PyEHub Documentation

Ralph Evins

Sep 02, 2023

PyEHub

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

CHAPTER 1

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 dicretory 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.

CHAPTER 2

ci

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

CHAPTER 3

epsilon_constraint

CHAPTER 4

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.

CHAPTER 5

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**() → logging.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

CHAPTER 6

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

CHAPTER 7

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.

CHAPTER 8

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*, *sheets: list = 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*) → collections.OrderedDict
Sorts a dictionary and all its sub-dicionaries 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*) → pandas.core.frame.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*) → collections.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 9

run

CHAPTER 10

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.

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.

objective
The objective "function" of the model.

recompile()
Clear all constraints and variables then compile again.

solve(*solver_settings: dict = 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 exported 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*, *arguments: tuple = None*)

A constraint will always be false.

CHAPTER 11

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.

c_matrix

Return a dictionary-format for the C matrix.

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

capacities

The list of capacities.

carbon_credits

The carbon credit of each stream.

carbon_factors

The carbon factor of each stream.

converter_names

Return the names of the converters.

converters

The list of converters.

converters_capacity

Return the capacities of the converters.

demand_stream_names

The sorted list of demand streams names.

demands

Return the TimeSeries that are demands.

export_streams

The names of streams that are exportable.

feed_in

The export price of each output stream.

fixed_capital_costs
Return the fixed capital cost for each converter.

fuel_price
Return the price of each fuel.

import_streams
The names of streams that are importable.

interest_rate
The interest rate.

linear_cost
Return the linear cost for each tech.

link_capacity
Return the capacities of the converters.

link_end
Return the id of the end node

link_length
Returns the length of each link

link_reactance
Return the reactance of each link

link_start
Return the id of the start node

link_thermal_loss
Return the thermal loss of each link

link_type
Return the type of each link

links_ids
Returns a list of id of all the links

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

output_stream_names
The sorted list of output streams names.

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

part_load_techs
The names of the converters that have a part load.

source
Return the TimeSeries that are sources.

source_stream_names
The sorted list of source streams names.

storage_annual_maintenance_cost
Returns annual maintenance cost of each storage

storage_capacity
The capacity of each storage.

storage_charge
The maximum charge of each storage.

storage_discharge
The maximum discharge of each storage.

storage_ef_ch
The charging efficiency of each storage.

storage_ef_disch
The discharging efficiency of each storage.

storage_fixed_capital_cost
Returns fixed capital cost of each storage

storage_lin_cost
The linear cost of each storage.

storage_loss
The decay of each storage.

storage_min_soc
The minimum state of charge of each storage.

storage_names
Return the names of the storages.

storage_npv
The net present value of each storage.

storages
The list of storages.

stream_names
Return the names of the streams.

stream_timeseries
Return the streams that have an availability timeseries.

streams
The list of streams.

tech_npv
The net present value of each converter.

time
Return the time steps of the model.

time_series_data
The data for the availability time series as a dictionary that is indexed by time.

time_series_list
The list of time series.

variable_maintenance_cost
The variable maintenance cost of each converter.

CHAPTER 12

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*, *model=None*, *values: Dict = 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.

values

Return the values.

CHAPTER 13

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

CHAPTER 14

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: pylp.problem.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

CHAPTER 15

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.

domain

The domain of the capacity.

lower_bound

The lower bound of the capacity as a float.

name

The name of the capacity.

upper_bound

The lower bound of the capacity as a float.

CHAPTER 16

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.

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

capacity
Return the capacity of the converter.

capital_cost
Return the capital cost of the converter.

efficiency
The efficiency of the converter.

fixed_capital_cost
Return the fixed capital cost of the converter.

Returns The fixed capital cost if it has one or 0.

has_part_load
Does the converter have a part load?

inputs
The names of the input streams for the converter.

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

is_dispatch
Is this a dispatch converter?

is_grid
Is this converter the grid?

is_roof_tech
Is this converter on the roof?

is_solar
Is the converter solar?

lifetime
The lifetime in years of the tech.

max_capacity
The maximum capacity of the converter.

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

name
The name of the converter.

output_ratios
The output ratios for each output stream.

outputs
The names of the output streams for the converter.

usage_maintenance_cost
The usage maintenance cost of the converter.

CHAPTER 17

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.

end_id
return end node id

length
return the length of the link

link_capacity
return the capacity of the connection

link_cost
return the cost of the connection

link_id
return network link id

link_reactance
return the reactance across the connection

link_type
return the type of the connection

start_id
return start node id

total_pressure_loss
return the pressure loss of the connection

total_thermal_loss
return the thermal loss of the connection

CHAPTER 18

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

CHAPTER 19

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.

annual_maintenance_cost
The annual maintenance cost of the storage

capacity
Return the capacity of the storage.

charge_efficiency
The charge efficiency of the storage.

cost
The cost of the storage.

decay
The decay of the storage.

discharge_efficiency
The discharge efficiency of the storage.

fixed_capital_cost
The fixed capital cost of the storage

lifetime
The life time in years of the storage.

max_charge
The maximum charge of the storage.

max_discharge
The maximum discharge of the storage.

min_state
The minimum state of charge of the storage.

name
Return the name of the storage.

stream
The stream that this storage holds.

CHAPTER 20

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.

co2
The C02 factor of the stream.

co2_credit
The C02 factor of the stream.

export_price
The export price of the stream.

exportable
If the stream is exportable.

importable
If the stream is importable.

is_output
Is this an output stream?

name
The name of the stream.

price
The price of the stream.

timeseries
The availability of the stream.

CHAPTER 21

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.

data
A dictionary from time to the values of the series.

is_demand
Is this time series a demand?

is_price
Is this time series a price

is_solar
Is this time series for solar data?

is_source
Is this time series a source?

name
The name of the time series.

stream
Return the name of the stream.

CHAPTER 22

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.

CHAPTER 23

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: pylp.expression.Expression*) → bool
Return True if the other expression is of the same type.

CHAPTER 24

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[pylp.constraint.Constraint] = None*, *minimize: bool = False*, *solver: str = 'glpk'*, *verbose: bool = False*, *options: list = None*, *solver_path: str = None*, ***kwargs*) → pylp.problem.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.

CHAPTER 25

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.

CHAPTER 26

variable

Contains classes for linear programming variables.

class pylp.variable.**BinaryVariable**(*name: str = None*)
A variable that represents a binary variable.

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

class pylp.variable.**IntegerVariable**(*name: str = None*)
A variable that represents an integer number.

class pylp.variable.**RealVariable**(*name: str = None*)
A variable that represents a real number.

class pylp.variable.**Variable**(*, *name: str = 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() → pulp.pulp.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.

CHAPTER 27

multi_run_period

multi_run_period.**merge_hubs**(*hubs*)

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

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 EHubs 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 pylp'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.

CHAPTER 28

Indices and tables

- genindex
- modindex
- search

Python Module Index

Index

D

E

F

G

H

I

L

M

N

O

P

R

S

T

U

V