capacity_constraint](#) () (in module [multiple_hubs](#)), 14
[link_cost](#) ([data_formats.request_format.network.Network_links](#) property), 41
[link_end](#) ([energy_hub.input_data.InputData](#) property), 28
[link_id](#) ([data_formats.request_format.network.Network_links](#) property), 41
[link_is_installed](#) () ([multiple_hubs.NetworkModel](#) method), 13
[link_is_installed_2C](#) () ([multiple_hubs.NetworkModel](#) method), 13
[link_length](#) ([energy_hub.input_data.InputData](#) property), 28
[link_reactance](#) ([data_formats.request_format.network.Network_links](#) property), 41
[link_reactance](#) ([energy_hub.input_data.InputData](#) property), 28
[links_start](#) ([energy_hub.input_data.InputData](#) property), 28
[link_thermal_loss](#) ([energy_hub.input_data.InputData](#) property), 28
[link_type](#) ([data_formats.request_format.network.Network_links](#) property), 41
[link_type](#) ([energy_hub.input_data.InputData](#) property), 28
[links_ids](#) ([energy_hub.input_data.InputData](#) property), 28
[lint](#) () (in module [ci](#)), 5
[loads](#) ([energy_hub.input_data.InputData](#) property), 28
[logger](#)
[module](#), 11
[lower_bound](#) ([data_formats.request_format.capacity.Capacity](#) property), 37
M
[main](#) () (in module [ci](#)), 5
[main](#) () (in module [excel_to_request_format](#)), 9
[main](#) () (in module [pylp.tutorial](#)), 57
[max_capacity](#) ([data_formats.request_format.converter.Converter](#) property), 40
[max_carbon_level](#) () ([energy_hub.ehub_model.EHubModel](#) method), 23
[max_charge](#) ([data_formats.request_format.storage.Storage](#) property), 45
[max_discharge](#) ([data_formats.request_format.storage.Storage](#) property), 45
[merge_hubs](#) () (in module [multi_run_period](#)), 61
[min_load](#) ([data_formats.request_format.converter.Converter](#) property), 40
[min_state](#) ([data_formats.request_format.storage.Storage](#) property), 45
module
[BatchRunExcel](#), 3
[ci](#), 5
[data_formats.request_format.capacity](#), 37
[data_formats.request_format.converter](#), 39
[data_formats.request_format.network](#), 41
[data_formats.request_format.request_format](#), 43
[data_formats.request_format.storage](#), 45
[data_formats.request_format.stream](#), 47
[data_formats.request_format.time_series](#), 49
[data_formats.response_format](#), 35
[energy_hub.ehub_model](#), 23
[energy_hub.input_data](#), 27
[energy_hub.param_var](#), 31
[energy_hub.utils](#), 33
[excel_to_request_format](#), 9

[logger](#), 11
[multi_run_period](#), 61
[multiple_hubs](#), 13
[network_to_request_format](#), 15
[outputter](#), 17
[pylp.constraint](#), 51
[pylp.expression](#), 53
[pylp.problem](#), 55
[pylp.tutorial](#), 57
[pylp.variable](#), 59
[multi_run_output\(\)](#) (in module [multi_run_period](#)), 61
[multi_run_period](#)
 module, 61
[multiple_hubs](#)
 module, 13
[multiple_hubs\(\)](#) (in module [multiple_hubs](#)), 14

N

[name](#) ([data_formats.request_format.capacity.Capacity](#) property), 37
[name](#) ([data_formats.request_format.converter.Converter](#) property), 40
[name](#) ([data_formats.request_format.storage.Storage](#) property), 45
[name](#) ([data_formats.request_format.stream.Stream](#) property), 47
[name](#) ([data_formats.request_format.time_series.TimeSeries](#) property), 49
[network_constraint\(\)](#) (in module [multiple_hubs](#)), 14
[Network_links](#) (class in [data_formats.request_format.network](#)), 41
[network_to_request_format](#)
 module, 15
[NetworkModel](#) (class in [multiple_hubs](#)), 13

O

[objective](#) ([energy_hub.ehub_model.EHubModel](#) property), 23
[output_excel\(\)](#) (in module [outputter](#)), 17
[output_ratios](#) ([data_formats.request_format.converter.Converter](#) property), 40
[output_stream_names](#) ([energy_hub.input_data.InputData](#) property), 28
[output_to_excel\(\)](#) (in module [BatchRunExcel](#)), 3
[outputs](#) ([data_formats.request_format.converter.Converter](#) property), 40
[outputter](#)
 module, 17

P

[parse_args\(\)](#) (in module [excel_to_request_format](#)), 9
[part_load](#) ([energy_hub.input_data.InputData](#) property), 28

[part_load_techs](#) ([energy_hub.input_data.InputData](#) property), 28
[place_in_dict\(\)](#) (in module [outputter](#)), 17
[plot_energy_balance\(\)](#) (in module [outputter](#)), 17
[plot_storages\(\)](#) (in module [outputter](#)), 17
[pretty_print\(\)](#) (in module [outputter](#)), 18
[price](#) ([data_formats.request_format.stream.Stream](#) property), 47
[print_capacities\(\)](#) (in module [outputter](#)), 18
[print_section\(\)](#) (in module [outputter](#)), 18
[print_warning\(\)](#) (in module [outputter](#)), 18
[pulp_name\(\)](#) ([pylp.variable.Variable](#) class method), 59
[pylp.constraint](#)
 module, 51
[pylp.expression](#)
 module, 53
[pylp.problem](#)
 module, 55
[pylp.tutorial](#)
 module, 57
[pylp.variable](#)
 module, 59

R

[RealVariable](#) (class in [pylp.variable](#)), 59
[recompile\(\)](#) ([energy_hub.ehub_model.EHubModel](#) method), 23
[RequestValidationError](#), 43
[ResponseValidationError](#), 35
[run_doctests\(\)](#) (in module [ci](#)), 5
[run_linting\(\)](#) (in module [ci](#)), 5
[run_split_period\(\)](#) (in module [multi_run_period](#)), 61
[run_system_test\(\)](#) (in module [ci](#)), 6

S

[same_converter_constraint\(\)](#) (in module [multi_run_period](#)), 61
[same_storage_constraint\(\)](#) (in module [multi_run_period](#)), 61
[solve\(\)](#) ([energy_hub.ehub_model.EHubModel](#) method), 23
[solve\(\)](#) (in module [pylp.problem](#)), 55
[sort_dict\(\)](#) (in module [outputter](#)), 18
[source](#) ([energy_hub.input_data.InputData](#) property), 28
[source_stream_names](#) ([energy_hub.input_data.InputData](#) property), 28
[split_hubs\(\)](#) (in module [multi_run_period](#)), 61
[start_id](#) ([data_formats.request_format.network.Network_links](#) property), 41
[Status](#) (class in [pylp.problem](#)), 55
[status](#) ([pylp.problem.Status](#) attribute), 55
[Storage](#) (class in [data_formats.request_format.storage](#)), 45

`storage_annual_maintenance_cost` (energy_hub.input_data.InputData property), 28
`storage_balance()` (energy_hub.ehub_model.EHubModel method), 24
`storage_capacity` (energy_hub.input_data.InputData property), 28
`storage_charge` (energy_hub.input_data.InputData property), 28
`storage_charge_rate()` (energy_hub.ehub_model.EHubModel method), 24
`storage_discharge` (energy_hub.input_data.InputData property), 29
`storage_discharge_rate()` (energy_hub.ehub_model.EHubModel method), 24
`storage_ef_ch` (energy_hub.input_data.InputData property), 29
`storage_ef_disch` (energy_hub.input_data.InputData property), 29
`storage_fixed_capital_cost` (energy_hub.input_data.InputData property), 29
`storage_input_positive()` (energy_hub.ehub_model.EHubModel method), 24
`storage_is_installed()` (energy_hub.ehub_model.EHubModel method), 24
`storage_is_installed_2()` (energy_hub.ehub_model.EHubModel method), 24
`storage_level_above_minimum()` (energy_hub.ehub_model.EHubModel method), 24
`storage_level_below_capacity()` (energy_hub.ehub_model.EHubModel method), 24
`storage_level_positive()` (energy_hub.ehub_model.EHubModel method), 24
`storage_lin_cost` (energy_hub.input_data.InputData property), 29
`storage_looping()` (energy_hub.ehub_model.EHubModel method), 25
`storage_loss` (energy_hub.input_data.InputData property), 29
`storage_min_soc` (energy_hub.input_data.InputData property), 29
`storage_names` (energy_hub.input_data.InputData property), 29
`storage_npv` (energy_hub.input_data.InputData property), 29
`storage_output_positive()` (energy_hub.ehub_model.EHubModel method), 25
`storages` (energy_hub.input_data.InputData property), 29
`Stream` (class in data_formats.request_format.stream), 47
`stream` (data_formats.request_format.storage.Storage property), 45
`stream` (data_formats.request_format.time_series.TimeSeries property), 49
`stream_info()` (in module outputter), 19
`stream_names` (energy_hub.input_data.InputData property), 29
`stream_timeseries` (energy_hub.input_data.InputData property), 29
`streams` (energy_hub.input_data.InputData property), 29

T

`tech_export_positive()` (energy_hub.ehub_model.EHubModel method), 25
`tech_import_positive()` (energy_hub.ehub_model.EHubModel method), 25
`tech_input_below_capacity()` (energy_hub.ehub_model.EHubModel method), 25
`tech_input_positive()` (energy_hub.ehub_model.EHubModel method), 25
`tech_is_installed()` (energy_hub.ehub_model.EHubModel method), 25
`tech_is_installed_2()` (energy_hub.ehub_model.EHubModel method), 25
`tech_is_on()` (energy_hub.ehub_model.EHubModel method), 25
`tech_is_on_2()` (energy_hub.ehub_model.EHubModel method), 25
`tech_npv` (energy_hub.input_data.InputData property), 29
`tech_part_loads()` (energy_hub.ehub_model.EHubModel method), 25
`time` (energy_hub.input_data.InputData property), 29
`time` (pylp.problem.Status attribute), 55

`time_series_data` (*energy_hub.input_data.InputData*
property), 29

`time_series_list` (*energy_hub.input_data.InputData*
property), 29

`TimeSeries` (class in *data_formats.request_format.time_series*),
49

`timeseries` (*data_formats.request_format.stream.Stream*
property), 47

`to_dataframe()` (in module *outputter*), 19

`to_dataframes()` (in module *outputter*), 19

`total_pressure_loss`
(*data_formats.request_format.network.Network_links*
property), 41

`total_thermal_loss` (*data_formats.request_format.network.Network_links*
property), 41

U

`upper_bound` (*data_formats.request_format.capacity.Capacity*
property), 37

`usage_maintenance_cost`
(*data_formats.request_format.converter.Converter*
property), 40

V

`validate()` (in module
data_formats.request_format.request_format),
43

`validate()` (in module *data_formats.response_format*),
35

`values` (*energy_hub.param_var.ConstantOrVar* prop-
erty), 31

`Variable` (class in *pylp.variable*), 59

`variable_maintenance_cost` (en-
ergy_hub.input_data.InputData property),
29