pyTFA cheat sheet

Getting started:

```
import pytfa
from math import *
from pytfa.io import import_matlab_model, load_thermoDB
```

Loading data:

```
cobra_model = import_matlab_model('model.mat')
thermo_data = load_thermoDB('../data/thermo_data.thermodb')
```

Create Thermo model:

```
tfa_model = pytfa.ThermoModel(thermo_data, cobra_model)
```

Object: ThermoModel

Important Attributes		
reactions	**DictList** of reaction objects	FBA
metabolites	DictList of metabolite objects	
log_concentration	DictList of variables for logarithmic concentrations	TFA
deltaG	DictList of variables for free energies	
deltaGstd	DictList of variables for standard free energies	

Object: Reaction

Important Attributes	
forward_variable	Variable object for the forward flux
reverse_variable	Variable object for the backward flux
bounds	Bounds on the Net Flux
lower_bound	Lower bound of the net flux
upper_bound	Upper bound of the net flux

Object: LogConcentration

Important Attributes	
variable	Variable object

Object: Variable

Important Attributes	
bounds	Variable bounds

Object: DictList

Important Attributes	
get_by_id()	Method to get an entry by its id e.g rxn_id for deltaG or met_id for logarithmic concentration

ThermoModel

Methods defined here:

__deepcopy__(self, memo)

```
:param memo:
:return:
```

__init__(self, thermo_data, model, name=None, temperature=298.15, min_ph=3, max_ph=9)

```
:param float temperature: the temperature (K) at which to perform the
 calculations
:param dict thermo_data: The thermodynamic database
:type temperature: float
```

add_constraint(self, kind, hook, expr, **kwargs)

```
Add a new constraint to a COBRApy cobra_model

:param cobra.core.model.Model model: The COBRApy cobra_model
:param string,cobra.Reaction hook: Either a string representing the n
ame
    of the variable to add to the cobra_model, or a reaction object i
f the
    kind allows it
:param sympy.core.expr.Expr expr: The expression of the constraint

:returns: The created constraint
:rtype: optlang.interface.Constraint
```

add_variable(self, kind, hook, **kwargs)

```
Add a new variable to a COBRApy cobra_model.

:param cobra.core.model.Model model: The COBRApy cobra_model
:param string,cobra.Reaction hook: Either a string representing the n
ame
    of the variable to add to the cobra_model, or a reaction object i
f the
    kind allows it

:returns: The created variable
:rtype: optlang.interface.Variable
```

convert(self, add_potentials=False, add_displacement=False, verbose=True)

```
Converts a cobra_model into a tFBA ready cobra_model by adding the
thermodynamic constraints required

.. warning::
    This function requires you to have already called
    :func:`~.pytfa.ThermoModel.prepare`, otherwise it will raise an E
xception !
```

copy(self)

get_constraints_of_type(self, constraint_type)

```
Convenience function that takes as input a constraint class and retur
ns
all its instances within the cobra_model
```

:param constraint_type:
:return:

get_primal(self, vartype, index_by_reactions=False)

Returns the primal value of the cobra_model for variables of a given type

:param vartype: Class of variable. Ex: pytfa.optim.variables.ThermoDisplacement
:return:

get_solution(self)

Overrides the cobra.core.solution method, to also get the supplementary
variables we added to the cobra_model
:return:

get_variables_of_type(self, variable_type)

Convenience function that takes as input a variable class and returns all its instances within the cobra_model

:param variable_type:
:return:

normalize_reactions(self)

Find reactions with important stoichiometry and normalizes them
:return:

optimize(self, objective_sense=None, **kwargs)

Call the Model.optimize function (which really is but an interface to the
solver's. Catches SolverError in the case of no solutions. Passes down
supplementary keyword arguments (see cobra.core.Model.optimize)
:type objective_sense: 'min' or 'max'

prepare(self)

Prepares a COBRA toolbox cobra_model for TFBA analysis by doing the following:

1. checks if a reaction is a transport reaction
2. checks the ReactionDB for Gibbs energies of formation of metabolites
3. computes the Gibbs energies of reactions

print_info(self)

Print information and counts for the cobra_model
:return:

regenerate_constraints(self)

Generates references to the cobra_model's constraints in self.**_cons_dict**
as tab-searchable attributes of the thermo cobra_model
:return:

regenerate_variables(self)

Generates references to the cobra_model's constraints in self.**_var_dict**
as tab-searchable attributes of the thermo cobra_model
:return:

remove_constraint(self, cons)

Removes a constraint

:param cons:
:return:

remove_variable(self, var)

Removes a variable

:param var:
:return:

repair(self)

Updates references to variables and constraints
:return:

Methods inherited from cobra.core.model.Model:

__add__(self, other_model)

Add the content of another model to this model (+).

The model is copied as a new object, with a new model identifier,
and copies of all the reactions in the other model are added to this
model. The objective is the sum of the objective expressions for the
two models.

__enter__(self)

Record all future changes to the model, undoing them when a call to
__exit__ is received

__exit__(self, type, value, traceback)

Pop the top context manager and trigger the undo functions

__getstate__(self)

Get state for serialization.

Ensures that the context stack is cleared prior to serialization,
since partial functions cannot be pickled reliably.

__iadd__(self, other_model)

Incrementally add the content of another model to this model (+=).

Copies of all the reactions in the other model are added to this
model. The objective is the sum of the objective expressions for the
two models.

__setstate__(self, state)

Make sure all cobra.Objects in the model point to the model.

add_boundary(self, metabolite, type='exchange', reaction_id=None, lb=None, ub=1000.0)

Add a boundary reaction for a given metabolite.

There are three different types of pre-defined boundary reactions:
exchange, demand, and sink reactions.
An exchange reaction is a reversible, imbalanced reaction that adds
to or removes an extracellular metabolite from the extracellular
compartment.
A demand reaction is an irreversible reaction that consumes an
intracellular metabolite.
A sink is similar to an exchange but specifically for intracellular
metabolites.

If you set the reaction `type` to something else, you must specify th
e
desired identifier of the created reaction along with its upper and
lower bound. The name will be given by the metabolite name and the
given `type`.

Parameters
----
metabolite : cobra.Metabolite
    Any given metabolite. The compartment is not checked but you are
    encouraged to stick to the definition of exchanges and sinks.
type : str, {"exchange", "demand", "sink"}

```
        Using one of the pre-defined reaction types is easiest. If you
        want to create your own kind of boundary reaction choose
        any other string, e.g., 'my-boundary'.
    reaction_id : str, optional
        The ID of the resulting reaction. Only used for custom reactions.
    lb : float, optional
        The lower bound of the resulting reaction. Only used for custom
        reactions.
    ub : float, optional
        The upper bound of the resulting reaction. For the pre-defined
        reactions this default value determines all bounds.

    Returns
    -------
    cobra.Reaction
        The created boundary reaction.

    Examples
    --------
    >>> import cobra.test
    >>> model = cobra.test.create_test_model("textbook")
    >>> demand = model.add_boundary(model.metabolites.atp_c, type="demand
    ")
    >>> demand.id
    'DM_atp_c'
    >>> demand.name
    'ATP demand'
    >>> demand.bounds
    (0, 1000.0)
    >>> demand.build_reaction_string()
    'atp_c --> '
```

add_cons_vars(self, what, **kwargs)

```
    Add constraints and variables to the model's mathematical problem.

    Useful for variables and constraints that can not be expressed with
    reactions and simple lower and upper bounds.

    Additions are reversed upon exit if the model itself is used as
    context.

    Parameters
    ----------
    what : list or tuple of optlang variables or constraints.
       The variables or constraints to add to the model. Must be of
       class `optlang.interface.Variable` or
       `optlang.interface.Constraint`.
    **kwargs : keyword arguments
       Passed to solver.add()
```

add_metabolites(self, metabolite_list)

```
    Will add a list of metabolites to the model object and add new
    constraints accordingly.

    The change is reverted upon exit when using the model as a context.

    Parameters
    ----------
    metabolite_list : A list of `cobra.core.Metabolite` objects
```

add_reaction(self, reaction)

```
    Will add a cobra.Reaction object to the model, if
    reaction.id is not in self.reactions.
```

```
    Parameters
    ----------
    reaction : cobra.Reaction
        The reaction to add

    Deprecated (0.6). Use `~cobra.Model.add_reactions` instead
```

add_reactions(self, reaction_list)

```
    Add reactions to the model.

    Reactions with identifiers identical to a reaction already in the
    model are ignored.

    The change is reverted upon exit when using the model as a context.

    Parameters
    ----------
    reaction_list : list
        A list of `cobra.Reaction` objects
```

get_metabolite_compartments(self)

```
    Return all metabolites' compartments.
```

merge(self, right, prefix_existing=None, inplace=True, objective='left')

```
    Merge two models to create a model with the reactions from both
    models.

    Custom constraints and variables from right models are also copied
    to left model, however note that, constraints and variables are
    assumed to be the same if they have the same name.

    right : cobra.Model
        The model to add reactions from
    prefix_existing : string
        Prefix the reaction identifier in the right that already exist
        in the left model with this string.
    inplace : bool
        Add reactions from right directly to left model object.
        Otherwise, create a new model leaving the left model untouched.
        When done within the model as context, changes to the models are
        reverted upon exit.
    objective : string
        One of 'left', 'right' or 'sum' for setting the objective of the
        resulting model to that of the corresponding model or the sum of
        both.
```

remove_cons_vars(self, what)

```
    Remove variables and constraints from the model's mathematical
    problem.

    Remove variables and constraints that were added directly to the
    model's underlying mathematical problem. Removals are reversed
    upon exit if the model itself is used as context.

    Parameters
    ----------
    what : list or tuple of optlang variables or constraints.
       The variables or constraints to add to the model. Must be of
       class `optlang.interface.Variable` or
       `optlang.interface.Constraint`.
```

remove_metabolites(self, metabolite_list, destructive=False)

```
    Remove a list of metabolites from the the object.
```

```
    The change is reverted upon exit when using the model as a context.

    Parameters
    ----------
    metabolite_list : list
        A list with `cobra.Metabolite` objects as elements.

    destructive : bool
        If False then the metabolite is removed from all
        associated reactions.  If True then all associated
        reactions are removed from the Model.
```

remove_reactions(self, reactions, remove_orphans=False)

```
    Remove reactions from the model.

    The change is reverted upon exit when using the model as a context.

    Parameters
    ----------
    reactions : list
        A list with reactions (`cobra.Reaction`), or their id's, to remov
    e

    remove_orphans : bool
        Remove orphaned genes and metabolites from the model as well
```

slim_optimize(self, error_value=nan, message=None)

```
    Optimize model without creating a solution object.

    Creating a full solution object implies fetching shadow prices and
    flux values for all reactions and metabolites from the solver
    object. This necessarily takes some time and in cases where only one
    or two values are of interest, it is recommended to instead use this
    function which does not create a solution object returning only the
    value of the objective. Note however that the `optimize()` function
    uses efficient means to fetch values so if you need fluxes/shadow
    prices for more than say 4 reactions/metabolites, then the total
    speed increase of `slim_optimize` versus `optimize` is  expected to
    be small or even negative depending on how you fetch the values
    after optimization.

    Parameters
    ----------
    error_value : float, None
       The value to return if optimization failed due to e.g.
       infeasibility. If None, raise `OptimizationError` if the
       optimization fails.
    message : string
       Error message to use if the model optimization did not succeed.

    Returns
    -------
    float
        The objective value.
```

summary(self, solution=None, threshold=1e-06, fva=None, names=False, floatfmt='.3g')

```
    Print a summary of the input and output fluxes of the model.

    Parameters
    ----------
    solution: cobra.Solution, optional
        A previously solved model solution to use for generating the
        summary. If none provided (default), the summary method will
```

```
        resolve the model. Note that the solution object must match the
        model, i.e., changes to the model such as changed bounds,
        added or removed reactions are not taken into account by this
        method.
    threshold : float, optional
        Threshold below which fluxes are not reported.
    fva : pandas.DataFrame, float or None, optional
        Whether or not to include flux variability analysis in the output
    .
        If given, fva should either be a previous FVA solution matching
        the model or a float between 0 and 1 representing the
        fraction of the optimum objective to be searched.
    names : bool, optional
        Emit reaction and metabolite names rather than identifiers (defau
    lt
        False).
    floatfmt : string, optional
        Format string for floats (default '.3g').
```

Data descriptors inherited from cobra.core.model.Model:

boundary

```
Boundary reactions in the model.
Reactions that either have no substrate or product.
```

compartments

constraints

```
The constraints in the cobra model.

In a cobra model, most constraints are metabolites and their
stoichiometries. However, for specific use cases, it may also be
useful to have other types of constraints. This property defines all
constraints currently associated with the model's problem.

Returns
-------
optlang.container.Container
    A container with all associated constraints.
```

demands

```
Demand reactions in model.
Irreversible reactions that accumulate or consume a metabolite in
the inside of the model.
```

description

exchanges

```
Exchange reactions in model.
Reactions that exchange mass with the exterior. Uses annotations
and heuristics to exclude non-exchanges such as sink reactions.
```

medium

objective

```
Get or set the solver objective

Before introduction of the optlang based problems,
this function returned the objective reactions as a list. With
optlang, the objective is not limited a simple linear summation of
individual reaction fluxes, making that return value ambiguous.
Henceforth, use `cobra.util.solver.linear_reaction_coefficients` to
get a dictionary of reactions with their linear coefficients (empty
if there are none)

The set value can be dictionary (reactions as keys, linear
coefficients as values), string (reaction identifier), int (reaction
```

```
    index), Reaction or problem.Objective or sympy expression
    directly interpreted as objectives.

    When using a `HistoryManager` context, this attribute can be set
    temporarily, reversed when the exiting the context.
```

objective_direction

```
    Get or set the objective direction.

    When using a `HistoryManager` context, this attribute can be set
    temporarily, reversed when exiting the context.
```

problem

```
    The interface to the model's underlying mathematical problem.

    Solutions to cobra models are obtained by formulating a mathematical
    problem and solving it. Cobrapy uses the optlang package to
    accomplish that and with this property you can get access to the
    problem interface directly.

    Returns
    -------
    optlang.interface
        The problem interface that defines methods for interacting with
        the problem and associated solver directly.
```

sinks

```
    Sink reactions in model.
    Reversible reactions that accumulate or consume a metabolite in
    the inside of the model.
```

solver

```
    Get or set the attached solver instance.

    The associated the solver object, which manages the interaction with
    the associated solver, e.g. glpk.

    This property is useful for accessing the optimization problem
    directly and to define additional non-metabolic constraints.

    Examples
    --------
    >>> import cobra.test
    >>> model = cobra.test.create_test_model("textbook")
    >>> new = model.problem.Constraint(model.objective.expression,
    >>> lb=0.99)
    >>> model.solver.add(new)
```

variables

```
    The mathematical variables in the cobra model.

    In a cobra model, most variables are reactions. However,
    for specific use cases, it may also be useful to have other types of
    variables. This property defines all variables currently associated
    with the model's problem.

    Returns
    -------
    optlang.container.Container
        A container with all associated variables.
```

Methods inherited from cobra.core.object.Object:

__repr__(self)

__str__(self)

Data descriptors inherited from cobra.core.object.Object:

__dict__
```
dictionary for instance variables (if defined)
```
__weakref__
```
list of weak references to the object (if defined)
```
id

Reaction

Method resolution order:

Reaction
cobra.core.object.Object
__builtin__.object

Methods defined here:

__add__(self, other)
```
Add two reactions

The stoichiometry will be the combined stoichiometry of the two
reactions, and the gene reaction rule will be both rules combined b
y an
and. All other attributes (i.e. reaction bounds) will match those o
f
the first reaction
```
__copy__(self)

__deepcopy__(self, memo)

__iadd__(self, other)

__imul__(self, coefficient)
```
Scale coefficients in a reaction by a given value

E.g. A -> B becomes 2A -> 2B.

If coefficient is less than zero, the reaction is reversed and the
bounds are swapped.
```
__init__(self, id=None, name='', subsystem='', lower_bound=0.0, upper_bound=1000.0, objective_coefficient=0.0)

__isub__(self, other)

__mul__(self, coefficient)

__radd__ = __add__(self, other)

__setstate__(self, state)
```
Probably not necessary to set _model as the cobra.Model that
contains self sets the _model attribute for all metabolites and gen
es
in the reaction.

However, to increase performance speed we do want to let the metabo
lite
and gene know that they are employed in this reaction
```
__str__(self)

__sub__(self, other)

add_metabolites(self, metabolites_to_add, combine=True, reversibly=True)
```
Add metabolites and stoichiometric coefficients to the reaction.
If the final coefficient for a metabolite is 0 then it is removed
```

```
    from the reaction.

    The change is reverted upon exit when using the model as a context.

    Parameters
    ----------
    metabolites_to_add : dict
        Dictionary with metabolite objects or metabolite identifiers as
        keys and coefficients as values. If keys are strings (name of a
        metabolite) the reaction must already be part of a model and a
        metabolite with the given name must exist in the model.

    combine : bool
        Describes behavior a metabolite already exists in the reaction.
        True causes the coefficients to be added.
        False causes the coefficient to be replaced.

    reversibly : bool
        Whether to add the change to the context to make the change
        reversibly or not (primarily intended for internal use).
```

build_reaction_from_string(self, reaction_str, verbose=True, fwd_arrow=None, rev_arrow=None, reversible_arrow=None, term_split='+')

```
    Builds reaction from reaction equation reaction_str using parser

    Takes a string and using the specifications supplied in the optiona
    l
    arguments infers a set of metabolites, metabolite compartments and
    stoichiometries for the reaction.  It also infers the reversibility
    of the reaction from the reaction arrow.

    Changes to the associated model are reverted upon exit when using
    the model as a context.

    Parameters
    ----------
    reaction_str : string
        a string containing a reaction formula (equation)
    verbose: bool
        setting verbosity of function
    fwd_arrow : re.compile
        for forward irreversible reaction arrows
    rev_arrow : re.compile
        for backward irreversible reaction arrows
    reversible_arrow : re.compile
        for reversible reaction arrows
    term_split : string
        dividing individual metabolite entries
```

build_reaction_string(self, use_metabolite_names=False)

```
    Generate a human readable reaction string
```

check_mass_balance(self)

```
    Compute mass and charge balance for the reaction

    returns a dict of {element: amount} for unbalanced elements.
    "charge" is treated as an element in this dict
    This should be empty for balanced reactions.
```

copy(self)

```
    Copy a reaction

    The referenced metabolites and genes are also copied.
```

delete(self, remove_orphans=False)

```
Removes the reaction from a model.

This removes all associations between a reaction the associated
model, metabolites and genes.

The change is reverted upon exit when using the model as a context.

Deprecated, use `reaction.remove_from_model` instead.

Parameters
----------
remove_orphans : bool
    Remove orphaned genes and metabolites from the model as well
```

get_coefficient(self, metabolite_id)

```
Return the stoichiometric coefficient of a metabolite.

Parameters
----------
metabolite_id : str or cobra.Metabolite
```

get_coefficients(self, metabolite_ids)

```
Return the stoichiometric coefficients for a list of metabolites.

Parameters
----------
metabolite_ids : iterable
    Containing ``str`` or ``cobra.Metabolite``s.
```

get_compartments(self)

```
lists compartments the metabolites are in
```

knock_out(self)

```
Knockout reaction by setting its bounds to zero.
```

remove_from_model(self, remove_orphans=False)

```
Removes the reaction from a model.

This removes all associations between a reaction the associated
model, metabolites and genes.

The change is reverted upon exit when using the model as a context.

Parameters
----------
remove_orphans : bool
    Remove orphaned genes and metabolites from the model as well
```

subtract_metabolites(self, metabolites, combine=True, reversibly=True)

```
Subtract metabolites from a reaction.

That means add the metabolites with -1*coefficient. If the final
coefficient for a metabolite is 0 then the metabolite is removed fr
om
the reaction.

Notes
-----
* A final coefficient < 0 implies a reactant.
* The change is reverted upon exit when using the model as a contex
t.

Parameters
----------
```

```
metabolites : dict
    Dictionary where the keys are of class Metabolite and the value
s
    are the coefficients. These metabolites will be added to the
    reaction.

combine : bool
    Describes behavior a metabolite already exists in the reaction.
    True causes the coefficients to be added.
    False causes the coefficient to be replaced.

reversibly : bool
    Whether to add the change to the context to make the change
    reversibly or not (primarily intended for internal use).
```

Data descriptors defined here:

boundary

```
Whether or not this reaction is an exchange reaction.

Returns `True` if the reaction has either no products or reactants.
```

bounds

```
Get or set the bounds directly from a tuple

Convenience method for setting upper and lower bounds in one line
using a tuple of lower and upper bound. Invalid bounds will raise a
n
AssertionError.

When using a `HistoryManager` context, this attribute can be set
temporarily, reversed when the exiting the context.
```

compartments

```
lists compartments the metabolites are in
```

flux

```
The flux value in the most recent solution.

Flux is the primal value of the corresponding variable in the model
.

Warnings
--------
* Accessing reaction fluxes through a `Solution` object is the safe
r,
  preferred, and only guaranteed to be correct way. You can see how
 to
  do so easily in the examples.
* Reaction flux is retrieved from the currently defined
  `self._model.solver`. The solver status is checked but there are
no
  guarantees that the current solver state is the one you are looki
ng
  for.
* If you modify the underlying model after an optimization, you wil
l
  retrieve the old optimization values.

Raises
------
RuntimeError
    If the underlying model was never optimized beforehand or the
```

```
        reaction is not part of a model.
    OptimizationError
        If the solver status is anything other than 'optimal'.
    AssertionError
        If the flux value is not within the bounds.

    Examples
    --------
    >>> import cobra.test
    >>> model = cobra.test.create_test_model("textbook")
    >>> solution = model.optimize()
    >>> model.reactions.PFK.flux
    7.477381962160283
    >>> solution.fluxes.PFK
    7.4773819621602833
```

flux_expression

```
    Forward flux expression

    Returns
    -------
    sympy expression
        The expression representing the the forward flux (if associated
        with model), otherwise None. Representing the net flux if
        model.reversible_encoding == 'unsplit' or None if reaction is
        not associated with a model
```

forward_variable

```
    An optlang variable representing the forward flux

    Returns
    -------
    optlang.interface.Variable
        An optlang variable for the forward flux or None if reaction is
        not associated with a model.
```

functional

```
    All required enzymes for reaction are functional.

    Returns
    -------
    bool
        True if the gene-protein-reaction (GPR) rule is fulfilled for
        this reaction, or if reaction is not associated to a model,
        otherwise False.
```

gene_name_reaction_rule

```
    Display gene_reaction_rule with names intead.

    Do NOT use this string for computation. It is intended to give a
    representation of the rule using more familiar gene names instead o
    f
    the often cryptic ids.
```

gene_reaction_rule

genes

lower_bound

```
    Get or set the lower bound

    Setting the lower bound (float) will also adjust the associated opt
    lang
    variables associated with the reaction. Infeasible combinations,
    such as a lower bound higher than the current upper bound will
    update the other bound.
```

When using a `HistoryManager` context, this attribute can be set
temporarily, reversed when the exiting the context.

metabolites

model

returns the model the reaction is a part of

objective_coefficient

Get the coefficient for this reaction in a linear
objective (float)

Assuming that the objective of the associated model is summation of
fluxes from a set of reactions, the coefficient for each reaction
can be obtained individually using this property. A more general wa
y
is to use the `model.objective` property directly.

products

Return a list of products for the reaction

reactants

Return a list of reactants for the reaction.

reaction

Human readable reaction string

reduced_cost

The reduced cost in the most recent solution.

Reduced cost is the dual value of the corresponding variable in the
model.

Warnings
----
* Accessing reduced costs through a `Solution` object is the safer,
  preferred, and only guaranteed to be correct way. You can see how
 to
  do so easily in the examples.
* Reduced cost is retrieved from the currently defined
  `self._model.solver`. The solver status is checked but there are
no
  guarantees that the current solver state is the one you are looki
ng
  for.
* If you modify the underlying model after an optimization, you wil
l
  retrieve the old optimization values.

Raises

RuntimeError
    If the underlying model was never optimized beforehand or the
    reaction is not part of a model.
OptimizationError
    If the solver status is anything other than 'optimal'.

Examples

>>> import cobra.test
>>> model = cobra.test.create_test_model("textbook")
>>> solution = model.optimize()
>>> model.reactions.PFK.reduced_cost
-8.673617379884035e-18

```
>>> solution.reduced_costs.PFK
-8.6736173798840355e-18
```

reverse_id

```
Generate the id of reverse_variable from the reaction's id.
```

reverse_variable

```
An optlang variable representing the reverse flux

Returns
-------
optlang.interface.Variable
    An optlang variable for the reverse flux or None if reaction is
    not associated with a model.
```

reversibility

```
Whether the reaction can proceed in both directions (reversible)

This is computed from the current upper and lower bounds.
```

upper_bound

```
Get or set the upper bound

Setting the upper bound (float) will also adjust the associated opt
lang
variables associated with the reaction. Infeasible combinations,
such as a upper bound lower than the current lower bound will
update the other bound.

When using a `HistoryManager` context, this attribute can be set
temporarily, reversed when the exiting the context.
```

x

```
The flux through the reaction in the most recent solution.

Flux values are computed from the primal values of the variables in
the solution.
```

y

```
The reduced cost of the reaction in the most recent solution.

Reduced costs are computed from the dual values of the variables in
the solution.
```

Methods inherited from cobra.core.object.Object:

__getstate__(self)

```
To prevent excessive replication during deepcopy.
```

__repr__(self)

Data descriptors inherited from cobra.core.object.Object:

__dict__

```
dictionary for instance variables (if defined)
```

__weakref__

```
list of weak references to the object (if defined)
```

id

Log_Concentration

Data and other attributes defined here:
prefix = 'LC_'

Methods inherited from MetaboliteVariable:
__init__(self, metabolite, **kwargs)

Data descriptors inherited from MetaboliteVariable:
id
model

Methods inherited from GenericVariable:
__add__(self, other)
```
Adding either two variables together or a variable and a numeric
results in a new variable
:param other:
:return: a new Generic Variable
```
__mul__(self, other)
```
Multiplying either two variables together or a variable and a numeric
results in a new variable
:param other:
:return: a new Generic Variable
```
__radd__(self, other)
```
Take priority on symmetric arithmetic operation
:param other:
:return:
```
__repr__(self)
__rmul__(self, other)
```
Take priority on symmetric arithmetic operation
:param other:
:return:
```
__rsub__(self, other)
```
Take priority on symmetric arithmetic operation
:param other:
:return:
```
__rtruediv__(self, other)
```
Take priority on symmetric arithmetic operation
:param other:
:return:
```
__sub__(self, other)
```
Substracting either two variables together or a variable and a numeri
c
results in a new variable
:param other:
:return: a new Generic Variable
```
__truediv__(self, other)
```
Dividing either two variables together or a variable and a numeric
results in a new variable
:param other:
:return: a new Generic Variable
```
get_interface(self)
```
Called upon completion of __init__, initializes the value of self.var
,
which is returned upon call, and stores the actual interfaced variabl
```

e.

:return: instance of Variable from the problem

get_operand(self, other)

For operations, choose if the operand is a GenericVariable, in which we return its optlang variable, or something else (presumably a numeric)
and we let optlang decide what to do

:param other:
:return:

make_name(self)

Needs to be overridden by the subclass, concats the id with a prefix

:return: None

make_result(self, new_variable)

Returns a Sympy expression
:param new_variable:
:return:

test_consistency(self, other)

Tests whether a candidate to an operation is of the right type and is from the same problem

:param other: an object
:return: None

Data descriptors inherited from GenericVariable:

__attrname__

Name the attribute the instances will have
Example: GenericVariable -> generic_variable
:return:

name

variable

Variable

Methods defined here:

__getstate__(self)

__init__(self, name, lb=None, ub=None, type='continuous', problem=None, *args, **kwargs)

__reduce__(self)

__repr__(self)

```
Does exactly the same as __str__ for now.
```

__setstate__(self, state)

__str__(self)

```
Print a string representation of variable.

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

set_bounds(self, lb, ub)

```
Change the lower and upper bounds of a variable.
```

to_json(self)

```
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)
```

Class methods defined here:

clone(cls, variable, **kwargs) from sympy.core.assumptions.ManagedProperties

```
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)
```

from_json(cls, json_obj) from sympy.core.assumptions.ManagedProperties

```
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))
```

Data descriptors defined here:

dual

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

lb

```
Lower bound of variable.
```

name

```
Name of variable.
```

primal

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

type
    Variable type ('either continuous, integer, or binary'.)
ub
    Upper bound of variable.

Data and other attributes defined here:
default_assumptions = {}

Static methods inherited from optlang.symbolics.Symbol:
__new__(cls, name, **kwargs)

Data descriptors inherited from optlang.symbolics.Symbol:
__dict__
    dictionary for instance variables (if defined)
__weakref__
    list of weak references to the object (if defined)

Methods inherited from sympy.core.symbol.Symbol:
__call__(self, *args)
__getnewargs__(self)
__new_stage2__(cls, name, **assumptions)
as_dummy(self)
    Return a Dummy having the same name and same assumptions as self.
as_real_imag(self, deep=True, **hints)
is_constant(self, *wrt, **flags)
sort_key(...)

Static methods inherited from sympy.core.symbol.Symbol:
__xnew__ = __new_stage2__(cls, name, **assumptions)

Data descriptors inherited from sympy.core.symbol.Symbol:
assumptions0
free_symbols

Data and other attributes inherited from sympy.core.symbol.Symbol:
is_Symbol = True
is_comparable = False
is_symbol = True

Data and other attributes inherited from sympy.core.expr.AtomicExpr:
is_Atom = True
is_number = False

Methods inherited from sympy.core.basic.Atom:
doit(self, **hints)
matches(self, expr, repl_dict={}, old=False)
xreplace(self, rule, hack2=False)

Class methods inherited from sympy.core.basic.Atom:
class_key(cls) from sympy.core.assumptions.ManagedProperties

Methods inherited from sympy.core.expr.Expr:
__abs__(self)
__add__(a, b)
__complex__(self)
__div__(a, b)
__float__(self)
__floordiv__(a, b)
__ge__(self, other)
__gt__(self, other)
__int__(self)
__le__(self, other)
__long__ = __int__(self)
__lt__(self, other)
__mod__(a, b)
__mul__(a, b)
__neg__(self)
__pos__(self)
__pow__(a, b)
__radd__(a, b)
__rdiv__(a, b)
__rfloordiv__(a, b)
__rmod__(a, b)
__rmul__(a, b)
__rpow__(a, b)
__rsub__(a, b)
__rtruediv__ = __rdiv__(a, b)
__sub__(a, b)
__truediv__ = __div__(a, b)
adjoint(self)
apart(self, x=None, **args)

```
See the apart function in sympy.polys
```

args_cnc(self, cset=False, warn=True, split_1=True)

```
Return [commutative factors, non-commutative factors] of self.

self is treated as a Mul and the ordering of the factors is maintaine
d.
If ``cset`` is True the commutative factors will be returned in a set
.
If there were repeated factors (as may happen with an unevaluated Mul
)
then an error will be raised unless it is explicitly supressed by
setting ``warn`` to False.

Note: -1 is always separated from a Number unless split_1 is False.

>>> from sympy import symbols, oo
>>> A, B = symbols('A B', commutative=0)
>>> x, y = symbols('x y')
>>> (-2*x*y).args_cnc()
[[-1, 2, x, y], []]
>>> (-2.5*x).args_cnc()
[[-1, 2.5, x], []]
>>> (-2*x*A*B*y).args_cnc()
[[-1, 2, x, y], [A, B]]
```

```
>>> (-2*x*A*B*y).args_cnc(split_1=False)
[[-2, x, y], [A, B]]
>>> (-2*x*y).args_cnc(cset=True)
[{-1, 2, x, y}, []]

The arg is always treated as a Mul:

>>> (-2 + x + A).args_cnc()
[[], [x - 2 + A]]
>>> (-oo).args_cnc() # -oo is a singleton
[[-1, oo], []]
```

as_base_exp(self)

as_coeff_Add(self, rational=False)

```
Efficiently extract the coefficient of a summation.
```

as_coeff_Mul(self, rational=False)

```
Efficiently extract the coefficient of a product.
```

as_coeff_add(self, *deps)

```
Return the tuple (c, args) where self is written as an Add, ``a``.

c should be a Rational added to any terms of the Add that are
independent of deps.

args should be a tuple of all other terms of ``a``; args is empty
if self is a Number or if self is independent of deps (when given).

This should be used when you don't know if self is an Add or not but
you want to treat self as an Add or if you want to process the
individual arguments of the tail of self as an Add.

- if you know self is an Add and want only the head, use self.args[0]
;
- if you don't want to process the arguments of the tail but need the
  tail then use self.as_two_terms() which gives the head and tail.
- if you want to split self into an independent and dependent parts
  use ``self.as_independent(*deps)``

>>> from sympy import S
>>> from sympy.abc import x, y
>>> (S(3)).as_coeff_add()
(3, ())
>>> (3 + x).as_coeff_add()
(3, (x,))
>>> (3 + x + y).as_coeff_add(x)
(y + 3, (x,))
>>> (3 + y).as_coeff_add(x)
(y + 3, ())
```

as_coeff_exponent(self, x)

```
``c*x**e -> c,e`` where x can be any symbolic expression.
```

as_coeff_mul(self, *deps, **kwargs)

```
Return the tuple (c, args) where self is written as a Mul, ``m``.

c should be a Rational multiplied by any factors of the Mul that are
independent of deps.

args should be a tuple of all other factors of m; args is empty
if self is a Number or if self is independent of deps (when given).

This should be used when you don't know if self is a Mul or not but
you want to treat self as a Mul or if you want to process the
individual arguments of the tail of self as a Mul.
```

```
    - if you know self is a Mul and want only the head, use self.args[0];
    - if you don't want to process the arguments of the tail but need the
      tail then use self.as_two_terms() which gives the head and tail;
    - if you want to split self into an independent and dependent parts
      use ``self.as_independent(*deps)``

    >>> from sympy import S
    >>> from sympy.abc import x, y
    >>> (S(3)).as_coeff_mul()
    (3, ())
    >>> (3*x*y).as_coeff_mul()
    (3, (x, y))
    >>> (3*x*y).as_coeff_mul(x)
    (3*y, (x,))
    >>> (3*y).as_coeff_mul(x)
    (3*y, ())
```

as_coefficient(self, expr)

```
    Extracts symbolic coefficient at the given expression. In
    other words, this functions separates 'self' into the product
    of 'expr' and 'expr'-free coefficient. If such separation
    is not possible it will return None.

    Examples
    ========

    >>> from sympy import E, pi, sin, I, Poly
    >>> from sympy.abc import x

    >>> E.as_coefficient(E)
    1
    >>> (2*E).as_coefficient(E)
    2
    >>> (2*sin(E)*E).as_coefficient(E)

    Two terms have E in them so a sum is returned. (If one were
    desiring the coefficient of the term exactly matching E then
    the constant from the returned expression could be selected.
    Or, for greater precision, a method of Poly can be used to
    indicate the desired term from which the coefficient is
    desired.)

    >>> (2*E + x*E).as_coefficient(E)
    x + 2
    >>> _.args[0]  # just want the exact match
    2
    >>> p = Poly(2*E + x*E); p
    Poly(x*E + 2*E, x, E, domain='ZZ')
    >>> p.coeff_monomial(E)
    2
    >>> p.nth(0, 1)
    2

    Since the following cannot be written as a product containing
    E as a factor, None is returned. (If the coefficient ``2*x`` is
    desired then the ``coeff`` method should be used.)

    >>> (2*E*x + x).as_coefficient(E)
    >>> (2*E*x + x).coeff(E)
    2*x
```

```
>>> (E*(x + 1) + x).as_coefficient(E)

>>> (2*pi*I).as_coefficient(pi*I)
2
>>> (2*I).as_coefficient(pi*I)

See Also
========

coeff: return sum of terms have a given factor
as_coeff_Add: separate the additive constant from an expression
as_coeff_Mul: separate the multiplicative constant from an expression
as_independent: separate x-dependent terms/factors from others
sympy.polys.polytools.coeff_monomial: efficiently find the single coe
fficient of a monomial in Poly
sympy.polys.polytools.nth: like coeff_monomial but powers of monomial
 terms are used
```

as_coefficients_dict(self)

```
Return a dictionary mapping terms to their Rational coefficient.
Since the dictionary is a defaultdict, inquiries about terms which
were not present will return a coefficient of 0. If an expression is
not an Add it is considered to have a single term.

Examples
========

>>> from sympy.abc import a, x
>>> (3*x + a*x + 4).as_coefficients_dict()
{1: 4, x: 3, a*x: 1}
>>> _[a]
0
>>> (3*a*x).as_coefficients_dict()
{a*x: 3}
```

as_content_primitive(self, radical=False, clear=True)

```
This method should recursively remove a Rational from all arguments
and return that (content) and the new self (primitive). The content
should always be positive and ``Mul(*foo.as_content_primitive()) == f
oo``.
The primitive need no be in canonical form and should try to preserve
the underlying structure if possible (i.e. expand_mul should not be
applied to self).

Examples
========

>>> from sympy import sqrt
>>> from sympy.abc import x, y, z

>>> eq = 2 + 2*x + 2*y*(3 + 3*y)

The as_content_primitive function is recursive and retains structure:

>>> eq.as_content_primitive()
(2, x + 3*y*(y + 1) + 1)

Integer powers will have Rationals extracted from the base:

>>> ((2 + 6*x)**2).as_content_primitive()
(4, (3*x + 1)**2)
>>> ((2 + 6*x)**(2*y)).as_content_primitive()
(1, (2*(3*x + 1))**(2*y))
```

```
Terms may end up joining once their as_content_primitives are added:

>>> ((5*(x*(1 + y)) + 2*x*(3 + 3*y))).as_content_primitive()
(11, x*(y + 1))
>>> ((3*(x*(1 + y)) + 2*x*(3 + 3*y))).as_content_primitive()
(9, x*(y + 1))
>>> ((3*(z*(1 + y)) + 2.0*x*(3 + 3*y))).as_content_primitive()
(1, 6.0*x*(y + 1) + 3*z*(y + 1))
>>> ((5*(x*(1 + y)) + 2*x*(3 + 3*y))**2).as_content_primitive()
(121, x**2*(y + 1)**2)
>>> ((5*(x*(1 + y)) + 2.0*x*(3 + 3*y))**2).as_content_primitive()
(1, 121.0*x**2*(y + 1)**2)

Radical content can also be factored out of the primitive:

>>> (2*sqrt(2) + 4*sqrt(10)).as_content_primitive(radical=True)
(2, sqrt(2)*(1 + 2*sqrt(5)))

If clear=False (default is True) then content will not be removed
from an Add if it can be distributed to leave one or more
terms with integer coefficients.

>>> (x/2 + y).as_content_primitive()
(1/2, x + 2*y)
>>> (x/2 + y).as_content_primitive(clear=False)
(1, x/2 + y)
```

as_expr(self, *gens)

```
Convert a polynomial to a SymPy expression.

Examples
========

>>> from sympy import sin
>>> from sympy.abc import x, y

>>> f = (x**2 + x*y).as_poly(x, y)
>>> f.as_expr()
x**2 + x*y

>>> sin(x).as_expr()
sin(x)
```

as_independent(self, *deps, **hint)

```
A mostly naive separation of a Mul or Add into arguments that are not
are dependent on deps. To obtain as complete a separation of variable
s
as possible, use a separation method first, e.g.:

* separatevars() to change Mul, Add and Pow (including exp) into Mul
* .expand(mul=True) to change Add or Mul into Add
* .expand(log=True) to change log expr into an Add

The only non-
naive thing that is done here is to respect noncommutative
ordering of variables and to always return (0, 0) for `self` of zero
regardless of hints.

For nonzero `self`, the returned tuple (i, d) has the
following interpretation:

* i will has no variable that appears in deps
```

```
* d will be 1 or else have terms that contain variables that are in d
eps
* if self is an Add then self = i + d
* if self is a Mul then self = i*d
* otherwise (self, S.One) or (S.One, self) is returned.

To force the expression to be treated as an Add, use the hint as_Add=
True

Examples
========

-- self is an Add

>>> from sympy import sin, cos, exp
>>> from sympy.abc import x, y, z

>>> (x + x*y).as_independent(x)
(0, x*y + x)
>>> (x + x*y).as_independent(y)
(x, x*y)
>>> (2*x*sin(x) + y + x + z).as_independent(x)
(y + z, 2*x*sin(x) + x)
>>> (2*x*sin(x) + y + x + z).as_independent(x, y)
(z, 2*x*sin(x) + x + y)

-- self is a Mul

>>> (x*sin(x)*cos(y)).as_independent(x)
(cos(y), x*sin(x))

non-
commutative terms cannot always be separated out when self is a Mul

>>> from sympy import symbols
>>> n1, n2, n3 = symbols('n1 n2 n3', commutative=False)
>>> (n1 + n1*n2).as_independent(n2)
(n1, n1*n2)
>>> (n2*n1 + n1*n2).as_independent(n2)
(0, n1*n2 + n2*n1)
>>> (n1*n2*n3).as_independent(n1)
(1, n1*n2*n3)
>>> (n1*n2*n3).as_independent(n2)
(n1, n2*n3)
>>> ((x-n1)*(x-y)).as_independent(x)
(1, (x - y)*(x - n1))

-- self is anything else:

>>> (sin(x)).as_independent(x)
(1, sin(x))
>>> (sin(x)).as_independent(y)
(sin(x), 1)
>>> exp(x+y).as_independent(x)
(1, exp(x + y))

-- force self to be treated as an Add:

>>> (3*x).as_independent(x, as_Add=True)
(0, 3*x)
```

```
-- force self to be treated as a Mul:

>>> (3+x).as_independent(x, as_Add=False)
(1, x + 3)
>>> (-3+x).as_independent(x, as_Add=False)
(1, x - 3)

Note how the below differs from the above in making the
constant on the dep term positive.

>>> (y*(-3+x)).as_independent(x)
(y, x - 3)

-- use .as_independent() for true independence testing instead
   of .has(). The former considers only symbols in the free
   symbols while the latter considers all symbols

>>> from sympy import Integral
>>> I = Integral(x, (x, 1, 2))
>>> I.has(x)
True
>>> x in I.free_symbols
False
>>> I.as_independent(x) == (I, 1)
True
>>> (I + x).as_independent(x) == (I, x)
True

Note: when trying to get independent terms, a separation method
might need to be used first. In this case, it is important to keep
track of what you send to this routine so you know how to interpret
the returned values

>>> from sympy import separatevars, log
>>> separatevars(exp(x+y)).as_independent(x)
(exp(y), exp(x))
>>> (x + x*y).as_independent(y)
(x, x*y)
>>> separatevars(x + x*y).as_independent(y)
(x, y + 1)
>>> (x*(1 + y)).as_independent(y)
(x, y + 1)
>>> (x*(1 + y)).expand(mul=True).as_independent(y)
(x, x*y)
>>> a, b=symbols('a b', positive=True)
>>> (log(a*b).expand(log=True)).as_independent(b)
(log(a), log(b))

See Also
========
.separatevars(), .expand(log=True), Add.as_two_terms(),
Mul.as_two_terms(), .as_coeff_add(), .as_coeff_mul()
```

as_leading_term(...)

```
Returns the leading (nonzero) term of the series expansion of self.

The _eval_as_leading_term routines are used to do this, and they must
always return a non-zero value.

Examples
========
```

```
>>> from sympy.abc import x
>>> (1 + x + x**2).as_leading_term(x)
1
>>> (1/x**2 + x + x**2).as_leading_term(x)
x**(-2)
```

as_numer_denom(self)

```
expression -> a/b -> a, b

This is just a stub that should be defined by
an object's class methods to get anything else.

See Also
========
normal: return a/b instead of a, b
```

as_ordered_factors(self, order=None)

```
Return list of ordered factors (if Mul) else [self].
```

as_ordered_terms(self, order=None, data=False)

```
Transform an expression to an ordered list of terms.

Examples
========

>>> from sympy import sin, cos
>>> from sympy.abc import x

>>> (sin(x)**2*cos(x) + sin(x)**2 + 1).as_ordered_terms()
[sin(x)**2*cos(x), sin(x)**2, 1]
```

as_powers_dict(self)

```
Return self as a dictionary of factors with each factor being
treated as a power. The keys are the bases of the factors and the
values, the corresponding exponents. The resulting dictionary should
be used with caution if the expression is a Mul and contains non-
commutative factors since the order that they appeared will be lost i
n
the dictionary.
```

as_terms(self)

```
Transform an expression to a list of terms.
```

cancel(self, *gens, **args)

```
See the cancel function in sympy.polys
```

coeff(self, x, n=1, right=False)

```
Returns the coefficient from the term(s) containing ``x**n``. If ``n`
`
is zero then all terms independent of ``x`` will be returned.

When ``x`` is noncommutative, the coefficient to the left (default) o
r
right of ``x`` can be returned. The keyword 'right' is ignored when
``x`` is commutative.

See Also
========

as_coefficient: separate the expression into a coefficient and factor
as_coeff_Add: separate the additive constant from an expression
as_coeff_Mul: separate the multiplicative constant from an expression
as_independent: separate x-dependent terms/factors from others
sympy.polys.polytools.coeff_monomial: efficiently find the single coe
fficient of a monomial in Poly
sympy.polys.polytools.nth: like coeff_monomial but powers of monomial
 terms are used
```

Examples
========

```
>>> from sympy import symbols
>>> from sympy.abc import x, y, z
```

You can select terms that have an explicit negative in front of them:

```
>>> (-x + 2*y).coeff(-1)
x
>>> (x - 2*y).coeff(-1)
2*y
```

You can select terms with no Rational coefficient:

```
>>> (x + 2*y).coeff(1)
x
>>> (3 + 2*x + 4*x**2).coeff(1)
0
```

You can select terms independent of x by making n=0; in this case
expr.as_independent(x)[0] is returned (and 0 will be returned instead
of None):

```
>>> (3 + 2*x + 4*x**2).coeff(x, 0)
3
>>> eq = ((x + 1)**3).expand() + 1
>>> eq
x**3 + 3*x**2 + 3*x + 2
>>> [eq.coeff(x, i) for i in reversed(range(4))]
[1, 3, 3, 2]
>>> eq -= 2
>>> [eq.coeff(x, i) for i in reversed(range(4))]
[1, 3, 3, 0]
```

You can select terms that have a numerical term in front of them:

```
>>> (-x - 2*y).coeff(2)
-y
>>> from sympy import sqrt
>>> (x + sqrt(2)*x).coeff(sqrt(2))
x
```

The matching is exact:

```
>>> (3 + 2*x + 4*x**2).coeff(x)
2
>>> (3 + 2*x + 4*x**2).coeff(x**2)
4
>>> (3 + 2*x + 4*x**2).coeff(x**3)
0
>>> (z*(x + y)**2).coeff((x + y)**2)
z
>>> (z*(x + y)**2).coeff(x + y)
0
```

In addition, no factoring is done, so 1 + z*(1 + y) is not obtained
from the following:

```
>>> (x + z*(x + x*y)).coeff(x)
```

```
1

If such factoring is desired, factor_terms can be used first:

>>> from sympy import factor_terms
>>> factor_terms(x + z*(x + x*y)).coeff(x)
z*(y + 1) + 1

>>> n, m, o = symbols('n m o', commutative=False)
>>> n.coeff(n)
1
>>> (3*n).coeff(n)
3
>>> (n*m + m*n*m).coeff(n) # = (1 + m)*n*m
1 + m
>>> (n*m + m*n*m).coeff(n, right=True) # = (1 + m)*n*m
m

If there is more than one possible coefficient 0 is returned:

>>> (n*m + m*n).coeff(n)
0

If there is only one possible coefficient, it is returned:

>>> (n*m + x*m*n).coeff(m*n)
x
>>> (n*m + x*m*n).coeff(m*n, right=1)
1
```

collect(self, syms, func=None, evaluate=True, exact=False, distribute_order_term=True)

```
See the collect function in sympy.simplify
```

combsimp(self)

```
See the combsimp function in sympy.simplify
```

compute_leading_term(self, x, logx=None)

```
as_leading_term is only allowed for results of .series()
This is a wrapper to compute a series first.
```

conjugate(self)

could_extract_minus_sign(self)

```
Canonical way to choose an element in the set {e, -e} where
e is any expression. If the canonical element is e, we have
e.could_extract_minus_sign() == True, else
e.could_extract_minus_sign() == False.

For any expression, the set ``{e.could_extract_minus_sign(),
(-e).could_extract_minus_sign()}`` must be ``{True, False}``.

>>> from sympy.abc import x, y
>>> (x-y).could_extract_minus_sign() != (y-
x).could_extract_minus_sign()
True
```

count_ops(self, visual=None)

```
wrapper for count_ops that returns the operation count.
```

diff(self, *symbols, **assumptions)

equals(self, other, failing_expression=False)

```
Return True if self == other, False if it doesn't, or None. If
failing_expression is True then the expression which did not simplify
to a 0 will be returned instead of None.

If ``self`` is a Number (or complex number) that is not zero, then
```

```
the result is False.

If ``self`` is a number and has not evaluated to zero, evalf will be
used to test whether the expression evaluates to zero. If it does so
and the result has significance (i.e. the precision is either -1, for
a Rational result, or is greater than 1) then the evalf value will be
used to return True or False.
```

expand(...)

```
Expand an expression using hints.

See the docstring of the expand() function in sympy.core.function for
more information.
```

extract_additively(self, c)

```
Return self - c if it's possible to subtract c from self and
make all matching coefficients move towards zero, else return None.

Examples
========

>>> from sympy.abc import x, y
>>> e = 2*x + 3
>>> e.extract_additively(x + 1)
x + 2
>>> e.extract_additively(3*x)
>>> e.extract_additively(4)
>>> (y*(x + 1)).extract_additively(x + 1)
>>> ((x + 1)*(x + 2*y + 1) + 3).extract_additively(x + 1)
(x + 1)*(x + 2*y) + 3

Sometimes auto-expansion will return a less simplified result
than desired; gcd_terms might be used in such cases:

>>> from sympy import gcd_terms
>>> (4*x*(y + 1) + y).extract_additively(x)
4*x*(y + 1) + x*(4*y + 3) - x*(4*y + 4) + y
>>> gcd_terms(_)
x*(4*y + 3) + y

See Also
========
extract_multiplicatively
coeff
as_coefficient
```

extract_branch_factor(self, allow_half=False)

```
Try to write self as ``exp_polar(2*pi*I*n)*z`` in a nice way.
Return (z, n).

>>> from sympy import exp_polar, I, pi
>>> from sympy.abc import x, y
>>> exp_polar(I*pi).extract_branch_factor()
(exp_polar(I*pi), 0)
>>> exp_polar(2*I*pi).extract_branch_factor()
(1, 1)
>>> exp_polar(-pi*I).extract_branch_factor()
(exp_polar(I*pi), -1)
>>> exp_polar(3*pi*I + x).extract_branch_factor()
(exp_polar(x + I*pi), 1)
>>> (y*exp_polar(-
5*pi*I)*exp_polar(3*pi*I + 2*pi*x)).extract_branch_factor()
(y*exp_polar(2*pi*x), -1)
```

```
>>> exp_polar(-I*pi/2).extract_branch_factor()
(exp_polar(-I*pi/2), 0)

If allow_half is True, also extract exp_polar(I*pi):

>>> exp_polar(I*pi).extract_branch_factor(allow_half=True)
(1, 1/2)
>>> exp_polar(2*I*pi).extract_branch_factor(allow_half=True)
(1, 1)
>>> exp_polar(3*I*pi).extract_branch_factor(allow_half=True)
(1, 3/2)
>>> exp_polar(-I*pi).extract_branch_factor(allow_half=True)
(1, -1/2)
```

extract_multiplicatively(self, c)

```
Return None if it's not possible to make self in the form
c * something in a nice way, i.e. preserving the properties
of arguments of self.

>>> from sympy import symbols, Rational

>>> x, y = symbols('x,y', real=True)

>>> ((x*y)**3).extract_multiplicatively(x**2 * y)
x*y**2

>>> ((x*y)**3).extract_multiplicatively(x**4 * y)

>>> (2*x).extract_multiplicatively(2)
x

>>> (2*x).extract_multiplicatively(3)

>>> (Rational(1, 2)*x).extract_multiplicatively(3)
x/6
```

factor(self, *gens, **args)

```
See the factor() function in sympy.polys.polytools
```

fourier_series(self, limits=None)

```
Compute fourier sine/cosine series of self.

See the docstring of the :func:`fourier_series` in sympy.series.fouri
er
for more information.
```

fps(self, x=None, x0=0, dir=1, hyper=True, order=4, rational=True, full=False)

```
Compute formal power power series of self.

See the docstring of the :func:`fps` function in sympy.series.formal
for
more information.
```

getO(self)

```
Returns the additive O(..) symbol if there is one, else None.
```

getn(self)

```
Returns the order of the expression.

The order is determined either from the O(...) term. If there
is no O(...) term, it returns None.

Examples
========

>>> from sympy import O
```

```
>>> from sympy.abc import x
>>> (1 + x + O(x**2)).getn()
2
>>> (1 + x).getn()
```

integrate(self, *args, **kwargs)

See the integrate function in sympy.integrals

invert(self, g, *gens, **args)

Return the multiplicative inverse of ``self`` mod ``g``
where ``self`` (and ``g``) may be symbolic expressions).

See Also
========
sympy.core.numbers.mod_inverse, sympy.polys.polytools.invert

is_algebraic_expr(self, *syms)

This tests whether a given expression is algebraic or not, in the
given symbols, syms. When syms is not given, all free symbols
will be used. The rational function does not have to be in expanded
or in any kind of canonical form.

This function returns False for expressions that are "algebraic
expressions" with symbolic exponents. This is a simple extension to t
he
is_rational_function, including rational exponentiation.

Examples
========

```
>>> from sympy import Symbol, sqrt
>>> x = Symbol('x', real=True)
>>> sqrt(1 + x).is_rational_function()
False
>>> sqrt(1 + x).is_algebraic_expr()
True
```

This function does not attempt any nontrivial simplifications that ma
y
result in an expression that does not appear to be an algebraic
expression to become one.

```
>>> from sympy import exp, factor
>>> a = sqrt(exp(x)**2 + 2*exp(x) + 1)/(exp(x) + 1)
>>> a.is_algebraic_expr(x)
False
>>> factor(a).is_algebraic_expr()
True
```

See Also
========
is_rational_function()

References
==========

- http://en.wikipedia.org/wiki/Algebraic_expression

is_polynomial(self, *syms)

Return True if self is a polynomial in syms and False otherwise.

This checks if self is an exact polynomial in syms. This function
returns False for expressions that are "polynomials" with symbolic
exponents. Thus, you should be able to apply polynomial algorithms t

```
o
expressions for which this returns True, and Poly(expr, \*syms) shoul
d
work if and only if expr.is_polynomial(\*syms) returns True. The
polynomial does not have to be in expanded form.  If no symbols are
given, all free symbols in the expression will be used.

This is not part of the assumptions system.  You cannot do
Symbol('z', polynomial=True).

Examples
========

>>> from sympy import Symbol
>>> x = Symbol('x')
>>> ((x**2 + 1)**4).is_polynomial(x)
True
>>> ((x**2 + 1)**4).is_polynomial()
True
>>> (2**x + 1).is_polynomial(x)
False


>>> n = Symbol('n', nonnegative=True, integer=True)
>>> (x**n + 1).is_polynomial(x)
False

This function does not attempt any nontrivial simplifications that ma
y
result in an expression that does not appear to be a polynomial to
become one.

>>> from sympy import sqrt, factor, cancel
>>> y = Symbol('y', positive=True)
>>> a = sqrt(y**2 + 2*y + 1)
>>> a.is_polynomial(y)
False
>>> factor(a)
y + 1
>>> factor(a).is_polynomial(y)
True

>>> b = (y**2 + 2*y + 1)/(y + 1)
>>> b.is_polynomial(y)
False
>>> cancel(b)
y + 1
>>> cancel(b).is_polynomial(y)
True

See also .is_rational_function()
```

is_rational_function(self, *syms)

```
Test whether function is a ratio of two polynomials in the given
symbols, syms. When syms is not given, all free symbols will be used.
The rational function does not have to be in expanded or in any kind
of
canonical form.

This function returns False for expressions that are "rational
functions" with symbolic exponents.  Thus, you should be able to call
.as_numer_denom() and apply polynomial algorithms to the result for
```

```
    expressions for which this returns True.

    This is not part of the assumptions system.  You cannot do
    Symbol('z', rational_function=True).

    Examples
    ========

    >>> from sympy import Symbol, sin
    >>> from sympy.abc import x, y

    >>> (x/y).is_rational_function()
    True

    >>> (x**2).is_rational_function()
    True

    >>> (x/sin(y)).is_rational_function(y)
    False

    >>> n = Symbol('n', integer=True)
    >>> (x**n + 1).is_rational_function(x)
    False

    This function does not attempt any nontrivial simplifications that ma
    y
    result in an expression that does not appear to be a rational functio
    n
    to become one.

    >>> from sympy import sqrt, factor
    >>> y = Symbol('y', positive=True)
    >>> a = sqrt(y**2 + 2*y + 1)/y
    >>> a.is_rational_function(y)
    False
    >>> factor(a)
    (y + 1)/y
    >>> factor(a).is_rational_function(y)
    True

    See also is_algebraic_expr().
```

leadterm(self, x)

```
    Returns the leading term a*x**b as a tuple (a, b).

    Examples
    ========

    >>> from sympy.abc import x
    >>> (1+x+x**2).leadterm(x)
    (1, 0)
    >>> (1/x**2+x+x**2).leadterm(x)
    (1, -2)
```

limit(self, x, xlim, dir='+')

```
    Compute limit x->xlim.
```

lseries(self, x=None, x0=0, dir='+', logx=None)

```
    Wrapper for series yielding an iterator of the terms of the series.

    Note: an infinite series will yield an infinite iterator. The followi
    ng,
    for exaxmple, will never terminate. It will just keep printing terms
```

```
of the sin(x) series::

  for term in sin(x).lseries(x):
      print term

The advantage of lseries() over nseries() is that many times you are
just interested in the next term in the series (i.e. the first term f
or
example), but you don't know how many you should ask for in nseries()
using the "n" parameter.

See also nseries().
```

normal(self)

nseries(self, x=None, x0=0, n=6, dir='+', logx=None)

```
Wrapper to _eval_nseries if assumptions allow, else to series.

If x is given, x0 is 0, dir='+', and self has x, then _eval_nseries i
s
called. This calculates "n" terms in the innermost expressions and
then builds up the final series just by "cross-
multiplying" everything
out.

The optional ``logx`` parameter can be used to replace any log(x) in
the
returned series with a symbolic value to avoid evaluating log(x) at 0
. A
symbol to use in place of log(x) should be provided.

Advantage -- it's fast, because we don't have to determine how many
terms we need to calculate in advance.

Disadvantage -- you may end up with less terms than you may have
expected, but the O(x**n) term appended will always be correct and
so the result, though perhaps shorter, will also be correct.

If any of those assumptions is not met, this is treated like a
wrapper to series which will try harder to return the correct
number of terms.

See also lseries().

Examples
========

>>> from sympy import sin, log, Symbol
>>> from sympy.abc import x, y
>>> sin(x).nseries(x, 0, 6)
x - x**3/6 + x**5/120 + O(x**6)
>>> log(x+1).nseries(x, 0, 5)
x - x**2/2 + x**3/3 - x**4/4 + O(x**5)

Handling of the ``logx`` parameter --- in the following example the
expansion fails since ``sin`` does not have an asymptotic expansion
at -oo (the limit of log(x) as x approaches 0):

>>> e = sin(log(x))
>>> e.nseries(x, 0, 6)
Traceback (most recent call last):
...
PoleError: ...
```

```
...
>>> logx = Symbol('logx')
>>> e.nseries(x, 0, 6, logx=logx)
sin(logx)

In the following example, the expansion works but gives only an Order
 term
unless the ``logx`` parameter is used:

>>> e = x**y
>>> e.nseries(x, 0, 2)
O(log(x)**2)
>>> e.nseries(x, 0, 2, logx=logx)
exp(logx*y)
```

nsimplify(self, constants=[], tolerance=None, full=False)

```
See the nsimplify function in sympy.simplify
```

powsimp(self, *args, **kwargs)

```
See the powsimp function in sympy.simplify
```

primitive(self)

```
Return the positive Rational that can be extracted non-recursively
from every term of self (i.e., self is treated like an Add). This is
like the as_coeff_Mul() method but primitive always extracts a positi
ve
Rational (never a negative or a Float).

Examples
========

>>> from sympy.abc import x
>>> (3*(x + 1)**2).primitive()
(3, (x + 1)**2)
>>> a = (6*x + 2); a.primitive()
(2, 3*x + 1)
>>> b = (x/2 + 3); b.primitive()
(1/2, x + 6)
>>> (a*b).primitive() == (1, a*b)
True
```

radsimp(self, **kwargs)

```
See the radsimp function in sympy.simplify
```

ratsimp(self)

```
See the ratsimp function in sympy.simplify
```

refine(self, assumption=True)

```
See the refine function in sympy.assumptions
```

removeO(self)

```
Removes the additive O(..) symbol if there is one
```

round(self, p=0)

```
Return x rounded to the given decimal place.

If a complex number would results, apply round to the real
and imaginary components of the number.

Examples
========

>>> from sympy import pi, E, I, S, Add, Mul, Number
>>> S(10.5).round()
11.
>>> pi.round()
3.
```

```
>>> pi.round(2)
3.14
>>> (2*pi + E*I).round()
6. + 3.*I

The round method has a chopping effect:

>>> (2*pi + I/10).round()
6.
>>> (pi/10 + 2*I).round()
2.*I
>>> (pi/10 + E*I).round(2)
0.31 + 2.72*I

Notes
=====

Do not confuse the Python builtin function, round, with the
SymPy method of the same name. The former always returns a float
(or raises an error if applied to a complex value) while the
latter returns either a Number or a complex number:

>>> isinstance(round(S(123), -2), Number)
False
>>> isinstance(S(123).round(-2), Number)
True
>>> isinstance((3*I).round(), Mul)
True
>>> isinstance((1 + 3*I).round(), Add)
True
```

separate(self, deep=False, force=False)

```
See the separate function in sympy.simplify
```

series(self, x=None, x0=0, n=6, dir='+', logx=None)

```
Series expansion of "self" around ``x = x0`` yielding either terms of
the series one by one (the lazy series given when n=None), else
all the terms at once when n != None.

Returns the series expansion of "self" around the point ``x = x0``
with respect to ``x`` up to ``O((x - x0)**n, x, x0)`` (default n is 6
).

If ``x=None`` and ``self`` is univariate, the univariate symbol will
be supplied, otherwise an error will be raised.

>>> from sympy import cos, exp
>>> from sympy.abc import x, y
>>> cos(x).series()
1 - x**2/2 + x**4/24 + O(x**6)
>>> cos(x).series(n=4)
1 - x**2/2 + O(x**4)
>>> cos(x).series(x, x0=1, n=2)
cos(1) - (x - 1)*sin(1) + O((x - 1)**2, (x, 1))
>>> e = cos(x + exp(y))
>>> e.series(y, n=2)
cos(x + 1) - y*sin(x + 1) + O(y**2)
>>> e.series(x, n=2)
cos(exp(y)) - x*sin(exp(y)) + O(x**2)

If ``n=None`` then a generator of the series terms will be returned.

>>> term=cos(x).series(n=None)
```

```
        >>> [next(term) for i in range(2)]
        [1, -x**2/2]

        For ``dir=+`` (default) the series is calculated from the right and
        for ``dir=-`` the series from the left. For smooth functions this
        flag will not alter the results.

        >>> abs(x).series(dir="+")
        x
        >>> abs(x).series(dir="-")
        -x
```

simplify(self, ratio=1.7, measure=None)

```
        See the simplify function in sympy.simplify
```

taylor_term(self, n, x, *previous_terms)

```
        General method for the taylor term.

        This method is slow, because it differentiates n-
        times. Subclasses can
        redefine it to make it faster by using the "previous_terms".
```

together(self, *args, **kwargs)

```
        See the together function in sympy.polys
```

transpose(self)

trigsimp(self, **args)

```
        See the trigsimp function in sympy.simplify
```

Methods inherited from sympy.logic.boolalg.Boolean:

__and__(self, other)

```
        Overloading for & operator
```

__invert__(self)

```
        Overloading for ~
```

__lshift__(self, other)

```
        Overloading for <<
```

__or__(self, other)

```
        Overloading for |
```

__rand__ = __and__(self, other)

```
        Overloading for & operator
```

__rlshift__ = __rshift__(self, other)

```
        Overloading for >>
```

__ror__ = __or__(self, other)

```
        Overloading for |
```

__rrshift__ = __lshift__(self, other)

```
        Overloading for <<
```

__rshift__(self, other)

```
        Overloading for >>
```

__rxor__ = __xor__(self, other)

__xor__(self, other)

Methods inherited from sympy.core.basic.Basic:

__eq__(self, other)

```
        Return a boolean indicating whether a == b on the basis of
        their symbolic trees.

        This is the same as a.compare(b) == 0 but faster.

        Notes
        =====
```

```
    If a class that overrides __eq__() needs to retain the
    implementation of __hash__() from a parent class, the
    interpreter must be told this explicitly by setting __hash__ =
    <ParentClass>.__hash__. Otherwise the inheritance of __hash__()
    will be blocked, just as if __hash__ had been explicitly set to
    None.

    References
    ==========

    from http://docs.python.org/dev/reference/datamodel.html#object.__has
    h__
```

__hash__(self)

__ne__(self, other)

```
    a != b  -
    > Compare two symbolic trees and see whether they are different

    this is the same as:

      a.compare(b) != 0

    but faster
```

__reduce_ex__(self, proto)

```
    Pickling support.
```

as_poly(self, *gens, **args)

```
    Converts ``self`` to a polynomial or returns ``None``.

    >>> from sympy import sin
    >>> from sympy.abc import x, y

    >>> print((x**2 + x*y).as_poly())
    Poly(x**2 + x*y, x, y, domain='ZZ')

    >>> print((x**2 + x*y).as_poly(x, y))
    Poly(x**2 + x*y, x, y, domain='ZZ')

    >>> print((x**2 + sin(y)).as_poly(x, y))
    None
```

atoms(self, *types)

```
    Returns the atoms that form the current object.

    By default, only objects that are truly atomic and can't
    be divided into smaller pieces are returned: symbols, numbers,
    and number symbols like I and pi. It is possible to request
    atoms of any type, however, as demonstrated below.

    Examples
    ========

    >>> from sympy import I, pi, sin
    >>> from sympy.abc import x, y
    >>> (1 + x + 2*sin(y + I*pi)).atoms()
    {1, 2, I, pi, x, y}

    If one or more types are given, the results will contain only
    those types of atoms.

    Examples
    ========
```

```
>>> from sympy import Number, NumberSymbol, Symbol
>>> (1 + x + 2*sin(y + I*pi)).atoms(Symbol)
{x, y}

>>> (1 + x + 2*sin(y + I*pi)).atoms(Number)
{1, 2}

>>> (1 + x + 2*sin(y + I*pi)).atoms(Number, NumberSymbol)
{1, 2, pi}

>>> (1 + x + 2*sin(y + I*pi)).atoms(Number, NumberSymbol, I)
{1, 2, I, pi}

Note that I (imaginary unit) and zoo (complex infinity) are special
types of number symbols and are not part of the NumberSymbol class.

The type can be given implicitly, too:

>>> (1 + x + 2*sin(y + I*pi)).atoms(x) # x is a Symbol
{x, y}

Be careful to check your assumptions when using the implicit option
since ``S(1).is_Integer = True`` but ``type(S(1))`` is ``One``, a spe
cial type
of sympy atom, while ``type(S(2))`` is type ``Integer`` and will find
 all
integers in an expression:

>>> from sympy import S
>>> (1 + x + 2*sin(y + I*pi)).atoms(S(1))
{1}

>>> (1 + x + 2*sin(y + I*pi)).atoms(S(2))
{1, 2}

Finally, arguments to atoms() can select more than atomic atoms: any
sympy type (loaded in core/__init__.py) can be listed as an argument
and those types of "atoms" as found in scanning the arguments of the
expression recursively:

>>> from sympy import Function, Mul
>>> from sympy.core.function import AppliedUndef
>>> f = Function('f')
>>> (1 + f(x) + 2*sin(y + I*pi)).atoms(Function)
{f(x), sin(y + I*pi)}
>>> (1 + f(x) + 2*sin(y + I*pi)).atoms(AppliedUndef)
{f(x)}

>>> (1 + x + 2*sin(y + I*pi)).atoms(Mul)
{I*pi, 2*sin(y + I*pi)}
```

compare(self, other)

```
Return -
1, 0, 1 if the object is smaller, equal, or greater than other.

Not in the mathematical sense. If the object is of a different type
from the "other" then their classes are ordered according to
the sorted_classes list.

Examples
========
```

```
        >>> from sympy.abc import x, y
        >>> x.compare(y)
        -1
        >>> x.compare(x)
        0
        >>> y.compare(x)
        1
```

copy(self)

count(self, query)

```
        Count the number of matching subexpressions.
```

dummy_eq(self, other, symbol=None)

```
        Compare two expressions and handle dummy symbols.

        Examples
        ========

        >>> from sympy import Dummy
        >>> from sympy.abc import x, y

        >>> u = Dummy('u')

        >>> (u**2 + 1).dummy_eq(x**2 + 1)
        True
        >>> (u**2 + 1) == (x**2 + 1)
        False

        >>> (u**2 + y).dummy_eq(x**2 + y, x)
        True
        >>> (u**2 + y).dummy_eq(x**2 + y, y)
        False
```

find(self, query, group=False)

```
        Find all subexpressions matching a query.
```

has(...)

```
        Test whether any subexpression matches any of the patterns.

        Examples
        ========

        >>> from sympy import sin
        >>> from sympy.abc import x, y, z
        >>> (x**2 + sin(x*y)).has(z)
        False
        >>> (x**2 + sin(x*y)).has(x, y, z)
        True
        >>> x.has(x)
        True

        Note ``has`` is a structural algorithm with no knowledge of
        mathematics. Consider the following half-open interval:

        >>> from sympy.sets import Interval
        >>> i = Interval.Lopen(0, 5); i
        Interval.Lopen(0, 5)
        >>> i.args
        (0, 5, True, False)
        >>> i.has(4)  # there is no "4" in the arguments
        False
        >>> i.has(0)  # there *is* a "0" in the arguments
        True
```

```
    Instead, use ``contains`` to determine whether a number is in the
    interval or not:

    >>> i.contains(4)
    True
    >>> i.contains(0)
    False


    Note that ``expr.has(*patterns)`` is exactly equivalent to
    ``any(expr.has(p) for p in patterns)``. In particular, ``False`` is
    returned when the list of patterns is empty.

    >>> x.has()
    False
```

is_hypergeometric(self, k)

match(self, pattern, old=False)

```
    Pattern matching.

    Wild symbols match all.

    Return ``None`` when expression (self) does not match
    with pattern. Otherwise return a dictionary such that::

      pattern.xreplace(self.match(pattern)) == self

    Examples
    ========

    >>> from sympy import Wild
    >>> from sympy.abc import x, y
    >>> p = Wild("p")
    >>> q = Wild("q")
    >>> r = Wild("r")
    >>> e = (x+y)**(x+y)
    >>> e.match(p**p)
    {p_: x + y}
    >>> e.match(p**q)
    {p_: x + y, q_: x + y}
    >>> e = (2*x)**2
    >>> e.match(p*q**r)
    {p_: 4, q_: x, r_: 2}
    >>> (p*q**r).xreplace(e.match(p*q**r))
    4*x**2

    The ``old`` flag will give the old-style pattern matching where
    expressions and patterns are essentially solved to give the
    match. Both of the following give None unless ``old=True``:

    >>> (x - 2).match(p - x, old=True)
    {p_: 2*x - 2}
    >>> (2/x).match(p*x, old=True)
    {p_: 2/x**2}
```

rcall(self, *args)

```
    Apply on the argument recursively through the expression tree.

    This method is used to simulate a common abuse of notation for
    operators. For instance in SymPy the the following will not work:

    ``(x+Lambda(y, 2*y))(z) == x+2*z``,
```

however you can use

```
>>> from sympy import Lambda
>>> from sympy.abc import x, y, z
>>> (x + Lambda(y, 2*y)).rcall(z)
x + 2*z
```

replace(self, query, value, map=False, simultaneous=True, exact=False)

Replace matching subexpressions of ``self`` with ``value``.

If ``map = True`` then also return the mapping {old: new} where ``old``
was a sub-expression found with query and ``new`` is the replacement
value for it. If the expression itself doesn't match the query, then
the returned value will be ``self.xreplace(map)`` otherwise it should
be ``self.subs(ordered(map.items()))``.

Traverses an expression tree and performs replacement of matching
subexpressions from the bottom to the top of the tree. The default
approach is to do the replacement in a simultaneous fashion so
changes made are targeted only once. If this is not desired or causes
problems, ``simultaneous`` can be set to False. In addition, if an
expression containing more than one Wild symbol is being used to match
subexpressions and the ``exact`` flag is True, then the match will only
succeed if non-zero values are received for each Wild that appears in
the match pattern.

The list of possible combinations of queries and replacement values
is listed below:

Examples
========

Initial setup

```
    >>> from sympy import log, sin, cos, tan, Wild, Mul, Add
    >>> from sympy.abc import x, y
    >>> f = log(sin(x)) + tan(sin(x**2))
```

1.1. type -> type
    obj.replace(type, newtype)

    When object of type ``type`` is found, replace it with the
    result of passing its argument(s) to ``newtype``.

```
    >>> f.replace(sin, cos)
    log(cos(x)) + tan(cos(x**2))
    >>> sin(x).replace(sin, cos, map=True)
    (cos(x), {sin(x): cos(x)})
    >>> (x*y).replace(Mul, Add)
    x + y
```

1.2. type -> func
    obj.replace(type, func)

    When object of type ``type`` is found, apply ``func`` to its
    argument(s). ``func`` must be written to handle the number
    of arguments of ``type``.

```
    >>> f.replace(sin, lambda arg: sin(2*arg))
    log(sin(2*x)) + tan(sin(2*x**2))
    >>> (x*y).replace(Mul, lambda *args: sin(2*Mul(*args)))
    sin(2*x*y)
```

2.1. pattern -> expr

```
    obj.replace(pattern(wild), expr(wild))

    Replace subexpressions matching ``pattern`` with the expression
    written in terms of the Wild symbols in ``pattern``.

    >>> a = Wild('a')
    >>> f.replace(sin(a), tan(a))
    log(tan(x)) + tan(tan(x**2))
    >>> f.replace(sin(a), tan(a/2))
    log(tan(x/2)) + tan(tan(x**2/2))
    >>> f.replace(sin(a), a)
    log(x) + tan(x**2)
    >>> (x*y).replace(a*x, a)
    y

    When the default value of False is used with patterns that have
    more than one Wild symbol, non-intuitive results may be obtained:

    >>> b = Wild('b')
    >>> (2*x).replace(a*x + b, b - a)
    2/x

    For this reason, the ``exact`` option can be used to make the
    replacement only when the match gives non-zero values for all
    Wild symbols:

    >>> (2*x + y).replace(a*x + b, b - a, exact=True)
    y - 2
    >>> (2*x).replace(a*x + b, b - a, exact=True)
    2*x
```

2.2. pattern -> func

```
    obj.replace(pattern(wild), lambda wild: expr(wild))

    All behavior is the same as in 2.1 but now a function in terms of
    pattern variables is used rather than an expression:

    >>> f.replace(sin(a), lambda a: sin(2*a))
    log(sin(2*x)) + tan(sin(2*x**2))
```

3.1. func -> func

```
    obj.replace(filter, func)

    Replace subexpression ``e`` with ``func(e)`` if ``filter(e)``
    is True.

    >>> g = 2*sin(x**3)
    >>> g.replace(lambda expr: expr.is_Number, lambda expr: expr**2)
    4*sin(x**9)
```

The expression itself is also targeted by the query but is done in such a fashion that changes are not made twice.

```
    >>> e = x*(x*y + 1)
    >>> e.replace(lambda x: x.is_Mul, lambda x: 2*x)
```

```
    2*x*(2*x*y + 1)

See Also
========
subs: substitution of subexpressions as defined by the objects
      themselves.
xreplace: exact node replacement in expr tree; also capable of
          using matching rules
```

rewrite(self, *args, **hints)

```
Rewrite functions in terms of other functions.

Rewrites expression containing applications of functions
of one kind in terms of functions of different kind. For
example you can rewrite trigonometric functions as complex
exponentials or combinatorial functions as gamma function.

As a pattern this function accepts a list of functions to
to rewrite (instances of DefinedFunction class). As rule
you can use string or a destination function instance (in
this case rewrite() will use the str() function).

There is also the possibility to pass hints on how to rewrite
the given expressions. For now there is only one such hint
defined called 'deep'. When 'deep' is set to False it will
forbid functions to rewrite their contents.

Examples
========

>>> from sympy import sin, exp
>>> from sympy.abc import x

Unspecified pattern:

>>> sin(x).rewrite(exp)
-I*(exp(I*x) - exp(-I*x))/2

Pattern as a single function:

>>> sin(x).rewrite(sin, exp)
-I*(exp(I*x) - exp(-I*x))/2

Pattern as a list of functions:

>>> sin(x).rewrite([sin, ], exp)
-I*(exp(I*x) - exp(-I*x))/2
```

subs(self, *args, **kwargs)

```
Substitutes old for new in an expression after sympifying args.

`args` is either:
  - two arguments, e.g. foo.subs(old, new)
  - one iterable argument, e.g. foo.subs(iterable). The iterable may
be
     o an iterable container with (old, new) pairs. In this case the
       replacements are processed in the order given with successive
       patterns possibly affecting replacements already made.
     o a dict or set whose key/value items correspond to old/new pair
s.
       In this case the old/new pairs will be sorted by op count and
in
       case of a tie, by number of args and the default_sort_key. The
```

```
       resulting sorted list is then processed as an iterable contain
er
       (see previous).

If the keyword ``simultaneous`` is True, the subexpressions will not
be
evaluated until all the substitutions have been made.

Examples
========

>>> from sympy import pi, exp, limit, oo
>>> from sympy.abc import x, y
>>> (1 + x*y).subs(x, pi)
pi*y + 1
>>> (1 + x*y).subs({x:pi, y:2})
1 + 2*pi
>>> (1 + x*y).subs([(x, pi), (y, 2)])
1 + 2*pi
>>> reps = [(y, x**2), (x, 2)]
>>> (x + y).subs(reps)
6
>>> (x + y).subs(reversed(reps))
x**2 + 2

>>> (x**2 + x**4).subs(x**2, y)
y**2 + y

To replace only the x**2 but not the x**4, use xreplace:

>>> (x**2 + x**4).xreplace({x**2: y})
x**4 + y

To delay evaluation until all substitutions have been made,
set the keyword ``simultaneous`` to True:

>>> (x/y).subs([(x, 0), (y, 0)])
0
>>> (x/y).subs([(x, 0), (y, 0)], simultaneous=True)
nan

This has the added feature of not allowing subsequent substitutions
to affect those already made:

>>> ((x + y)/y).subs({x + y: y, y: x + y})
1
>>> ((x + y)/y).subs({x + y: y, y: x + y}, simultaneous=True)
y/(x + y)

In order to obtain a canonical result, unordered iterables are
sorted by count_op length, number of arguments and by the
default_sort_key to break any ties. All other iterables are left
unsorted.

>>> from sympy import sqrt, sin, cos
>>> from sympy.abc import a, b, c, d, e

>>> A = (sqrt(sin(2*x)), a)
>>> B = (sin(2*x), b)
>>> C = (cos(2*x), c)
>>> D = (x, d)
```

```
>>> E = (exp(x), e)

>>> expr = sqrt(sin(2*x))*sin(exp(x)*x)*cos(2*x) + sin(2*x)

>>> expr.subs(dict([A, B, C, D, E]))
a*c*sin(d*e) + b

The resulting expression represents a literal replacement of the
old arguments with the new arguments. This may not reflect the
limiting behavior of the expression:

>>> (x**3 - 3*x).subs({x: oo})
nan

>>> limit(x**3 - 3*x, x, oo)
oo

If the substitution will be followed by numerical
evaluation, it is better to pass the substitution to
evalf as

>>> (1/x).evalf(subs={x: 3.0}, n=21)
0.333333333333333333333

rather than

>>> (1/x).subs({x: 3.0}).evalf(21)
0.333333333333333314830

as the former will ensure that the desired level of precision is
obtained.

See Also
========
replace: replacement capable of doing wildcard-like matching,
         parsing of match, and conditional replacements
xreplace: exact node replacement in expr tree; also capable of
          using matching rules
evalf: calculates the given formula to a desired level of precision
```

Class methods inherited from sympy.core.basic.Basic:

fromiter(cls, args, **assumptions) from sympy.core.assumptions.ManagedProperties

```
Create a new object from an iterable.

This is a convenience function that allows one to create objects from
any iterable, without having to convert to a list or tuple first.

Examples
========

>>> from sympy import Tuple
>>> Tuple.fromiter(i for i in range(5))
(0, 1, 2, 3, 4)
```

Data descriptors inherited from sympy.core.basic.Basic:

args

```
Returns a tuple of arguments of 'self'.

Examples
========
```

```
>>> from sympy import cot
>>> from sympy.abc import x, y

>>> cot(x).args
(x,)

>>> cot(x).args[0]
x

>>> (x*y).args
(x, y)

>>> (x*y).args[1]
y

Notes
=====

Never use self._args, always use self.args.
Only use _args in __new__ when creating a new function.
Don't override .args() from Basic (so that it's easy to
change the interface in the future if needed).
```

canonical_variables

```
Return a dictionary mapping any variable defined in
``self.variables`` as underscore-suffixed numbers
corresponding to their position in ``self.variables``. Enough
underscores are added to ensure that there will be no clash with
existing free symbols.

Examples
========

>>> from sympy import Lambda
>>> from sympy.abc import x
>>> Lambda(x, 2*x).canonical_variables
{x: 0_}
```

func

```
The top-level function in an expression.

The following should hold for all objects::

    >> x == x.func(*x.args)

Examples
========

>>> from sympy.abc import x
>>> a = 2*x
>>> a.func
<class 'sympy.core.mul.Mul'>
>>> a.args
(2, x)
>>> a.func(*a.args)
2*x
>>> a == a.func(*a.args)
True
```

is_algebraic

is_antihermitian

is_commutative

is_complex
is_composite
is_even
is_finite
is_hermitian
is_imaginary
is_infinite
is_integer
is_irrational
is_negative
is_noninteger
is_nonnegative
is_nonpositive
is_nonzero
is_odd
is_polar
is_positive
is_prime
is_rational
is_real
is_transcendental
is_zero

Data and other attributes inherited from sympy.core.basic.Basic:
is_Add = False
is_AlgebraicNumber = False
is_Boolean = False
is_Derivative = False
is_Dummy = False
is_Equality = False
is_Float = False
is_Function = False
is_Indexed = False
is_Integer = False
is_Matrix = False
is_Mul = False
is_Not = False
is_Number = False
is_NumberSymbol = False
is_Order = False
is_Piecewise = False
is_Point = False
is_Poly = False
is_Pow = False
is_Rational = False
is_Relational = False
is_Vector = False
is_Wild = False

Methods inherited from sympy.core.evalf.EvalfMixin:

evalf(self, n=15, subs=None, maxn=100, chop=False, strict=False, quad=None, verbose=False)

```
    Evaluate the given formula to an accuracy of n digits.
    Optional keyword arguments:

        subs=<dict>
            Substitute numerical values for symbols, e.g.
            subs={x:3, y:1+pi}. The substitutions must be given as a
            dictionary.

        maxn=<integer>
            Allow a maximum temporary working precision of maxn digits
            (default=100)

        chop=<bool>
            Replace tiny real or imaginary parts in subresults
            by exact zeros (default=False)

        strict=<bool>
            Raise PrecisionExhausted if any subresult fails to evaluate
            to full accuracy, given the available maxprec
            (default=False)

        quad=<str>
            Choose algorithm for numerical quadrature. By default,
            tanh-sinh quadrature is used. For oscillatory
            integrals on an infinite interval, try quad='osc'.

        verbose=<bool>
            Print debug information (default=False)
```

n = evalf(self, n=15, subs=None, maxn=100, chop=False, strict=False, quad=None, verbose=False)

```
    Evaluate the given formula to an accuracy of n digits.
    Optional keyword arguments:

        subs=<dict>
            Substitute numerical values for symbols, e.g.
            subs={x:3, y:1+pi}. The substitutions must be given as a
            dictionary.

        maxn=<integer>
            Allow a maximum temporary working precision of maxn digits
            (default=100)

        chop=<bool>
            Replace tiny real or imaginary parts in subresults
            by exact zeros (default=False)

        strict=<bool>
            Raise PrecisionExhausted if any subresult fails to evaluate
            to full accuracy, given the available maxprec
            (default=False)

        quad=<str>
            Choose algorithm for numerical quadrature. By default,
            tanh-sinh quadrature is used. For oscillatory
            integrals on an infinite interval, try quad='osc'.

        verbose=<bool>
            Print debug information (default=False)
```

DictList

Method resolution order:

DictList
__builtin__.list
__builtin__.object

Methods defined here:

__add__(self, other)

```
x.__add__(y) <==> x + y

Parameters
----------
other : iterable
    other must contain only unique id's which do not intersect
    with self
```

__contains__(self, object)

```
DictList.__contains__(object) <==> object in DictList

object: str or :class:`~cobra.core.Object.Object`
```

__copy__(self)

__delitem__(self, index)

__delslice__(self, i, j)

__dir__(self)

__getattr__(self, attr)

__getitem__(self, i)

__getslice__(self, i, j)

__getstate__(self)

```
gets internal state

This is only provided for backwards compatibility so older
versions of cobrapy can load pickles generated with cobrapy. In
reality, the "_dict" state is ignored when loading a pickle
```

__iadd__(self, other)

```
x.__iadd__(y) <==> x += y

Parameters
----------
other : iterable
    other must contain only unique id's whcih do not intersect
    with self
```

__init__(self, *args)

```
Instantiate a combined dict and list.

Parameters
----------
args : iterable
    iterable as single argument to create new DictList from
```

__isub__(self, other)

```
x.__sub__(y) <==> x -= y

Parameters
----------
other : iterable
    other must contain only unique id's present in the list
```

__reduce__(self)

__setitem__(self, i, y)
__setslice__(self, i, j, y)
__setstate__(self, state)
sets internal state

Ignore the passed in state and recalculate it. This is only for
compatibility with older pickles which did not correctly specify
the initialization class
__sub__(self, other)
x.__sub__(y) <==> x - y

Parameters
----
other : iterable
other must contain only unique id's present in the list
add(self, x)
Opposite of `remove`. Mirrors set.add
append(self, object)
append object to end
extend(self, iterable)
extend list by appending elements from the iterable
get_by_any(self, iterable)
Get a list of members using several different ways of indexing

Parameters
----
iterable : list (if not, turned into single element list)
list where each element is either int (referring to an index in
in this DictList), string (a id of a member in this DictList) or
member of this DictList for pass-through

Returns
----
list
a list of members
get_by_id(self, id)
return the element with a matching id
has_id(self, id)
index(self, id, *args)
Determine the position in the list

id: A string or a :class:`~cobra.core.Object.Object`
insert(self, index, object)
insert object before index
list_attr(self, attribute)
return a list of the given attribute for every object
pop(self, *args)
remove and return item at index (default last).
query(self, search_function, attribute=None)
Query the list

Parameters
----
search_function : a string, regular expression or function
Used to find the matching elements in the list.
- a regular expression (possibly compiled), in which case the
given attribute of the object should match the regular expression
.

```
        - a function which takes one argument and returns True for
        desired values

    attribute : string or None
        the name attribute of the object to passed as argument to the
        `search_function`. If this is None, the object itself is used.

    Returns
    -------
    DictList
        a new list of objects which match the query

    Examples
    --------
    >>> import cobra.test
    >>> model = cobra.test.create_test_model('textbook')
    >>> model.reactions.query(lambda x: x.boundary)
    >>> import re
    >>> regex = re.compile('^g', flags=re.IGNORECASE)
    >>> model.metabolites.query(regex, attribute='name')
```

remove(self, x)

```
    .. warning :: Internal use only
```

reverse(self)

```
    reverse *IN PLACE*
```

sort(self, cmp=None, key=None, reverse=False)

```
    stable sort *IN PLACE*

    cmp(x, y) -> -1, 0, 1
```

union(self, iterable)

```
    adds elements with id's not already in the model
```

Data descriptors defined here:

__dict__

```
    dictionary for instance variables (if defined)
```

__weakref__

```
    list of weak references to the object (if defined)
```

Methods inherited from __builtin__.list:

__eq__(...)

```
    x.__eq__(y) <==> x==y
```

__ge__(...)

```
    x.__ge__(y) <==> x>=y
```

__getattribute__(...)

```
    x.__getattribute__('name') <==> x.name
```

__gt__(...)

```
    x.__gt__(y) <==> x>y
```

__imul__(...)

```
    x.__imul__(y) <==> x*=y
```

__iter__(...)

```
    x.__iter__() <==> iter(x)
```

__le__(...)

```
    x.__le__(y) <==> x<=y
```

__len__(...)

```
    x.__len__() <==> len(x)
```

__lt__(...)

```
    x.__lt__(y) <==> x<y
```

__mul__(...)

```
      x.__mul__(n) <==> x*n
```
__ne__(...)
```
      x.__ne__(y) <==> x!=y
```
__repr__(...)
```
      x.__repr__() <==> repr(x)
```
__reversed__(...)
```
      L.__reversed__() -- return a reverse iterator over the list
```
__rmul__(...)
```
      x.__rmul__(n) <==> n*x
```
__sizeof__(...)
```
      L.__sizeof__() -- size of L in memory, in bytes
```
count(...)
```
      L.count(value) -> integer -- return number of occurrences of value
```

Data and other attributes inherited from __builtin__.list:

__hash__ = None

__new__ = <built-in method __new__ of type object>
```
      T.__new__(S, ...) -> a new object with type S, a subtype of T
```