
optlang Documentation

Release 1.4.4-dirty

Niko Sonnenschein

Dec 12, 2018

Contents

1	Quick start	3
1.1	Using a particular solver	4
1.2	Quadratic programming	4
1.3	Integer programming	4
2	Example	5
3	Users's guide	9
3.1	Installation	9
3.2	Contribute	10
3.3	API reference	11
4	Indices and tables	21

Optlang is a Python package implementing a modeling language for solving mathematical optimization problems, i.e. maximizing or minimizing an objective function over a set of variables subject to a number of constraints. Optlang provides a common interface to a series of optimization tools, so different solver backends can be changed in a transparent way.

In contrast to e.g. the commonly used General Algebraic Modeling System (GAMS), optlang has a simple and intuitive interface using native Python algebra syntax, and is free and open-source.

Optlang takes advantage of the symbolic math library [SymPy](#) to allow objective functions and constraints to be easily formulated from symbolic expressions of variables (see examples). Scientists can thus use optlang to formulate their optimization problems using mathematical expressions derived from domain knowledge.

Currently supported solvers are:

- [GLPK](#) (LP/MILP; via [swiglpk](#))
- [CPLEX](#) (LP/MILP/QP)
- [Gurobi](#) (LP/MILP/QP)
- [inspyred](#) (heuristic optimization; experimental)

Support for the following solvers is in the works:

- [GAMS](#) (LP/MILP/QP/NLP; will include support for solving problems on [neos-server.org](#))
- [SOPLEX](#) (exact LP)
- [MOSEK](#), (LP/MILP/QP)

CHAPTER 1

Quick start

Consider the following linear programming optimization problem (example taken from [GLPK documentation](#)):

$$\begin{aligned} \text{Max } & 10x_1 + 6x_2 + 4x_3 \\ \text{s.t. } & x_1 + x_2 + x_3 \leq 100 \\ & 10x_1 + 4x_2 + 5x_3 \leq 600 \\ & 2x_1 + 2x_2 + 6x_3 \leq 300 \\ & x_1 \geq 0, x_2 \geq 0, x_3 \geq 0 \end{aligned}$$

Formulating and solving the problem is straightforward

```
from optlang import Model, Variable, Constraint, Objective

# All the (symbolic) variables are declared, with a name and optionally a lower and/
↳ or upper bound.
x1 = Variable('x1', lb=0)
x2 = Variable('x2', lb=0)
x3 = Variable('x3', lb=0)

# A constraint is constructed from an expression of variables and a lower and/or
↳ upper bound (lb and ub).
c1 = Constraint(x1 + x2 + x3, ub=100)
c2 = Constraint(10 * x1 + 4 * x2 + 5 * x3, ub=600)
c3 = Constraint(2 * x1 + 2 * x2 + 6 * x3, ub=300)

# An objective can be formulated
obj = Objective(10 * x1 + 6 * x2 + 4 * x3, direction='max')

# Variables, constraints and objective are combined in a Model object, which can
↳ subsequently be optimized.
model = Model(name='Simple model')
model.objective = obj
model.add([c1, c2, c3])
status = model.optimize()
```

(continues on next page)

(continued from previous page)

```
print("status:", model.status)
print("objective value:", model.objective.value)
print("-----")
for var_name, var in model.variables.items():
    print(var_name, "=", var.primal)
```

You should see the following output:

```
status: optimal
objective value: 733.333333333
-----
x2 = 66.6666666667
x3 = 0.0
x1 = 33.3333333333
```

1.1 Using a particular solver

If you have more than one solver installed, it's also possible to specify which one to use, by importing directly from the respective solver interface, e.g. from `optlang.glpk_interface` import `Model`, `Variable`, `Constraint`, `Objective`

1.2 Quadratic programming

A QP problem can be generated in the same way by creating an objective with a quadratic expression. In the above example the objective could be `obj = Objective(x1 ** 2 + x2 ** 2 - 10 * x1, direction="min")` to specify a quadratic minimization problem.

1.3 Integer programming

Integer (or mixed integer) problems can be specified by assigning the type of one or more variables to 'integer' or 'binary'. If the solver supports integer problems it will automatically use the relevant optimization algorithm and return an integer solution.

CHAPTER 2

Example

The GAMS example (<http://www.gams.com/docs/example.htm>) can be formulated and solved in optlang like this:

```
from optlang import Variable, Constraint, Objective, Model

# Define problem parameters
# Note this can be done using any of Python's data types. Here we have chosen
↪ dictionaries
supply = {"Seattle": 350, "San_Diego": 600}
demand = {"New_York": 325, "Chicago": 300, "Topeka": 275}

distances = { # Distances between locations in thousands of miles
    "Seattle": {"New_York": 2.5, "Chicago": 1.7, "Topeka": 1.8},
    "San_Diego": {"New_York": 2.5, "Chicago": 1.8, "Topeka": 1.4}
}

freight_cost = 9 # Cost per case per thousand miles

# Define variables
variables = {}
for origin in supply:
    variables[origin] = {}
    for destination in demand:
        # Construct a variable with a name, bounds and type
        var = Variable(name="{}_to_{}".format(origin, destination), lb=0, type=
↪ "integer")
        variables[origin][destination] = var

# Define constraints
constraints = []
for origin in supply:
    const = Constraint(
        sum(variables[origin].values()),
        ub=supply[origin],
        name="{}_supply".format(origin)
```

(continues on next page)

(continued from previous page)

```

    )
    constraints.append(const)
for destination in demand:
    const = Constraint(
        sum(row[destination] for row in variables.values()),
        lb=demand[destination],
        name="{}_demand".format(destination)
    )
    constraints.append(const)

# Define the objective
obj = Objective(
    sum(freight_cost * distances[ori][dest] * variables[ori][dest] for ori in supply_
↪for dest in demand),
    direction="min"
)

# We can print the objective and constraints
print(obj)
print("")
for const in constraints:
    print(const)

print("")

# Put everything together in a Model
model = Model()
model.add(constraints) # Variables are added implicitly
model.objective = obj

# Optimize and print the solution
status = model.optimize()
print("Status:", status)
print("Objective value:", model.objective.value)
print("")
for var in model.variables:
    print(var.name, ":", var.primal)

```

Outputting the following:

```

Minimize
16.2*San_Diego_to_Chicago + 22.5*San_Diego_to_New_York + 12.6*San_Diego_to_Topeka + ↪
↪15.3*Seattle_to_Chicago + 22.5*Seattle_to_New_York + 16.2*Seattle_to_Topeka

Seattle_supply: Seattle_to_Chicago + Seattle_to_New_York + Seattle_to_Topeka <= 350
San_Diego_supply: San_Diego_to_Chicago + San_Diego_to_New_York + San_Diego_to_Topeka
↪<= 600
Chicago_demand: 300 <= San_Diego_to_Chicago + Seattle_to_Chicago
Topeka_demand: 275 <= San_Diego_to_Topeka + Seattle_to_Topeka
New_York_demand: 325 <= San_Diego_to_New_York + Seattle_to_New_York

Status: optimal
Objective value: 15367.5

Seattle_to_New_York : 50
Seattle_to_Chicago : 300
Seattle_to_Topeka : 0
San_Diego_to_Chicago : 0

```

(continues on next page)

(continued from previous page)

```
San_Diego_to_Topeka : 275  
San_Diego_to_New_York : 275
```

Here we forced all variables to have integer values. To allow non-integer values, leave out `type="integer"` in the `Variable` constructor (defaults to `'continuous'`).

3.1 Installation

Install optlang using pip:

```
pip install optlang
```

Or download the source distribution and run:

```
python setup.py install
```

You can run optlang's test suite like this (you need to install nose first though):

```
python setup.py test
```

3.1.1 Solvers

To solve optimization problems, at least one supported solver must be installed. Installing optlang using `pip` will also automatically install GLPK. To use other solvers (e.g. commercial solvers) it is necessary to install them manually. Optlang interfaces with all solvers through importable python modules. If the python module corresponding to the solver can be imported without errors the solver interface should be available as an optlang submodule (e.g. `optlang.glpk_interface`).

The required python modules for the currently supported solvers are:

- GLPK: `swiglpk` (automatically installed by `pip install optlang`)
 - GLPK is an open source Linear Programming library. Swiglpk can be installed from binary wheels or from source. Installing from source requires swig and GLPK.
- Cplex: `cplex`
 - Cplex is a very efficient commercial linear and quadratic mixed-integer solver from IBM. Academic licenses are available for students and researchers.

- Gurobi: `gurobipy`
 - Gurobi is a very efficient commercial linear and quadratic mixed-integer solver. Academic licenses are available for students and researchers.
- SciPy: `scipy.optimize.linprog`
 - The SciPy `linprog` function is a very basic implementation of the simplex algorithm for solving linear optimization problems. `Linprog` is included in all recent versions of SciPy.

After importing `optlang` you can check `optlang.available_solvers` to verify that a solver is recognized.

3.1.2 Issues

Local installations like

```
python setup.py install
```

might fail installing the dependencies (unresolved issue with `easy_install`). Running

```
pip install -r requirements.txt
```

beforehand should fix this issue.

3.2 Contribute

Contributions to `optlang` are very welcome. Fork `optlang` at [github](#), implement your feature and send us a pull request. Also, please use the GitHub [issue tracker](#) to let us know about bugs or feature requests, or if you have problems or questions regarding `optlang`.

3.2.1 Add solver interface

Put your interface for new solver `XYZ` into a python module with the name `xyz_interface.py`. Please use the existing solver interfaces as a reference for how to wrap a solver. For example, start by subclassing `interface.Model`.

```
class Model(interface.Model):
    def __init__(self, problem=None, *args, **kwargs):
        super(Model, self).__init__(*args, **kwargs)
```

Then you can override the abstract methods defined in `interface.Model`. For example, override `_add_constraints()`.

```
def _add_constraint(self, constraint):
    # Check that constraint is actually supported by XYZ
    if not constraint.is_Linear:
        raise ValueError("XYZ only supports linear constraints. %s is not linear." %
            constraint)
    # Add the constraint to the user level interface
    super(Model, self)._add_constraint(constraint)
    # Add variables that are not yet in the model ...
    for var in constraint.variables:
        if var.name not in self.variables:
            self._add_variable(var)
    # Link the model to the constraint
```

(continues on next page)

(continued from previous page)

```

constraint.problem = self
# Add solver specific code ...
xyz_add_rows(self.problem, 1)
index = xyz_get_num_rows(self.problem)
xyz_set_row_name(self.problem, index, constraint.name)
...

```

3.3 API reference

3.3.1 Model

The model object represents an optimization problem and contains the variables, constraints and objective that make up the problem. Variables and constraints can be added and removed using the `.add` and `.remove` methods, while the objective can be changed by setting the objective attribute, e.g. `model.objective = Objective(expr, direction="max")`.

Once the problem has been formulated the optimization can be performed by calling the `.optimize` method. This will return the status of the optimization, most commonly 'optimal', 'infeasible' or 'unbounded'.

class `optlang.interface.Model` (*name=None, objective=None, variables=None, constraints=None, *args, **kwargs*)

Bases: `object`

The model object represents an optimization problem and contains the variables, constraints and objective that make up the problem. Variables and constraints can be added and removed using the `.add` and `.remove` methods, while the objective can be changed by setting the objective attribute, e.g. `model.objective = Objective(expr, direction="max")`.

Once the problem has been formulated the optimization can be performed by calling the `.optimize` method. This will return the status of the optimization, most commonly 'optimal', 'infeasible' or 'unbounded'.

objective

str – The objective function.

name

str, optional – The name of the optimization problem.

variables

Container, read-only – Contains the variables of the optimization problem. The keys are the variable names and values are the actual variables.

constraints

Container, read-only – Contains the variables of the optimization problem. The keys are the constraint names and values are the actual constraints.

status

str, read-only – The status of the optimization problem.

Examples

```

>>> model = Model(name="my_model")
>>> x1 = Variable("x1", lb=0, ub=20)
>>> x2 = Variable("x2", lb=0, ub=10)
>>> c1 = Constraint(2 * x1 - x2, lb=0, ub=0) # Equality constraint

```

(continues on next page)

(continued from previous page)

```
>>> model.add([x1, x2, c1])
>>> model.objective = Objective(x1 + x2, direction="max")
>>> model.optimize()
'optimal'
>>> x1.primal, x2.primal
'(5.0, 10.0)'
```

add (*stuff*, *sloppy=False*)
Add variables and constraints.

Parameters

- **stuff** (*iterable*, *Variable*, *Constraint*) – Either an iterable containing variables and constraints or a single variable or constraint.
- **sloppy** (*bool*) – Check constraints for variables that are not part of the model yet.

Returns

Return type None

classmethod clone (*model*, *use_json=True*, *use_lp=False*)

Make a copy of a model. The model being copied can be of the same type or belong to a different solver interface. This is the preferred way of copying models.

Example

```
>>> new_model = Model.clone(old_model)
```

constraint_values

The primal values of all constraints.

Returns

Return type collections.OrderedDict

constraints

The model constraints.

classmethod from_json (*json_obj*)

Constructs a Model from the provided json-object.

Example

```
>>> import json
>>> with open("path_to_file.json") as infile:
>>>     model = Model.from_json(json.load(infile))
```

interface

Provides access to the solver interface the model belongs to

Returns a Python module, for example `optlang.glpk_interface`

is_integer

objective

The model's objective function.

optimize()

Solve the optimization problem using the relevant solver back-end. The status returned by this method tells whether an optimal solution was found, if the problem is infeasible etc. Consult `optlang.statuses` for more elaborate explanations of each status.

The objective value can be accessed from `'model.objective.value'`, while the solution can be retrieved by `'model.primal_values'`.

Returns `status` – Solution status.

Return type `str`

primal_values

The primal values of model variables.

The primal values are rounded to the bounds. :returns: :rtype: `collections.OrderedDict`

reduced_costs

The reduced costs/dual values of all variables.

Returns

Return type `collections.OrderedDict`

remove(stuff)

Remove variables and constraints.

Parameters `stuff(iterable, str, Variable, Constraint)` – Either an iterable containing variables and constraints to be removed from the model or a single variable or constraint (or their names).

Returns

Return type `None`

shadow_prices

The shadow prices of model (dual values of all constraints).

Returns

Return type `collections.OrderedDict`

status

The solver status of the model.

to_json()

Returns a json-compatible object from the model that can be saved using the json module. Variables, constraints and objective contained in the model will be saved. Configurations will not be saved.

Example

```
>>> import json
>>> with open("path_to_file.json", "w") as outfile:
>>>     json.dump(model.to_json(), outfile)
```

update(callback=<type 'int'>)

Process all pending model modifications.

variables

The model variables.

3.3.2 Variable

Variable objects are used to represent each variable of the optimization problem. When the optimization is performed, the combination of variable values that optimizes the objective function, while not violating any constraints will be identified. The type of a variable ('continuous', 'integer' or 'binary') can be set using the `type` keyword of the constructor or it can be changed after initialization by `var.type = 'binary'`.

The variable class subclasses the `sympy.Symbol` class, which means that symbolic expressions of variables can be constructed by using regular python syntax, e.g. `my_expression = 2 * var1 + 3 * var2 ** 2`. Expressions like this are used when constructing Constraint and Objective objects.

Once a problem has been optimized, the primal and dual values of a variable can be accessed from the `primal` and `dual` attributes, respectively.

```
class optlang.interface.Variable(name, lb=None, ub=None, type='continuous', problem=None, *args, **kwargs)
```

Bases: `optlang.symbolics.Symbol`

Optimization variables.

Variable objects are used to represent each variable of the optimization problem. When the optimization is performed, the combination of variable values that optimizes the objective function, while not violating any constraints will be identified. The type of a variable ('continuous', 'integer' or 'binary') can be set using the `type` keyword of the constructor or it can be changed after initialization by `var.type = 'binary'`.

The variable class subclasses the `sympy.Symbol` class, which means that symbolic expressions of variables can be constructed by using regular python syntax, e.g. `my_expression = 2 * var1 + 3 * var2 ** 2`. Expressions like this are used when constructing Constraint and Objective objects. Once a problem has been optimized, the primal and dual values of a variable can be accessed from the `primal` and `dual` attributes, respectively.

name

str – The variable's name.

lb

float or None, optional – The lower bound, if None then -inf.

ub

float or None, optional – The upper bound, if None then inf.

type

str, optional – The variable type, 'continuous' or 'integer' or 'binary'.

problem

Model or None, optional – A reference to the optimization model the variable belongs to.

Examples

```
>>> Variable('x', lb=-10, ub=10)
'-10 <= x <= 10'
```

classmethod clone (*variable*, ***kwargs*)

Make a copy of another variable. The variable being copied can be of the same type or belong to a different solver interface.

Example

```
>>> var_copy = Variable.clone(old_var)
```

default_assumptions = {}

dual

The dual of variable (None if no solution exists).

classmethod from_json (*json_obj*)

Constructs a Variable from the provided json-object.

Example

```
>>> import json
>>> with open("path_to_file.json") as infile:
>>>     var = Variable.from_json(json.load(infile))
```

lb

Lower bound of variable.

name

Name of variable.

primal

The primal of variable (None if no solution exists).

set_bounds (*lb, ub*)

Change the lower and upper bounds of a variable.

to_json ()

Returns a json-compatible object from the Variable that can be saved using the json module.

Example

```
>>> import json
>>> with open("path_to_file.json", "w") as outfile:
>>>     json.dump(var.to_json(), outfile)
```

type

Variable type ('either continuous, integer, or binary'.)

ub

Upper bound of variable.

3.3.3 Constraint

class optlang.interface.**Constraint** (*expression, lb=None, ub=None, indicator_variable=None, active_when=1, *args, **kwargs*)

Bases: optlang.interface.OptimizationExpression

Constraint objects represent the mathematical (in-)equalities that constrain an optimization problem. A constraint is formulated by a symbolic expression of variables and a lower and/or upper bound. Equality constraints can be formulated by setting the upper and lower bounds to the same value.

Some solvers support indicator variables. This lets a binary variable act as a switch that decides whether the constraint should be active (cannot be violated) or inactive (can be violated).

The constraint expression can be an arbitrary combination of variables, however the individual solvers have limits to the forms of constraints they allow. Most solvers only allow linear constraints, meaning that the expression should be of the form $a * \text{var1} + b * \text{var2} + c * \text{var3} \dots$

expression

sympy – The mathematical expression defining the constraint.

name

str, optional – The constraint’s name.

lb

float or None, optional – The lower bound, if None then -inf.

ub

float or None, optional – The upper bound, if None then inf.

indicator_variable

Variable – The indicator variable (needs to be binary).

active_when

0 or 1 (default 0) – When the constraint should

problem

Model or None, optional – A reference to the optimization model the variable belongs to.

Examples

```
>>> expr = 2.4 * var1 - 3.8 * var2
>>> c1 = Constraint(expr, lb=0, ub=10)
```

```
>>> indicator_var = Variable("var3", type="binary") # Only possible with some_
↳ solvers
>>> c2 = Constraint(var2, lb=0, ub=0, indicator_variable=indicator_var, active_
↳ when=1) # When the indicator is 1, var2 is constrained to be 0
```

active_when

Activity relation of constraint to indicator variable (if supported).

classmethod clone (*constraint, model=None, **kwargs*)

Make a copy of another constraint. The constraint being copied can be of the same type or belong to a different solver interface.

Parameters

- **constraint** (*interface.Constraint (or subclass)*) – The constraint to copy
- **model** (*Model or None*) – The variables of the new constraint will be taken from this model. If None, new variables will be constructed.

Example

```
>>> const_copy = Constraint.clone(old_constraint)
```

dual

Dual of constraint (None if no solution exists).

classmethod from_json (*json_obj*, *variables=None*)

Constructs a Variable from the provided json-object.

Example

```
>>> import json
>>> with open("path_to_file.json") as infile:
>>>     constraint = Constraint.from_json(json.load(infile))
```

indicator_variable

The indicator variable of constraint (if available).

lb

Lower bound of constraint.

primal

Primal of constraint (None if no solution exists).

to_json ()

Returns a json-compatible object from the constraint that can be saved using the json module.

Example

```
>>> import json
>>> with open("path_to_file.json", "w") as outfile:
>>>     json.dump(constraint.to_json(), outfile)
```

ub

Upper bound of constraint.

3.3.4 Objective

class optlang.interface.**Objective** (*expression*, *value=None*, *direction='max'*, **args*, ***kwargs*)

Bases: optlang.interface.OptimizationExpression

Objective objects are used to represent the objective function of an optimization problem. An objective consists of a symbolic expression of variables in the problem and a direction. The direction can be either 'min' or 'max' and specifies whether the problem is a minimization or a maximization problem.

After a problem has been optimized, the optimal objective value can be accessed from the 'value' attribute of the model's objective, i.e. `obj_val = model.objective.value`.

expression

sympy – The mathematical expression defining the objective.

name

str, optional – The name of the constraint.

direction

'max' or 'min' – The optimization direction.

value

float, read-only – The current objective value.

problem

solver – The low-level solver object.

classmethod clone (*objective, model=None, **kwargs*)

Make a copy of an objective. The objective being copied can be of the same type or belong to a different solver interface.

Example

```
>>> new_objective = Objective.clone(old_objective)
```

direction

The direction of optimization. Either 'min' or 'max'.

classmethod from_json (*json_obj, variables=None*)

Constructs an Objective from the provided json-object.

Example

```
>>> import json
>>> with open("path_to_file.json") as infile:
>>>     obj = Objective.from_json(json.load(infile))
```

set_linear_coefficients (*coefficients*)

Set linear coefficients in objective.

coefficients [dict] A dictionary of the form {variable1: coefficient1, variable2: coefficient2, ... }

to_json ()

Returns a json-compatible object from the objective that can be saved using the json module.

Example

```
>>> import json
>>> with open("path_to_file.json", "w") as outfile:
>>>     json.dump(obj.to_json(), outfile)
```

value

The objective value.

3.3.5 Configuration

Optlang provides high-level solver configuration via `interface.Model.configuration`.

class `optlang.interface.Configuration` (*problem=None, *args, **kwargs*)

Bases: `object`

Optimization solver configuration. This object allows the user to change certain parameters and settings in the solver. It is meant to allow easy access to a few common and important parameters. For information on changing other solver parameters, please consult the documentation from the solver provider. Some changeable

parameters are listed below. Note that some solvers might not implement all of these and might also implement additional parameters.

verbosity

int from 0 to 3 – Changes the level of output.

timeout

int or None – The time limit in second the solver will use to optimize the problem.

presolve

Boolean or 'auto' – Tells the solver whether to use (solver-specific) pre-processing to simplify the problem. This can decrease solution time, but also introduces overhead. If set to 'auto' the solver will first try to solve without pre-processing, and only turn in on in case no optimal solution can be found.

lp_method

str – Select which algorithm the LP solver uses, e.g. simplex, barrier, etc.

classmethod **clone** (*config*, *problem=None*, ***kwargs*)

presolve

Turn pre-processing on or off. Set to 'auto' to only use presolve if no optimal solution can be found.

timeout

Timeout parameter (seconds).

verbosity

Verbosity level.

0: no output 1: error and warning messages only 2: normal output 3: full output

CHAPTER 4

Indices and tables

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

A

active_when (optlang.interface.Constraint attribute), 16
add() (optlang.interface.Model method), 12

C

clone() (optlang.interface.Configuration class method), 19
clone() (optlang.interface.Constraint class method), 16
clone() (optlang.interface.Model class method), 12
clone() (optlang.interface.Objective class method), 18
clone() (optlang.interface.Variable class method), 14
Configuration (class in optlang.interface), 18
Constraint (class in optlang.interface), 15
constraint_values (optlang.interface.Model attribute), 12
constraints (optlang.interface.Model attribute), 11, 12

D

default_assumptions (optlang.interface.Variable attribute), 15
direction (optlang.interface.Objective attribute), 17, 18
dual (optlang.interface.Constraint attribute), 16
dual (optlang.interface.Variable attribute), 15

E

expression (optlang.interface.Constraint attribute), 16
expression (optlang.interface.Objective attribute), 17

F

from_json() (optlang.interface.Constraint class method), 17
from_json() (optlang.interface.Model class method), 12
from_json() (optlang.interface.Objective class method), 18
from_json() (optlang.interface.Variable class method), 15

I

indicator_variable (optlang.interface.Constraint attribute), 16, 17
interface (optlang.interface.Model attribute), 12

is_integer (optlang.interface.Model attribute), 12

L

lb (optlang.interface.Constraint attribute), 16, 17
lb (optlang.interface.Variable attribute), 14, 15
lp_method (optlang.interface.Configuration attribute), 19

M

Model (class in optlang.interface), 11

N

name (optlang.interface.Constraint attribute), 16
name (optlang.interface.Model attribute), 11
name (optlang.interface.Objective attribute), 17
name (optlang.interface.Variable attribute), 14, 15

O

Objective (class in optlang.interface), 17
objective (optlang.interface.Model attribute), 11, 12
optimize() (optlang.interface.Model method), 12

P

presolve (optlang.interface.Configuration attribute), 19
primal (optlang.interface.Constraint attribute), 17
primal (optlang.interface.Variable attribute), 15
primal_values (optlang.interface.Model attribute), 13
problem (optlang.interface.Constraint attribute), 16
problem (optlang.interface.Objective attribute), 18
problem (optlang.interface.Variable attribute), 14

R

reduced_costs (optlang.interface.Model attribute), 13
remove() (optlang.interface.Model method), 13

S

set_bounds() (optlang.interface.Variable method), 15
set_linear_coefficients() (optlang.interface.Objective method), 18
shadow_prices (optlang.interface.Model attribute), 13

status (optlang.interface.Model attribute), [11](#), [13](#)

T

timeout (optlang.interface.Configuration attribute), [19](#)

to_json() (optlang.interface.Constraint method), [17](#)

to_json() (optlang.interface.Model method), [13](#)

to_json() (optlang.interface.Objective method), [18](#)

to_json() (optlang.interface.Variable method), [15](#)

type (optlang.interface.Variable attribute), [14](#), [15](#)

U

ub (optlang.interface.Constraint attribute), [16](#), [17](#)

ub (optlang.interface.Variable attribute), [14](#), [15](#)

update() (optlang.interface.Model method), [13](#)

V

value (optlang.interface.Objective attribute), [17](#), [18](#)

Variable (class in optlang.interface), [14](#)

variables (optlang.interface.Model attribute), [11](#), [13](#)

verbosity (optlang.interface.Configuration attribute), [19](#)