pyTrnsysType

Release 1.1.2-dev

Samuel Letellier-Duchesne

Jul 08, 2019

REFERENCE GUIDE

CHAPTER

ONE

PYTRNSYSTYPE

A python TRNSYS type parser

1.1 Installation

```
pip install pytrnsystype
```

1.2 Usage

Since TRNSYS 18, type proformas can be exported to XML schemas. *pyTrnsysType* builds on this easy to read data structure to easily create TrnsysModel using the most popular scripting language in the data science community: [python](https://www.economist.com/graphic-detail/2018/07/26/python-is-becoming-the-worlds-most-popular-coding-language).

From the xml file of a type proforma, simply create a TrnsysModel object by invoking the *from_xml()* constructor. Make sure to pass a string to the method by reading the *_io.TextIOWrapper* produced by the *open()* method:

```
>>> from pyTrnsysType import TrnsysModel
>>> with open("tests/input_files/Type951.xml") as xml:
...     pipe1 = TrnsysModel.from_xml(xml.read())
```

Calling *pipe1* will display it's Type number and Name:

```
>>> pipe1
Type951: Ecoflex 2-Pipe: Buried Piping System
```

Then, *pipe1* can be used to **get** and **set** attributes such as inputs, outputs and parameters. For example, to set the *Number of Fluid Nodes*, simply set the new value as you would change a dict value:

```
>>> from pyTrnsysType import TrnsysModel
>>> with open("tests/input_files/Type951.xml") as xml:
...     pipe1 = TrnsysModel.from_xml(xml.read())
>>> pipe1.parameters['Number_of_Fluid_Nodes'] = 50
>>> pipe1.parameters['Number_of_Fluid_Nodes']
Number of Fluid Nodes; units=-; value=50
The number of nodes into which each pipe will be divided. Increasing the number of␣
↪nodes will improve the accuracy but cost simulation run-time.
```

Since the *Number of Fluid Nodes* is a cycle parameter, the number of outputs is modified dynamically:

calling *pipe1.outputs* should display 116 Outputs.

The new outputs are now accessible and can also be accessed with loops:

```
for i in range(1,50):
    print(pipe1.outputs["Average_Fluid_Temperature_Pipe_1_{}".format(i)])
```

1.3 Connecting outputs with inputs

Connecting model outputs to other model inputs is quite straightforward and uses a simple mapping technique. For example, to map the first two ouputs of *pipe1* to the first two outputs of *pipe2*, we create a mapping of the form *mapping = {0:0, 1:1}*. In other words, this means that the output 0 of pipe1 is connected to the input 1 of pipe2 and the output 1 of pipe1 is connected to the output 1 of pipe2. Keep in mind that since python traditionally uses 0-based indexing, it has been decided the same logic in this package even though TRNSYS uses 1-based indexing. The package will internally assign the 1-based index.

For convenience, the mapping can also be done using the output/input names such as *mapping = {'Outlet_Air_Temperature': 'Inlet_Air_Temperature', 'Outlet_Air_Humidity_Ratio': 'Inlet_Air_Humidity_Ratio'}*:

```
# First let's create a second pipe, by copying the first one:
pipe2 = pipe1.copy()
# Then, connect pipe1 to pipe2:
pipe1.connect_to(pipe2, mapping={0:0, 1:1})
```

1.4 Simulation Cards

The Simulation Cards is a chuck of code that informs TRNSYS of various simulation constrols such as start time end time and time-step. pyTrnsysType implements many of those *Statements* with a series of Statement objects.

For instance, to create simulation cards using default values, simply call the *all()* constructor:

```
>>> from pyTrnsysType import ControlCards
>>> cc = ControlCards.all()
>>> print(cc)
*** Control Cards
SOLVER 0 1 1          ! Solver statement    Minimum relaxation factor       Maximum␣
↪relaxation factor
MAP                   ! MAP statement
NOLIST                ! NOLIST statement
NOCHECK 0             ! CHECK Statement
DFQ 1                 ! TRNSYS numerical integration solver method
SIMULATION 0 8760 1   ! Start time  End time        Time step
TOLERANCES 0.01 0.01  ! Integration Convergence
LIMITS 25 10 25       ! Max iterations      Max warnings    Trace limit
EQSOLVER 0            ! EQUATION SOLVER statement
```

1.5 Equations

In the TRNSYS studio, equations are components holding a list of user-defined expressions. In pyTrnsysType a similar approach has been taken: the *Equation* class handles the creation of equations and the *EquationCollection* class handles the block of equations. Here's an example:

First, create a series of Equation by invoking the *from_expression* constructor. This allows you two input the equation as a string.

```
>>> from pyTrnsysType import Equation, EquationCollection
>>> equa1 = Equation.from_expression("TdbAmb = [011,001]")
>>> equa2 = Equation.from_expression("rhAmb = [011,007]")
>>> equa3 = Equation.from_expression("Tsky = [011,004]")
>>> equa4 = Equation.from_expression("vWind = [011,008]")
```

One can create

```
>>> equa_col_1 = EquationCollection([equa1, equa2, equa3, equa4],
                                    name='test')
```

1.5.1 Reference

TrnsysModel

MetaData	General information that is associated with a *TrnsysModel*.
ExternalFile	**param question**
ExternalFileCollection	
TrnsysModel	Main Class for holding TRNSYS components.
TypeVariable	Class containing a proforma variable.
TypeCycle	**param role**
CycleCollection	
Parameter	A subclass of *TypeVariable* specific to parameters
Input	A subclass of *TypeVariable* specific to inputs
Output	A subclass of *TypeVariable* specific to outputs
Derivative	the DERIVATIVES for a given TypeModel specify initial values, such as the initial temperatures of various nodes in a thermal storage tank or the initial zone temperatures in a multi zone building.
VariableCollection	A collection of `VariableType` as a dict.
InputCollection	Subclass of *VariableCollection* specific to Inputs
OutputCollection	Subclass of *VariableCollection* specific to Ouputs
ParameterCollection	Subclass of *VariableCollection* specific to Parameters
StudioHeader	Each TrnsysModel has a StudioHeader which handles the studio comments such as position, UNIT_NAME, model, POSITION, LAYER, LINK_STYLE
LinkStyle	{anchors}:{e}:{rgb_int}:{f}:{g}:{h}: 40:20:0:20:1:0:0:0:1:189,462: 432,462: 432,455: 459,455
AnchorPoint	Handles the anchor point.

pyTrnsysType.trnsymodel.MetaData

class pyTrnsysType.trnsymodel.**MetaData**(*object=None*, *author=None*, *organization=None*, *editor=None*, *creationDate=None*, *modifictionDate=None*, *mode=None*, *validation=None*, *icon=None*, *type=None*, *maxInstance=None*, *keywords=None*, *details=None*, *comment=None*, *variables=None*, *plugin=None*, *variablesComment=None*, *cycles=None*, *source=None*, *externalFiles=None*, *model=None*, ***kwargs*)

General information that is associated with a *TrnsysModel*. This information is contained in the General Tab of the Proforma.

Parameters

- **object** (*str*) – A generic name describing the component model.
- **author** (*str*) – The name of the person who wrote the model.
- **organization** (*str*) – The name of organization with which the Author is affiliated.
- **editor** (*str*) – Often, the person creating the Simulation Studio Proforma is not the original author and so the name of the Editor may also be important.
- **creationDate** (*str*) – This is the date of when the model was first written.
- **modifictionDate** (*str*) – This is the date when the Proforma was mostly recently revised.
- **mode** (*int*) – 1-Detailed, 2-Simplified, 3-Empirical, 4- Conventional
- **validation** (*int*) – Determine the type of validation that was performed on this model. This can be 1-qualitative, 2-numerical, 3-analytical, 4-experimental and 5-'in assembly' meaning that it was verified as part of a larger system which was verified.
- **icon** (*Path*) – Path to the icon.
- **type** (*int*) – The type number.
- **maxInstance** (*int*) – The maximum number of instances this type can be used.
- **keywords** (*str*) – keywords associated with this model.
- **details** (*str*) – The detailed description contains an explanation of the model including a mathematical description of the model
- **comment** (*str*) – The text entered here will appear as a comment in the TRNSYS input file. This allows to attach important information about the component to all its users, including users who prefer to edit the input file with a text editor. This text should be short, to avoid overloading the input file.
- **variables** (*dict, optional*) – a list of *TypeVariable*.
- **plugin** (*Path*) – The plug-in path contains the path to the an external application which will be executed to modify component properties instead of the classical properties window.
- **variablesComment** (*str*) – #todo What is this?
- **cycles** (*list, optional*) – List of TypeCycle.
- **source** (*Path*) – Path of the source code.
- **externalFiles** (ExternalFileCollection) – A class handling ExternalFiles for this object.

- **model** (*Path*) – Path of the xml or tmf file.
- ****kwargs** –

classmethod from_tag(*tag*)
Class mothod used to create a TrnsysModel from a xml Tag

Parameters **tag** (*Tag*) – The XML tag with its attributes and contents.

check_extra_tags(*kwargs*)
Detect extra tags in the proforma and warn.

Parameters **kwargs** (*dict*) – dictionary of extra keyword-arguments that would be passed to the constructor.

pyTrnsysType.trnsymodel.ExternalFile

class pyTrnsysType.trnsymodel.**ExternalFile**(*question*, *default*, *answers*, *parameter*, *designate*)

Parameters

- **question** (*str*) –
- **default** (*str*) –
- **answers** (*list of str*) –
- **parameter** (*str*) –
- **designate** (*bool*) – If True, the external files are assigned to logical unit numbers from within the TRNSYS input file. Files that are assigned to a logical unit number using a DESIGNATE statement will not be opened by the TRNSYS kernel.

classmethod from_tag(*tag*)

Parameters **tag** (*Tag*) – The XML tag with its attributes and contents.

pyTrnsysType.trnsymodel.ExternalFileCollection

class pyTrnsysType.trnsymodel.**ExternalFileCollection**(***kwargs*)

classmethod from_dict(*dictionary*)
Construct an *ExternalFileCollection* from a dict of *ExternalFile* objects with the object's id as a key.

Parameters **dictionary** (*dict*) – The dict of {key: *ExternalFile*}

clear() → None. Remove all items from D.

get(*k*[, *d*]) → D[k] if k in D, else d. d defaults to None.

items() → a set-like object providing a view on D's items

keys() → a set-like object providing a view on D's keys

pop(*k*[, *d*]) → v, remove specified key and return the corresponding value.
If key is not found, d is returned if given, otherwise KeyError is raised.

popitem() → (k, v), remove and return some (key, value) pair
as a 2-tuple; but raise KeyError if D is empty.

setdefault (*k*[, *d*]) → D.get(k,d), also set D[k]=d if k not in D

update ([*E*], ***F*) → None. Update D from mapping/iterable E and F.
If E present and has a .keys() method, does: for k in E: D[k] = E[k] If E present and lacks .keys() method, does: for (k, v) in E: D[k] = v In either case, this is followed by: for k, v in F.items(): D[k] = v

values () → an object providing a view on D's values

pyTrnsysType.trnsymodel.TrnsysModel

class pyTrnsysType.trnsymodel.**TrnsysModel** (*meta*, *name*, *studio=None*)
Main Class for holding TRNSYS components. Alone, this __init__ method does not do much. See the *from_xml()* class method for the official constructor of this class.

Parameters

- **meta** (MetaData) – A class containing the model's metadata.
- **name** (*str*) – A user-defined name for this model.
- **studio** (StudioHeader) – A class for handling TRNSYS studio related functions.

classmethod from_xml (*xml*)
Class method to create a *TrnsysModel* from an xml string.

Examples

Simply pass the xml path to the constructor.

```
>>> from pyTrnsysType import TrnsysModel
>>> fan1 = TrnsysModel.from_xml("Tests/input_files/Type146.xml")
```

Parameters **xml** (*str or Path*) – The path of the xml file.

Returns The TRNSYS model.

Return type TrnsysType

copy (*invalidate_connections=True*)
copy object

Parameters **invalidate_connections** (*bool*) – If True, connections to other models will be reset.

connect_to (*other*, *mapping=None*, *link_style=None*)
Connect the outputs of `self` to the inputs of `other`.

Important: Keep in mind that since python traditionally uses 0-based indexing, the same logic is used in this package even though TRNSYS uses traditionally 1-based indexing. The package will internally handle the 1-based index in the output *.dck* file.

Examples

Connect two *TrnsysModel* objects together by creating a mapping of the outputs of pipe_1 to the intputs of pipe_2. In this example we connect output_0 of pipe_1 to input_0 of pipe_2 and output_1 of pipe_1 to input_1 of pipe_2:

```
>>> pipe_1.connect_to(pipe_2, mapping={0:0, 1:1})
```

The same can be acheived using input/output names.

```
>>> pipe_1.connect_to(pipe_2, mapping={'Outlet_Air_Temperature':
>>> 'Inlet_Air_Temperature', 'Outlet_Air_Humidity_Ratio':
>>> 'Inlet_Air_Humidity_Ratio'})
```

Parameters

- **other** (*TrnsysModel*) – The other object
- **mapping** (*dict*) – Mapping of inputs to outputs numbers
- **link_style** (*dict, optional*) –

Raises **TypeError** – A *TypeError is raised when trying to connect to anything other than a :class:'TrnsysModel .*

invalidate_connections()
iterate over inputs/outputs and force _connected_to to None

set_link_style(*other*, *loc='best'*, *color='#1f78b4'*, *path=None*, ***kwargs*)
Set outgoing link styles. Adds a LinkStyle object to the origin Model's studio attribute.

Parameters

- **other** (*TrnsysModel*) – The destination model.
- **loc** (*str or tuple*) – loc (str): The location of the anchor. The strings 'top-left', 'top-right', 'bottom-left', 'bottom-right' place the anchor point at the corresponding corner of the *TrnsysModel*. The strings 'top-center', 'center-right', 'bottom-center', 'center-left' place the anchor point at the edge of the corresponding *TrnsysModel*. The string 'best' places the anchor point at the location, among the eight locations defined so far, with the shortest distance with the destination *TrnsysModel* (other). The location can also be a 2-tuple giving the coordinates of the origin *TrnsysModel* and the destination *TrnsysModel*.
- **color** (*str*) – color string. Can be a single color format string (default='#1f78b4').
- **path** (*LineString or MultiLineString, optional*) – The path of the link.
- ****kwargs** –

set_canvas_position(*pt*)
Set position of self in the canvas. Use cartesian coordinates: origin 0,0 is at bottom-left.

Info: The Studio Canvas origin corresponds to the top-left of the canvas. The x coordinates increase from left to right, while the y coordinates increase from top to bottom.

- top-left = "* $POSITION 0 0"
- bottom-left = "* $POSITION 0 2000"
- top-right = "* $POSITION 2000" 0
- bottom-right = "* $POSITION 2000 2000"

For convenience, users should deal with cartesian coordinates. pyTrnsysType will deal with the transformation.

Parameters **pt** (*Point or 2-tuple*) – The Point geometry or a tuple of (x, y) coordinates.

property inputs
returns the model's inputs.

Type *InputCollection*

property outputs
returns the model's outputs.

Type *OutputCollection*

property derivatives
returns the model's derivatives

Type TypeVariableCollection

property parameters
returns the model's parameters.

Type *ParameterCollection*

property external_files
returns the model's external files

Type *ExternalFileCollection*

property unit_number
Returns the model's unit number (unique)

Type int

property type_number
104 for Type104

Type int

Type Returns the model's type number, eg.

property unit_name
'Type104'

Type str

Type Returns the model's unit name, eg.

property model
The path of this model's proforma

Type str

property anchor_points
Returns the 8-AnchorPoints as a dict with the anchor point location ('top-left', etc.) as a key.

Type dict

property centroid
Returns the model's center Point().

Type Point

pyTrnsysType.trnsymodel.TypeVariable

class pyTrnsysType.trnsymodel.**TypeVariable**(*val*, *order=None*, *name=None*, *role=None*, *dimension=None*, *unit=None*, *type=None*, *min=None*, *max=None*, *boundaries=None*, *default=None*, *symbol=None*, *definition=None*, *model=None*)

Class containing a proforma variable.

Parameters

- **val** (*int, float, _Quantity*) – The actual value holded by this object.
- **order** (*str*) –
- **name** (*str*) – This name will be seen by the user in the connections window and all other variable information windows.
- **role** (*str*) – The role of the variables such as input, output, etc. Changing the role of a standard component requires reprogramming and recompiling the component.
- **dimension** (*str*) – The dimension of the variable (power, temperature, etc.): This dimension must be already defined in the unit dictionary (refer to section 2.9) to be used. The pre-defined dimension 'any' allows to make a variable compatible with any other variable: no checks are performed on such variables if the user attempts to connect them to other variables.
- **unit** (*str*) – The unit of the variable that the TRNSYS program requires for the specified dimension (C, F, K etc.)
- **type** (*type or str*) – The type of the variable: Real, integer, Boolean, or string.
- **min** (*int, float or pint._Quantity*) – The minimum value. The minimum and maximum can be "-INF" or "+INF" to indicate no limit (infinity). +/-INF is the default value.
- **max** (*int, float or pint._Quantity*) – The maximum value. The minimum and maximum can be "-INF" or "+INF" to indicate no limit (infinity). +/-INF is the default value.
- **boundaries** (*str*) – This setting determines if the minimum and maximum are included or not in the range. choices are "[;]", "[;[", "];]" ,"];["
- **default** (*int, float or pint._Quantity*) – the default value of the variable. The default value is replaced by the initial value for the inputs and derivatives and suppressed for the outputs
- **symbol** (*str*) – The symbol of the unit (not used).
- **definition** (*str*) – A short description of the variable.
- **model** –

classmethod from_tag(*tag*, *model=None*)

Class method to create a TypeVariable from an XML tag.

Parameters

- **tag** (*Tag*) – The XML tag with its attributes and contents.
- **model** –

copy()

TypeVariable: Make a copy of `self`

property is_connected
Whether or not this TypeVariable is connected to another type

property connected_to
The TrnsysModel to which this component is connected

property idx
The 0-based index of the TypeVariable

pyTrnsysType.trnsymodel.TypeCycle

class pyTrnsysType.trnsymodel.**TypeCycle**(*role=None*, *firstRow=None*, *lastRow=None*, *cycles=None*, *minSize=None*, *maxSize=None*, *paramName=None*, *question=None*, ***kwargs*)

Parameters

- **role** –
- **firstRow** –
- **lastRow** –
- **cycles** –
- **minSize** –
- **maxSize** –
- **paramName** –
- **question** –
- ****kwargs** –

classmethod from_tag(*tag*)

Parameters **tag** (*Tag*) – The XML tag with its attributes and contents.

property idxs
0-based index of the TypeVariable(s) concerned with this cycle

pyTrnsysType.trnsymodel.CycleCollection

class pyTrnsysType.trnsymodel.**CycleCollection**(*initlist=None*)

append(*item*)
S.append(value) – append value to the end of the sequence

clear() → None – remove all items from S

count(*value*) → integer – return number of occurrences of value

extend(*other*)
S.extend(iterable) – extend sequence by appending elements from the iterable

index(*value*[, *start*[, *stop*]]) → integer – return first index of value.
Raises ValueError if the value is not present.

Supporting start and stop arguments is optional, but recommended.

insert(*i*, *item*)
S.insert(index, value) – insert value before index

pop([*index*]) → item – remove and return item at index (default last).
Raise IndexError if list is empty or index is out of range.

remove(*item*)
S.remove(value) – remove first occurrence of value. Raise ValueError if the value is not present.

reverse()
S.reverse() – reverse *IN PLACE*

pyTrnsysType.trnsymodel.Parameter

class pyTrnsysType.trnsymodel.**Parameter**(*val*, ***kwargs*)
A subclass of *TypeVariable* specific to parameters

A subclass of *TypeVariable* specific to parameters.

Parameters

- **val** –
- ****kwargs** –

property connected_to
The TrnsysModel to which this component is connected

copy()
TypeVariable: Make a copy of `self`

classmethod from_tag(*tag*, *model=None*)
Class method to create a TypeVariable from an XML tag.

Parameters

- **tag** (*Tag*) – The XML tag with its attributes and contents.
- **model** –

property idx
The 0-based index of the TypeVariable

property is_connected
Whether or not this TypeVariable is connected to another type

pyTrnsysType.trnsymodel.Input

class pyTrnsysType.trnsymodel.**Input**(*val*, ***kwargs*)
A subclass of *TypeVariable* specific to inputs

A subclass of *TypeVariable* specific to inputs.

Parameters

- **val** –
- ****kwargs** –

property connected_to
The TrnsysModel to which this component is connected

copy()
TypeVariable: Make a copy of `self`

classmethod from_tag(*tag*, *model=None*)
Class method to create a TypeVariable from an XML tag.

Parameters

- **tag** (*Tag*) – The XML tag with its attributes and contents.
- **model** –

property idx
The 0-based index of the TypeVariable

property is_connected
Whether or not this TypeVariable is connected to another type

pyTrnsysType.trnsymodel.Output

class pyTrnsysType.trnsymodel.**Output**(*val*, ***kwargs*)
A subclass of *TypeVariable* specific to outputs

A subclass of *TypeVariable* specific to outputs.

Parameters

- **val** –
- ****kwargs** –

property connected_to
The TrnsysModel to which this component is connected

copy()
TypeVariable: Make a copy of `self`

classmethod from_tag(*tag*, *model=None*)
Class method to create a TypeVariable from an XML tag.

Parameters

- **tag** (*Tag*) – The XML tag with its attributes and contents.
- **model** –

property idx
The 0-based index of the TypeVariable

property is_connected
Whether or not this TypeVariable is connected to another type

pyTrnsysType.trnsymodel.Derivative

class pyTrnsysType.trnsymodel.**Derivative**(*val*, ***kwargs*)
the DERIVATIVES for a given TypeModel specify initial values, such as the initial temperatures of various nodes in a thermal storage tank or the initial zone temperatures in a multi zone building.

A subclass of *TypeVariable* specific to derivatives.

Parameters

- **val** –
- ****kwargs** –

property connected_to
The TrnsysModel to which this component is connected

copy()
TypeVariable: Make a copy of `self`

classmethod from_tag(*tag*, *model=None*)
Class method to create a TypeVariable from an XML tag.

Parameters

- **tag** (*Tag*) – The XML tag with its attributes and contents.
- **model** –

property idx
The 0-based index of the TypeVariable

property is_connected
Whether or not this TypeVariable is connected to another type

pyTrnsysType.trnsymodel.VariableCollection

class pyTrnsysType.trnsymodel.**VariableCollection**(***kwargs*)
A collection of `VariableType` as a dict. Handles getting and setting variable values.

classmethod from_dict(*dictionary*)

Parameters **dictionary** –

property size
The number of parameters

clear() → None. Remove all items from D.

get(*k*[, *d*]) → D[k] if k in D, else d. d defaults to None.

items() → a set-like object providing a view on D's items

keys() → a set-like object providing a view on D's keys

pop(*k*[, *d*]) → v, remove specified key and return the corresponding value.
If key is not found, d is returned if given, otherwise KeyError is raised.

popitem() → (k, v), remove and return some (key, value) pair
as a 2-tuple; but raise KeyError if D is empty.

setdefault(*k*[, *d*]) → D.get(k,d), also set D[k]=d if k not in D

update([*E*], ***F*) → None. Update D from mapping/iterable E and F.
If E present and has a .keys() method, does: for k in E: D[k] = E[k] If E present and lacks .keys() method, does: for (k, v) in E: D[k] = v In either case, this is followed by: for k, v in F.items(): D[k] = v

values() → an object providing a view on D's values

pyTrnsysType.trnsymodel.InputCollection

class pyTrnsysType.trnsymodel.**InputCollection**

Subclass of *VariableCollection* specific to Inputs

clear() → None. Remove all items from D.

classmethod from_dict(*dictionary*)

Parameters **dictionary** –

get(*k*[, *d*]) → D[k] if k in D, else d. d defaults to None.

items() → a set-like object providing a view on D's items

keys() → a set-like object providing a view on D's keys

pop(*k*[, *d*]) → v, remove specified key and return the corresponding value.
If key is not found, d is returned if given, otherwise KeyError is raised.

popitem() → (k, v), remove and return some (key, value) pair
as a 2-tuple; but raise KeyError if D is empty.

setdefault(*k*[, *d*]) → D.get(k,d), also set D[k]=d if k not in D

property size
The number of parameters

update([*E*], ***F*) → None. Update D from mapping/iterable E and F.
If E present and has a .keys() method, does: for k in E: D[k] = E[k] If E present and lacks .keys() method, does: for (k, v) in E: D[k] = v In either case, this is followed by: for k, v in F.items(): D[k] = v

values() → an object providing a view on D's values

pyTrnsysType.trnsymodel.OutputCollection

class pyTrnsysType.trnsymodel.**OutputCollection**

Subclass of *VariableCollection* specific to Ouputs

clear() → None. Remove all items from D.

classmethod from_dict(*dictionary*)

Parameters **dictionary** –

get(*k*[, *d*]) → D[k] if k in D, else d. d defaults to None.

items() → a set-like object providing a view on D's items

keys() → a set-like object providing a view on D's keys

pop(*k*[, *d*]) → v, remove specified key and return the corresponding value.
If key is not found, d is returned if given, otherwise KeyError is raised.

popitem() → (k, v), remove and return some (key, value) pair
as a 2-tuple; but raise KeyError if D is empty.

setdefault(*k*[, *d*]) → D.get(k,d), also set D[k]=d if k not in D

property size
The number of parameters

update([*E*], ***F*) → None. Update D from mapping/iterable E and F.
If E present and has a .keys() method, does: for k in E: D[k] = E[k] If E present and lacks .keys() method, does: for (k, v) in E: D[k] = v In either case, this is followed by: for k, v in F.items(): D[k] = v

values() → an object providing a view on D's values

pyTrnsysType.trnsymodel.ParameterCollection

class pyTrnsysType.trnsymodel.**ParameterCollection**
Subclass of *VariableCollection* specific to Parameters

clear() → None. Remove all items from D.

classmethod from_dict(*dictionary*)

Parameters **dictionary** –

get(*k*[, *d*]) → D[k] if k in D, else d. d defaults to None.

items() → a set-like object providing a view on D's items

keys() → a set-like object providing a view on D's keys

pop(*k*[, *d*]) → v, remove specified key and return the corresponding value.
If key is not found, d is returned if given, otherwise KeyError is raised.

popitem() → (k, v), remove and return some (key, value) pair
as a 2-tuple; but raise KeyError if D is empty.

setdefault(*k*[, *d*]) → D.get(k,d), also set D[k]=d if k not in D

property size
The number of parameters

update([*E*], ***F*) → None. Update D from mapping/iterable E and F.
If E present and has a .keys() method, does: for k in E: D[k] = E[k] If E present and lacks .keys() method, does: for (k, v) in E: D[k] = v In either case, this is followed by: for k, v in F.items(): D[k] = v

values() → an object providing a view on D's values

pyTrnsysType.trnsymodel.StudioHeader

class pyTrnsysType.trnsymodel.**StudioHeader**(*unit_name*, *model*, *position*, *layer=None*)
Each TrnsysModel has a StudioHeader which handles the studio comments such as position, UNIT_NAME, model, POSITION, LAYER, LINK_STYLE

Parameters

- **unit_name** (*str*) – The unit_name, eg.: "Type104".
- **model** (*Path*) – The path of the tmf/xml file.
- **position** (*Point, optional*) – The Point containing coordinates on the canvas.
- **layer** (*list, optional*) – list of layer names on which the model is placed. Defaults to "Main".

classmethod from_trnsysmodel(*model*)

Parameters **model** (TrnsysModel) –

pyTrnsysType.trnsymodel.LinkStyle

class pyTrnsysType.trnsymodel.**LinkStyle**(*u*, *v*, *loc*, *e*, *rgb*, *f*, *g*, *h*, *path*)
{anchors}:{e}:{rgb_int}:{f}:{g}:{h}: 40:20:0:20:1:0:0:0:1:189,462: 432,462: 432,455: 459,455

Parameters

- **u** (TrnsysModel) – from Model.
- **v** (TrnsysModel) – to Model.
- **loc** (*str or tuple*) – loc (str): The location of the anchor. The strings 'top-left', 'top-right', 'bottom-left', 'bottom-right' place the anchor point at the corresponding corner of the *TrnsysModel*. The strings 'top-center', 'center-right', 'bottom-center', 'center-left' place the anchor point at the edge of the corresponding *TrnsysModel*. The string 'best' places the anchor point at the location, among the eight locations defined so far, with the shortest distance with the destination *TrnsysModel* (other). The location can also be a 2-tuple giving the coordinates of the origin *TrnsysModel* and the destination *TrnsysModel*.
- **e** –
- **rgb** (*tuple*) – The color of the line
- **f** –
- **g** –
- **h** –
- **path** (*LineString or MultiLineString*) –

to_deck()
0:20:40:20:1:0:0:0:1:513,441:471,441:471,430:447,430

pyTrnsysType.trnsymodel.AnchorPoint

class pyTrnsysType.trnsymodel.**AnchorPoint**(*model*, *offset=10*)
Handles the anchor point. There are 6 anchor points around a component

Parameters

- **model** (TrnsysModel) – The TrnsysModel
- **offset** (*float*) – The offset to give the anchor points from the center of the model position.

studio_anchor(*other*, *loc*)
Return the studio anchor based on a location.

Parameters

- **other** – TrnsysModel
- **loc** (*2-tuple*) –

find_best_anchors(*other*)

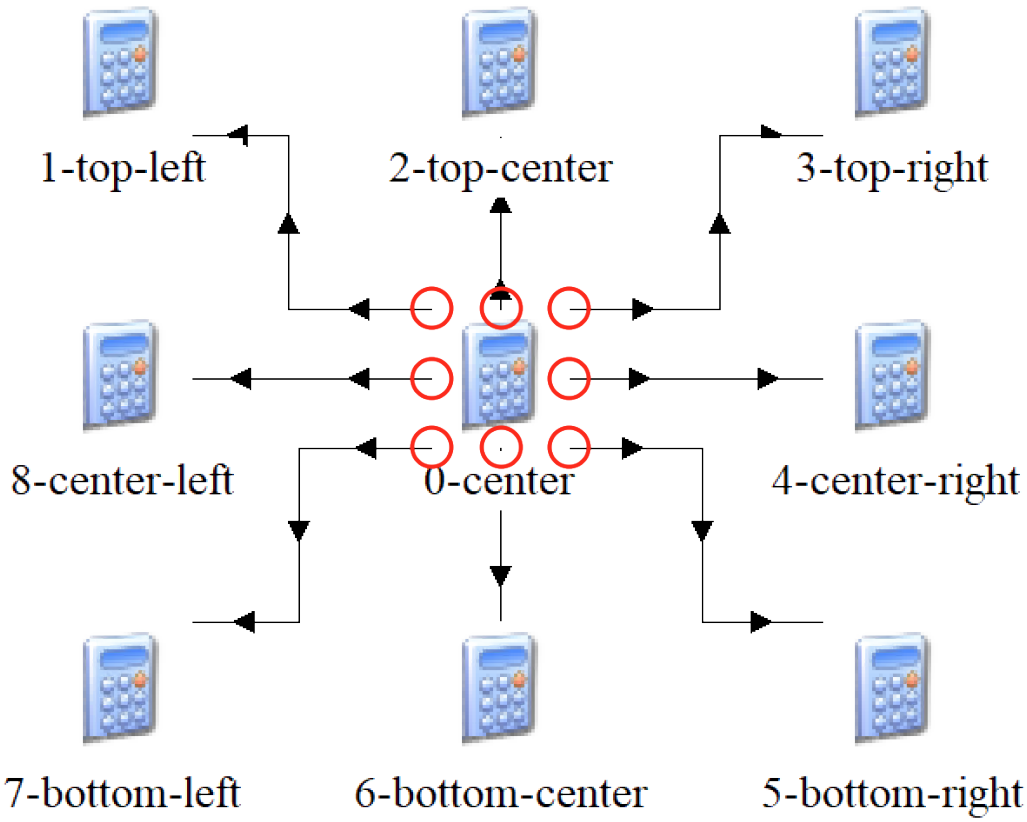
Parameters **other** –

get_octo_pts_dict(*offset=10*)
Define 8-anchor Point around the *TrnsysModel* in cartesian space and return a named-dict with human readable meaning. These points are equally dispersed at the four corners and 4 edges of the center, at distance = offset

See `set_link_style()` or `trnsymodel.LinkStyle` for more details.

Parameters **`offset`** (*float*) – The offset around the center point of `self`.

Note: In the Studio, a component has 8 anchor points at the four corners and four edges. units.Links can be created on these connections.



Statements

Statement	This is the base class for many of the TRNSYS Simulation Control and Listing Control Statements.
Version	Added with TRNSYS version 15.
NaNCheck	One problem that has plagued TRNSYS simulation debuggers is that in Fortran, the "Not a Number" (NaN) condition can be passed along through numerous subroutines without being flagged as an error.
OverwriteCheck	A common error in non standard and user written TRNSYS Type routines is to reserve too little space in the global output array.

Continued on next page

Table 2 – continued from previous page

TimeReport	The statement TIME_REPORT turns on or off the internal calculation of the time spent on each unit.
List	The LIST statement is used to turn on the TRNSYS processor listing after it has been turned off by a NOLIST statement.
Simulation	The SIMULATION statement is required for all simulations, and must be placed in the TRNSYS input file prior to the first UNIT-TYPE statement.
Tolerances	The TOLERANCES statement is an optional control statement used to specify the error tolerances to be used during a TRNSYS simulation.
Limits	The LIMITS statement is an optional control statement used to set limits on the number of iterations that will be performed by TRNSYS during a time step before it is determined that the differential equations and/or algebraic equations are not converging.
DFQ	The optional DFQ card allows the user to select one of three algorithms built into TRNSYS to numerically solve differential equations (see Manual 08-Programmer's Guide for additional information about solution of differential equations).
NoCheck	TRNSYS allows up to 20 different INPUTS to be removed from the list of INPUTS to be checked for convergence (see Section 1.9).
NoList	The NOLIST statement is used to turn off the listing of the TRNSYS input file.
Map	The MAP statement is an optional control statement that is used to obtain a component output map listing which is particularly useful in debugging component interconnections.
EqSolver	With the release of TRNSYS 16, new methods for solving blocks of EQUATIONS statements were added.
End	The END statement must be the last line of a TRNSYS input file.
Solver	A SOLVER command has been added to TRNSYS to select the computational scheme.

pyTrnsysType.statements.Statement

class pyTrnsysType.statements.**Statement**

This is the base class for many of the TRNSYS Simulation Control and Listing Control Statements. It implements common methods such as the repr() method.

pyTrnsysType.statements.Version

class pyTrnsysType.statements.**Version**(*v=(18, 0)*)

Added with TRNSYS version 15. The idea of the command is that by labeling decks with the TRNSYS version number that they were created under, it is easy to keep TRNSYS backwards compatible. The version number is saved by the TRNSYS kernel and can be acted upon.

Initialize the Version statement

Parameters **v** (*tuple*) – A tuple of (major, minor) eg. 18.0 :> (18, 0)

pyTrnsysType.statements.NaNCheck

class pyTrnsysType.statements.**NaNCheck**(*n=0*)

One problem that has plagued TRNSYS simulation debuggers is that in Fortran, the "Not a Number" (NaN) condition can be passed along through numerous subroutines without being flagged as an error. For example, a division by zero results in a variable being set to NaN. This NaN can then be used in subsequent equation, causing them to be set to NaN as well. The problem persists for a time until a Range Check or an Integer Overflow error occurs and actually stops simulation progress. To alleviate the problem, the NAN_CHECK Statement was added as an optional debugging feature in TRNSYS input files.

Initialize a NaNCheck object.

Hint: If the NAN_CHECK statement is present (n=1), then the TRNSYS kernel checks every output of each component at each iteration and generates a clean error if ever one of those outputs has been set to the FORTRAN NaN condition. Because this checking is very time consuming, users are not advised to leave NAN_CHECK set in their input files as it causes simulations to run much more slowly.

Parameters **n** (*int*) – Is 0 if the NAN_CHECK feature is not desired or 1 if NAN_CHECK feature is desired. Default is 0.

pyTrnsysType.statements.OverwriteCheck

class pyTrnsysType.statements.**OverwriteCheck**(*n=0*)

A common error in non standard and user written TRNSYS Type routines is to reserve too little space in the global output array. By default, each Type is accorded 20 spots in the global TRNSYS output array. However, there is no way to prevent the Type from then writing in (for example) the 21st spot; the entire global output array is always accessible. By activating the OVERWRITE_CHECK statement, the TRNSYS kernel checks to make sure that each Type did not write outside its allotted space. As with the NAN_CHECK statement, OVERWRITE_CHECK is a time consuming process and should only be used as a debugging tool when a simulation is ending in error.

Initialize an OVERWRITE_CHECK object.

Hint: OVERWRITE_CHECK is a time consuming process and should only be used as a debugging tool when a simulation is ending in error.

Parameters **n** (*int*) – Is 0 if the OVERWRITE_CHECK feature is not desired or 1 if OVERWRITE_CHECK feature is desired.

pyTrnsysType.statements.TimeReport

class pyTrnsysType.statements.**TimeReport**(*n=0*)

The statement TIME_REPORT turns on or off the internal calculation of the time spent on each unit. If this feature is desired, the listing file will contain this information at the end of the file.

Initialize a TIME_REPORT object.

Parameters **n** (*int*) – Is 0 if the TIME_REPORT feature is not desired or 1 if TIME_REPORT feature is desired.

pyTrnsysType.statements.List

class pyTrnsysType.statements.**List**(*activate=False*)

The LIST statement is used to turn on the TRNSYS processor listing after it has been turned off by a NOLIST statement.

Hint: The listing is assumed to be on at the beginning of a TRNSYS input file. As many LIST cards as desired may appear in a TRNSYS input file and may be located anywhere in the input file.

Parameters **activate** (*bool*) –

pyTrnsysType.statements.Simulation

class pyTrnsysType.statements.**Simulation**(*start=0*, *stop=8760*, *step=1*)

The SIMULATION statement is required for all simulations, and must be placed in the TRNSYS input file prior to the first UNIT-TYPE statement. The simulation statement determines the starting and stopping times of the simulation as well as the time step to be used.

Initialize the Simulation statement

Attention: With TRNSYS 16 and beyond, the starting time is now specified as the time at the beginning of the first time step.

Parameters

- **start** (*int*) – The hour of the year at which the simulation is to begin.
- **stop** (*int*) – The hour of the year at which the simulation is to end.
- **step** (*float*) – The time step to be used (hours).

pyTrnsysType.statements.Tolerances

class pyTrnsysType.statements.**Tolerances**(*epsilon_d=0.01*, *epsilon_a=0.01*)

The TOLERANCES statement is an optional control statement used to specify the error tolerances to be used during a TRNSYS simulation.

Parameters

- **epsilon_d** – is a relative (and -epsilon_d is an absolute) error tolerance controlling the integration error.
- **epsilon_a** – is a relative (and -epsilon_a is an absolute) error tolerance controlling the convergence of input and output variables.

pyTrnsysType.statements.Limits

class pyTrnsysType.statements.**Limits**(*m=25*, *n=10*, *p=None*)

The LIMITS statement is an optional control statement used to set limits on the number of iterations that will be performed by TRNSYS during a time step before it is determined that the differential equations and/or algebraic equations are not converging.

Parameters

- **m** (*int*) – is the maximum number of iterations which can be performed during a time-step before a WARNING message is printed out.
- **n** (*int*) – is the maximum number of WARNING messages which may be printed before the simulation terminates in ERROR.
- **p** (*int, optional*) – is an optional limit. If any component is called p times in one time step, then the component will be traced (See Section 2.3.5) for all subsequent calls in the timestep. When p is not specified by the user, TRNSYS sets p equal to m.

pyTrnsysType.statements.DFQ

class pyTrnsysType.statements.**DFQ**(*k=1*)

The optional DFQ card allows the user to select one of three algorithms built into TRNSYS to numerically solve differential equations (see Manual 08-Programmer's Guide for additional information about solution of differential equations).

Initialize the The Differential Equation Solving Method Statement

Parameters **k** (*int, optional*) – an integer between 1 and 3. If a DFQ card is not present in the TRNSYS input file, DFQ 1 is assumed.

Note: The three numerical integration algorithms are:

1. Modified-Euler method (a 2nd order Runge-Kutta method)
2. Non-self-starting Heun's method (a 2nd order Predictor-Corrector method)
3. Fourth-order Adams method (a 4th order Predictor-Corrector method)

---

pyTrnsysType.statements.NoCheck

class pyTrnsysType.statements.**NoCheck**(*inputs=None*)

TRNSYS allows up to 20 different INPUTS to be removed from the list of INPUTS to be checked for convergence (see Section 1.9).

Parameters **inputs** (*list of Input*) –

pyTrnsysType.statements.NoList

class pyTrnsysType.statements.**NoList**(*active=True*)

The NOLIST statement is used to turn off the listing of the TRNSYS input file.

Parameters **active** (*bool*) – Setting active to True will add the NOLIST statement

pyTrnsysType.statements.Map

class pyTrnsysType.statements.**Map**(*active=True*)

The MAP statement is an optional control statement that is used to obtain a component output map listing which is particularly useful in debugging component interconnections.

Setting active to True will add the MAP statement

Parameters **active** (*bool*) – Setting active to True will add the MAP statement

pyTrnsysType.statements.EqSolver

class pyTrnsysType.statements.**EqSolver**(*n=0*)

With the release of TRNSYS 16, new methods for solving blocks of EQUATIONS statements were added. For additional information on EQUATIONS statements, please refer to section 6.3.9. The order in which blocks of EQUATIONS are solved is controlled by the EQSOLVER statement.

Hint: n can have any of the following values:

1. n=0 (default if no value is provided) if a component output or TIME changes, update the block of equations that depend upon those values. Then update components that depend upon the first block of equations. Continue looping until all equations have been updated appropriately. This equation blocking method is most like the method used in TRNSYS version 15 and before.
2. n=1 if a component output or TIME changes by more than the value set in the TOLERANCES Statement (see Section 6.3.3), update the block of equations that depend upon those values. Then update components that depend upon the first block of equations. Continue looping until all equations have been updated appropriately.
3. n=2 treat equations as a component and update them only after updating all components.

Parameters **n** (*int*) – The order in which the equations are solved.

pyTrnsysType.statements.End

class pyTrnsysType.statements.**End**

The END statement must be the last line of a TRNSYS input file. It signals the TRNSYS processor that no more control statements follow and that the simulation may begin.

pyTrnsysType.statements.Solver

class pyTrnsysType.statements.**Solver**(*k=0*, *rf_min=1*, *rf_max=1*)

A SOLVER command has been added to TRNSYS to select the computational scheme. The optional SOLVER card allows the user to select one of two algorithms built into TRNSYS to numerically solve the system of algebraic and differential equations.

Parameters

- **k** (*int*) – the solution algorithm.
- **rf_min** (*float*) – the minimum relaxation factor.
- **rf_max** (*float*) – the maximum relaxation factor.

Note: k is either the integer 0 or 1. If a SOLVER card is not present in the TRNSYS input file, SOLVER 0 is assumed. If k = 0, the SOLVER statement takes two additional parameters, RFmin and RFmax:

The two solution algorithms (k) are:

- 0: Successive Substitution
- 1: Powell's Method

Constant	The CONSTANTS statement is useful when simulating a number of systems with identical component configurations but with different parameter values, initial input values, or initial values of time dependent variables.
Equation	The EQUATIONS statement allows variables to be defined as algebraic functions of constants, previously defined variables, and outputs from TRNSYS components.
ControlCards	The *ControlCards* is a container for all the TRNSYS Simulation Control Statements and Listing Control Statements.
ConstantCollection	Initialize a new ConstantCollection.
EquationCollection	A class that behaves like a list and that collects one or more Equations.

pyTrnsysType.input_file.Constant

class pyTrnsysType.input_file.**Constant**(*name=None*, *equals_to=None*, *doc=None*)

The CONSTANTS statement is useful when simulating a number of systems with identical component configurations but with different parameter values, initial input values, or initial values of time dependent variables.

Parameters

- **name** (*str*) – The left hand side of the equation.
- **equals_to** (*str, TypeVariable*) – The right hand side of the equation.
- **doc** (*str, optional*) – A small description optionally printed in the deck file.

classmethod from_expression(*expression*, *doc=None*)

Create a Constant from a string expression. Anything before the equal sign ("=") will become the Constant's name and anything after will become the equality statement.

Hint: The simple expressions are processed much as FORTRAN arithmetic statements are, with one significant exceptions. Expressions are evaluated from left to right with no precedence accorded to any operation over another. This rule must constantly be borne in mind when writing long expressions.

Parameters

- **expression** (*str*) – A user-defined expression to parse.
- **doc** (*str, optional*) – A small description optionally printed in the deck file.

property constant_number

The equation number. Unique

pyTrnsysType.input_file.Equation

class pyTrnsysType.input_file.**Equation**(*name=None*, *equals_to=None*, *doc=None*)

The EQUATIONS statement allows variables to be defined as algebraic functions of constants, previously defined variables, and outputs from TRNSYS components. These variables can then be used in place of numbers in the TRNSYS input file to represent inputs to components; numerical values of parameters; and initial values of inputs and time-dependent variables. The capabilities of the EQUATIONS statement overlap but greatly exceed those of the CONSTANTS statement described in the previous section.

Hint: In pyTrnsysType, the Equation class works hand in hand with the *EquationCollection* class. This class behaves a little bit like the equation component in the TRNSYS Studio, meaning that you can list equation in a block, give it a name, etc. See the *EquationCollection* class for more details.

Parameters

- **name** (*str*) – The left hand side of the equation.
- **equals_to** (*str, TypeVariable*) – The right hand side of the equation.
- **doc** (*str, optional*) – A small description optionally printed in the deck file.

classmethod from_expression(*expression*, *doc=None*)

Create an equation from a string expression. Anything before the equal sign ("=") will become a Constant and anything after will become the equality statement.

Example

Create a simple expression like so:

```
>>> equa1 = Equation.from_expression("TdbAmb = [011,001]")
```

Parameters

- **expression** (*str*) – A user-defined expression to parse.
- **doc** (*str, optional*) – A small description optionally printed in the deck file.

classmethod from_symbolic_expression(*name*, *func*, **args*, *doc=None*)

Crate an equation from a string with a catch. The underlying engine will use Sympy and symbolic variables.

Examples

In this example, we define a variable (var_a) and we want it to be equal to the 'Outlet Air Humidity Ratio' divided by 12 + log(Temperature to heat source). In a TRNSYS deck file one would have to manually determine the unit numbers and output numbers and write something like : '[1, 2]/12 + log([1, 1])'. With the *from_symbolic_expression()*, we can do this very simply:

1. first, define the name of the variable:

```
>>> name = "var_a"
```

2. then, define the expression as a string. Here, the variables *a* and *b* are symbols that represent the two type outputs. Note that their name has bee chosen arbitrarily.

```
>>> exp = "log(a) + b / 12"
>>> # would be also equivalent to
>>> exp = "log(x) + y / 12"
```

3. here, we define the actual variables (the type outputs) after loading our model from its proforma:

```
>>> from pyTrnsysType import TrnsysModel
>>> fan = TrnsysModel.from_xml("fan_type.xml")
>>> vars = (fan.outputs[0], fan.outputs[1])
```

Important: The order of the symbolic variable encountered in the string expression (step 2), from left to right, must be the same for the tuple of variables. For instance, *a* is followed by *b*, therefore *fan.outputs[0]* is followed by *fan.outputs[1]*.

4. finally, we create the Equation. Note that vars is passed with the '*' declaration to unpack the tuple.

```
>>> from pyTrnsysType.input_file import Equation
>>> eq = Equation.from_symbolic_expression(name, exp, *vars)
>>> print(eq)
[1, 1]/12 + log([1, 2])
```

Parameters

- **name** (*str*) – The name of the variable (left-hand side), of the equation.
- **func** (*str*) – The expression to evaluate. Use any variable name and mathematical expression.
- ***args** (*tuple*) – A tuple of `TypeVariable` that will replace the any variable name specified in the above expression.
- **doc** (*str, optional*) – A small description optionally printed in the deck file.

Returns The Equation Statement object.

Return type *Equation*

property eq_number
The equation number. Unique

pyTrnsysType.input_file.ControlCards

class pyTrnsysType.input_file.**ControlCards**(*version*, *simulation*, *tolerances=None*, *limits=None*, *nancheck=None*, *overwritecheck=None*, *timereport=None*, *constants=None*, *equations=None*, *dfq=None*, *nocheck=None*, *eqsolver=None*, *solver=None*, *nolist=None*, *list=None*, *map=None*)

The *ControlCards* is a container for all the TRNSYS Simulation Control Statements and Listing Control Statements. It implements the `_to_deck()` method which pretty-prints the statements with their docstrings.

Each simulation must have SIMULATION and END statements. The other simulation control statements are optional. Default values are assumed for TOLERANCES, LIMITS, SOLVER, EQSOLVER and DFQ if they are not present

Parameters

- **version** (Version) – The VERSION Statement. labels the deck with the TRNSYS version number. See Version for more details.
- **simulation** (Simulation) – The SIMULATION Statement.determines the starting and stopping times of the simulation as well as the time step to be used. See Simulation for more details.
- **tolerances** (Tolerances, *optional*) – Convergence Tolerances (TOLERANCES). Specifies the error tolerances to be used during a TRNSYS simulation. See Tolerances for more details.
- **limits** (Limits, *optional*) – The LIMITS Statement. Sets limits on the number of iterations that will be performed by TRNSYS during a time step before it is determined that the differential equations and/or algebraic equations are not converging. See Limits for more details.
- **nancheck** (NaNCheck, *optional*) – The NAN_CHECK Statement. An optional debugging feature in TRNSYS. If the NAN_CHECK statement is present, then the TRNSYS kernel checks every output of each component at each iteration and generates a clean error if ever one of those outputs has been set to the FORTRAN NaN condition. See NaNCheck for more details.
- **overwritecheck** (OverwriteCheck, *optional*) – The OVERWRITE_CHECK Statement. An optional debugging feature in TRNSYS. Checks to make sure that each Type did not write outside its allotted space. See OverwriteCheck for more details.
- **timereport** (TimeReport, *optional*) – The TIME_REPORT Statement. Turns on or off the internal calculation of the time spent on each unit. See TimeReport for more details.
- **constants** (*Constants*, *optional*) – The CONSTANTS Statement. See Constants for more details.
- **equations** (*Equations*, *optional*) – The EQUATIONS Statement. See Equations for more details.
- **dfq** (DFQ, *optional*) – Allows the user to select one of three algorithms built into TRNSYS to numerically solve differential equations. See DFQ for more details.
- **nocheck** (NoCheck, *optional*) – The Convergence Check Suppression Statement. Remove up to 20 inputs for the convergence check. See NoCheck for more details.
- **eqsolver** (EqSolver, *optional*) – The Equation Solving Method Statement. The order in which blocks of EQUATIONS are solved is controlled by the EQSOLVER statement. See EqSolver for more details.
- **solver** (Solver, *optional*) – The SOLVER Statement. Select the computational scheme. See Solver for more details.
- **nolist** (NoList, *optional*) – The NOLIST Statement. See NoList for more details.
- **list** (List, *optional*) – The LIST Statement. See List for more details.
- **map** (Map, *optional*) – The MAP Statement. See Map for more details.

Note: Some Statements have not been implemented because only TRNSYS gods use them. Here is a list of Statements that have been ignored:

- The Convergence Promotion Statement (ACCELERATE)
- The Calling Order Specification Statement (LOOP)

classmethod all()
Returns a SimulationCard with all available Statements initialized with their default values. This class method is not recommended since many of the Statements are a time consuming process and should be used as a debugging tool.

classmethod debug_template()
Returns a SimulationCard with useful debugging Statements.

classmethod basic_template()
Returns a SimulationCard with only the required Statements

pyTrnsysType.input_file.ConstantCollection

class pyTrnsysType.input_file.**ConstantCollection**(*initlist=None*, *name=None*)
Initialize a new ConstantCollection.

Example

```
>>> c_1 = Constant.from_expression("A = 1")
>>> c_2 = Constant.from_expression("B = 2")
>>> ConstantCollection([c_1, c_2])
```

Parameters

- **initlist** (*Iterable, optional*) – An iterable.
- **name** (*str*) – A user defined name for this collection of constants. This name will be used to identify this block of constants in the .dck file;

append(*item*)
S.append(value) – append value to the end of the sequence

clear() → None – remove all items from S

count(*value*) → integer – return number of occurrences of value

extend(*other*)
S.extend(iterable) – extend sequence by appending elements from the iterable

index(*value*[, *start*[, *stop*]]) → integer – return first index of value.
Raises ValueError if the value is not present.

Supporting start and stop arguments is optional, but recommended.

insert(*i*, *item*)
S.insert(index, value) – insert value before index

pop([*index*]) → item – remove and return item at index (default last).
Raise IndexError if list is empty or index is out of range.

remove(*item*)
S.remove(value) – remove first occurrence of value. Raise ValueError if the value is not present.

reverse()
S.reverse() – reverse *IN PLACE*

pyTrnsysType.input_file.EquationCollection

class pyTrnsysType.input_file.**EquationCollection**(*initlist=None*, *name=None*)

A class that behaves like a list and that collects one or more `Equations`. This class behaves a little bit like the equation component in the TRNSYS Studio, meaning that you can list equation in a block, give it a name, etc.

Hint: Creating equations in PyTrnsysType is done trough the *`Equation`* class. Equations are than collected in this EquationCollection. See the *`Equation`* class for more details.

Initialize a new EquationCollection.

Example

```
>>> equa1 = Equation.from_expression("TdbAmb = [011,001]")
>>> equa2 = Equation.from_expression("rhAmb = [011,007]")
>>> EquationCollection([equa1, equa2])
```

Parameters

- **initlist** (*Iterable, optional*) – An iterable.
- **name** (*str*) – A user defined name for this collection of equations. This name will be used to identify this block of equations in the .dck file;

append(*item*)
S.append(value) – append value to the end of the sequence

clear() → None – remove all items from S

count(*value*) → integer – return number of occurrences of value

extend(*other*)
S.extend(iterable) – extend sequence by appending elements from the iterable

index(*value*[, *start*[, *stop*]]) → integer – return first index of value.
Raises ValueError if the value is not present.

Supporting start and stop arguments is optional, but recommended.

insert(*i*, *item*)
S.insert(index, value) – insert value before index

pop([*index*]) → item – remove and return item at index (default last).
Raise IndexError if list is empty or index is out of range.

remove(*item*)
S.remove(value) – remove first occurrence of value. Raise ValueError if the value is not present.

reverse()
S.reverse() – reverse *IN PLACE*

Utils

affine_transform	Apply affine transformation to geometry.

Continued on next page

Table 4 – continued from previous page

get_int_from_rgb	Simple utility to convert an RBG color to its TRNSYS Studio compatible int color.
get_rgb_from_int	Simple utility to convert an rgb int color to its red, green and blue colors.
DeckFilePrinter	Print derivative of a function of symbols in deck file form.
print_my_latex	Most of the printers define their own wrappers for print().
TypeVariableSymbol	This is a subclass of the sympy Symbol class.

pyTrnsysType.utils.affine_transform

pyTrnsysType.utils.**affine_transform**(*geom*, *matrix=None*)

Apply affine transformation to geometry. By, default, flip geometry along the x axis.

Hint: visit affine_matrix for other affine transformation matrices.

#/media/ File:2D_affine_transformation_matrix.svg

Parameters

- **geom** (*BaseGeometry*) – The geometry.
- **matrix** (*np.array*) – The coefficient matrix is provided as a list or tuple.

pyTrnsysType.utils.get_int_from_rgb

pyTrnsysType.utils.**get_int_from_rgb**(*rgb*)

Simple utility to convert an RBG color to its TRNSYS Studio compatible int color. Values are used ranging from 0 to 255 for each of the components.

Important: Unlike Java, the TRNSYS Studio will want an integer where bits 0-7 are the blue value, 8-15 the green, and 16-23 the red.

Examples

Get the rgb int from an rgb 3-tuple

```
>>> get_int_from_rgb((211, 122, 145))
9534163
```

Parameters **rgb** (*tuple*) – The red, green and blue values. All values assumed to be in range [0, 255].

Returns the rgb int.

Return type (int)

pyTrnsysType.utils.get_rgb_from_int

pyTrnsysType.utils.**get_rgb_from_int**(*rgb_int*)

Simple utility to convert an rgb int color to its red, green and blue colors. Values are used ranging from 0 to 255 for each of the components.

Important: Unlike Java, the TRNSYS Studio will want an integer where bits 0-7 are the blue value, 8-15 the green, and 16-23 the red.

Examples

Get the rgb tuple from a an rgb int.

```
>>> get_rgb_from_int(9534163)
(211, 122, 145)
```

Parameters **rgb_int** (*int*) – An rgb int representation.

Returns (r, g, b) tuple.

Return type (tuple)

pyTrnsysType.utils.DeckFilePrinter

class pyTrnsysType.utils.**DeckFilePrinter**(*settings=None*)

Print derivative of a function of symbols in deck file form. This will override the sympy.printing.str.StrPrinter#_print_Symbol() method to print the TypeVariable's unit_number and output number.

doprint(*expr*)

Returns printer's representation for expr (as a string)

emptyPrinter(*expr*)

str(object='') -> str str(bytes_or_buffer[, encoding[, errors]]) -> str

Create a new string object from the given object. If encoding or errors is specified, then the object must expose a data buffer that will be decoded using the given encoding and error handler. Otherwise, returns the result of object.__str__() (if defined) or repr(object). encoding defaults to sys.getdefaultencoding(). errors defaults to 'strict'.

classmethod set_global_settings(***settings*)

Set system-wide printing settings.

pyTrnsysType.utils.print_my_latex

pyTrnsysType.utils.**print_my_latex**(*expr*)

Most of the printers define their own wrappers for print(). These wrappers usually take printer settings. Our printer does not have any settings.

Parameters **expr** –

pyTrnsysType.utils.TypeVariableSymbol

class pyTrnsysType.utils.**TypeVariableSymbol**

This is a subclass of the sympy Symbol class. It is a bit of a hack, so hopefully nothing bad will happen.

TypeVariableSymbol are identified by TypeVariable and assumptions:

```
>>> from pyTrnsysType import TypeVariableSymbol
>>> TypeVariableSymbol("x") == TypeVariableSymbol("x")
True
>>> TypeVariableSymbol("x", real=True) == TypeVariableSymbol("x",
real=False)
False
```

Parameters

- **type_variable** (TypeVariable) – The TypeVariable to defined as a Symbol.
- ****assumptions** – See sympy.core.assumptions for more details.

apart (*x=None*, ***args*)

See the apart function in sympy.polys

property 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).

args_cnc (*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 maintained. 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 suppressed 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_coeff_Add(*rational=False*)
Efficiently extract the coefficient of a summation.

as_coeff_Mul(*rational=False*)
Efficiently extract the coefficient of a product.

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

as_coeff_mul(**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(*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
```

(continues on next page)

(continued from previous page)

```
>>> 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 coefficient of a monomial in Poly

sympy.polys.polytools.nth() like coeff_monomial but powers of monomial terms are used

as_coefficients_dict()

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(*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()) == foo`. The primitive need not 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_dummy()

Return the expression with any objects having structurally bound symbols replaced with unique, canonical symbols within the object in which they appear and having only the default assumption for commutativity being True.

Examples

```
>>> from sympy import Integral, Symbol
>>> from sympy.abc import x, y
>>> r = Symbol('r', real=True)
```

(continues on next page)

(continued from previous page)

```
>>> Integral(r, (r, x)).as_dummy()
Integral(_0, (_0, x))
>>> _.variables[0].is_real is None
True
```

Notes

Any object that has structural dummy variables should have a property, *bound_symbols* that returns a list of structural dummy symbols of the object itself.

Lambda and Subs have bound symbols, but because of how they are cached, they already compare the same regardless of their bound symbols:

```
>>> from sympy import Lambda
>>> Lambda(x, x + 1) == Lambda(y, y + 1)
True
```

as_expr(**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(**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 variables 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 either have terms that contain variables that are in deps, or be equal to 0 (when self is an Add) or 1 (when self is a Mul)
- 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)
```

(continues on next page)

(continued from previous page)

```
>>> (-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()`*, `Add.as_two_terms()`, `Mul.as_two_terms()`, *`as_coeff_add()`*, *`as_coeff_mul()`*

`as_leading_term`(**symbols*)

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()
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(*order=None*)
Return list of ordered factors (if Mul) else [self].

as_ordered_terms(*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_poly(**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
```

as_powers_dict()
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 in the dictionary.

See also:

as_ordered_factors() An alternative for noncommutative applications, returning an ordered list of factors.

args_cnc() Similar to as_ordered_factors, but guarantees separation of commutative and noncommutative factors.

as_real_imag(*deep=True*, ***hints*)

Performs complex expansion on 'self' and returns a tuple containing collected both real and imaginary parts. This method can't be confused with re() and im() functions, which does not perform complex expansion at evaluation.

However it is possible to expand both re() and im() functions and get exactly the same results as with a single call to this function.

```
>>> from sympy import symbols, I
```

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

```
>>> (x + y*I).as_real_imag()
(x, y)
```

```
>>> from sympy.abc import z, w
```

```
>>> (z + w*I).as_real_imag()
(re(z) - im(w), re(w) + im(z))
```

as_set()

Rewrites Boolean expression in terms of real sets.

Examples

```
>>> from sympy import Symbol, Eq, Or, And
>>> x = Symbol('x', real=True)
>>> Eq(x, 0).as_set()
{0}
>>> (x > 0).as_set()
Interval.open(0, oo)
>>> And(-2 < x, x < 2).as_set()
Interval.open(-2, 2)
>>> Or(x < -2, 2 < x).as_set()
Union(Interval.open(-oo, -2), Interval.open(2, oo))
```

as_terms()

Transform an expression to a list of terms.

property assumptions0

Return object *type* assumptions.

For example:

Symbol('x', real=True) Symbol('x', integer=True)

are different objects. In other words, besides Python type (Symbol in this case), the initial assumptions are also forming their typeinfo.

Examples

```
>>> from sympy import Symbol
>>> from sympy.abc import x
>>> x.assumptions0
{'commutative': True}
>>> x = Symbol("x", positive=True)
>>> x.assumptions0
{'commutative': True, 'complex': True, 'hermitian': True,
'imaginary': False, 'negative': False, 'nonnegative': True,
'nonpositive': False, 'nonzero': True, 'positive': True, 'real': True,
'zero': False}
```

atoms (**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.

```
>>> 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 special 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)}
```

property binary_symbols

Return from the atoms of self those which are free symbols.

For most expressions, all symbols are free symbols. For some classes this is not true. e.g. Integrals use Symbols for the dummy variables which are bound variables, so Integral has a method to return all symbols except those. Derivative keeps track of symbols with respect to which it will perform a derivative; those are bound variables, too, so it has its own free_symbols method.

Any other method that uses bound variables should implement a free_symbols method.

cancel(**gens*, ***args*)

See the cancel function in sympy.polys

property canonical_variables

Return a dictionary mapping any variable defined in `self.bound_symbols` to Symbols that do not clash with any existing symbol in the expression.

Examples

```
>>> from sympy import Lambda
>>> from sympy.abc import x
>>> Lambda(x, 2*x).canonical_variables
{x: _0}
```

classmethod class_key()

Nice order of classes.

coeff(*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) or 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 coefficient 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(*syms*, *func=None*, *evaluate=True*, *exact=False*, *distribute_order_term=True*)
See the collect function in sympy.simplify

combsimp()
See the combsimp function in sympy.simplify

compare(*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
```

compute_leading_term(*x*, *logx=None*)
as_leading_term is only allowed for results of .series() This is a wrapper to compute a series first.

could_extract_minus_sign()
Return True if self is not in a canonical form with respect to its sign.

For most expressions, e, there will be a difference in e and -e. When there is, True will be returned for one and False for the other; False will be returned if there is no difference.

Examples

```
>>> from sympy.abc import x, y
>>> e = x - y
>>> {i.could_extract_minus_sign() for i in (e, -e)}
{False, True}
```

count(*query*)
Count the number of matching subexpressions.

count_ops(*visual=None*)
wrapper for count_ops that returns the operation count.

doit(***hints*)
Evaluate objects that are not evaluated by default like limits, integrals, sums and products. All objects of this kind will be evaluated recursively, unless some species were excluded via 'hints' or unless the 'deep' hint was set to 'False'.

```
>>> from sympy import Integral
>>> from sympy.abc import x
```

```
>>> 2*Integral(x, x)
2*Integral(x, x)
```

```
>>> (2*Integral(x, x)).doit()
x**2
```

```
>>> (2*Integral(x, x)).doit(deep=False)
2*Integral(x, x)
```

dummy_eq(*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
```

equals(*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.

evalf(*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)

Notes

When Floats are naively substituted into an expression, precision errors may adversely affect the result. For example, adding 1e16 (a Float) to 1 will truncate to 1e16; if 1e16 is then subtracted, the result will be 0. That is exactly what happens in the following:

```
>>> from sympy.abc import x, y, z
>>> values = {x: 1e16, y: 1, z: 1e16}
>>> (x + y - z).subs(values)
0
```

Using the subs argument for evalf is the accurate way to evaluate such an expression:

```
>>> (x + y - z).evalf(subs=values)
1.00000000000000
```

expand(*deep=True*, *modulus=None*, *power_base=True*, *power_exp=True*, *mul=True*, *log=True*, *multinomial=True*, *basic=True*, ***hints*)
Expand an expression using hints.

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

property expr_free_symbols
Like `free_symbols`, but returns the free symbols only if they are contained in an expression node.

Examples

```
>>> from sympy.abc import x, y
>>> (x + y).expr_free_symbols
{x, y}
```

If the expression is contained in a non-expression object, don't return the free symbols. Compare:

```
>>> from sympy import Tuple
>>> t = Tuple(x + y)
>>> t.expr_free_symbols
set()
>>> t.free_symbols
{x, y}
```

extract_additively(*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(*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(*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.

Examples

```
>>> 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(**gens*, ***args*)

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

find(*query*, *group=False*)
Find all subexpressions matching a query.

fourier_series(*limits=None*)
Compute fourier sine/cosine series of self.

See the docstring of the *fourier_series()* in sympy.series.fourier for more information.

fps(*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 *fps()* function in sympy.series.formal for more information.

property free_symbols
Return from the atoms of self those which are free symbols.

For most expressions, all symbols are free symbols. For some classes this is not true. e.g. Integrals use Symbols for the dummy variables which are bound variables, so Integral has a method to return all symbols except those. Derivative keeps track of symbols with respect to which it will perform a derivative; those are bound variables, too, so it has its own free_symbols method.

Any other method that uses bound variables should implement a free_symbols method.

classmethod fromiter(*args*, ***assumptions*)
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)
```

property 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
```

gammasimp()
See the gammasimp function in sympy.simplify

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

getn()
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()
```

has(**patterns*)
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
```

integrate(**args*, ***kwargs*)
See the integrate function in sympy.integrals

invert(*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(**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 the 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 may 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

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

is_constant(**wrt*, ***flags*)
Return True if self is constant, False if not, or None if the constancy could not be determined conclusively.

If an expression has no free symbols then it is a constant. If there are free symbols it is possible that the expression is a constant, perhaps (but not necessarily) zero. To test such expressions, two strategies are tried:

1) numerical evaluation at two random points. If two such evaluations give two different values and the values have a precision greater than 1 then self is not constant. If the evaluations agree or could not be obtained with any precision, no decision is made. The numerical testing is done only if `wrt` is different than the free symbols.

2) differentiation with respect to variables in 'wrt' (or all free symbols if omitted) to see if the expression is constant or not. This will not always lead to an expression that is zero even though an expression is constant (see added test in test_expr.py). If all derivatives are zero then self is constant with respect to the given symbols.

If neither evaluation nor differentiation can prove the expression is constant, None is returned unless two numerical values happened to be the same and the flag `failing_number` is True – in that case the numerical value will be returned.

If flag simplify=False is passed, self will not be simplified; the default is True since self should be simplified before testing.

Examples

```
>>> from sympy import cos, sin, Sum, S, pi
>>> from sympy.abc import a, n, x, y
>>> x.is_constant()
False
>>> S(2).is_constant()
True
>>> Sum(x, (x, 1, 10)).is_constant()
True
>>> Sum(x, (x, 1, n)).is_constant()
False
>>> Sum(x, (x, 1, n)).is_constant(y)
True
>>> Sum(x, (x, 1, n)).is_constant(n)
False
>>> Sum(x, (x, 1, n)).is_constant(x)
True
>>> eq = a*cos(x)**2 + a*sin(x)**2 - a
>>> eq.is_constant()
True
>>> eq.subs({x: pi, a: 2}) == eq.subs({x: pi, a: 3}) == 0
True
```

```
>>> (0**x).is_constant()
False
>>> x.is_constant()
False
>>> (x**x).is_constant()
False
>>> one = cos(x)**2 + sin(x)**2
>>> one.is_constant()
True
>>> ((one - 1)**(x + 1)).is_constant() in (True, False) # could be 0 or 1
True
```

is_polynomial(**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 to expressions for which this returns True, and Poly(expr, *syms) should 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 may 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(**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 may result in an expression that does not appear to be a rational function 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(*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(*x*, *xlim*, *dir='+'*)

Compute limit x->xlim.

lseries(*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 following, 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 for example), but you don't know how many you should ask for in nseries() using the "n" parameter.

See also nseries().

match (*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}
```

matches (*expr*, *repl_dict={}*, *old=False*)
Helper method for match() that looks for a match between Wild symbols in self and expressions in expr.

Examples

```
>>> from sympy import symbols, Wild, Basic
>>> a, b, c = symbols('a b c')
>>> x = Wild('x')
>>> Basic(a + x, x).matches(Basic(a + b, c)) is None
True
>>> Basic(a + x, x).matches(Basic(a + b + c, b + c))
{x_: b + c}
```

n (*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)

Notes

When Floats are naively substituted into an expression, precision errors may adversely affect the result. For example, adding 1e16 (a Float) to 1 will truncate to 1e16; if 1e16 is then subtracted, the result will be 0. That is exactly what happens in the following:

```
>>> from sympy.abc import x, y, z
>>> values = {x: 1e16, y: 1, z: 1e16}
>>> (x + y - z).subs(values)
0
```

Using the subs argument for evalf is the accurate way to evaluate such an expression:

```
>>> (x + y - z).evalf(subs=values)
1.00000000000000
```

nseries (*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 is 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(*constants=[]*, *tolerance=None*, *full=False*)
See the nsimplify function in sympy.simplify

powsimp(**args*, ***kwargs*)
See the powsimp function in sympy.simplify

primitive()
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 positive 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(***kwargs*)
See the radsimp function in sympy.simplify

ratsimp()
See the ratsimp function in sympy.simplify

rcall(**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
```

refine(*assumption=True*)
See the refine function in sympy.assumptions

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

replace(*query*, *value*, *map=False*, *simultaneous=True*, *exact=None*)
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, b = map(Wild, 'ab')
>>> 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
```

Matching is exact by default when more than one Wild symbol is used: matching fails unless the match gives non-zero values for all Wild symbols:

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

When set to False, the results may be non-intuitive:

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

round(*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()
```

(continues on next page)

(continued from previous page)

```
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 (*deep=False*, *force=False*)
See the separate function in sympy.simplify

series (*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(*ratio=1.7*, *measure=None*, *rational=False*, *inverse=False*)
See the simplify function in sympy.simplify

sort_key(*order=None*)
Return a sort key.

Examples

```
>>> from sympy.core import S, I
```

```
>>> sorted([S(1)/2, I, -I], key=lambda x: x.sort_key())
[1/2, -I, I]
```

```
>>> S("[x, 1/x, 1/x**2, x**2, x**(1/2), x**(1/4), x**(3/2)]")
[x, 1/x, x**(-2), x**2, sqrt(x), x**(1/4), x**(3/2)]
>>> sorted(_, key=lambda x: x.sort_key())
[x**(-2), 1/x, x**(1/4), sqrt(x), x, x**(3/2), x**2]
```

subs(**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 pairs.** 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 container (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)
```

(continues on next page)

(continued from previous page)

```
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)
0.33333333333333333333
```

rather than

```
>>> (1/x).subs({x: 3.0}).evalf(21)
0.33333333333333314830
```

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

taylor_term(*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(**args*, ***kwargs*)
See the together function in sympy.polys

trigsimp(***args*)
See the trigsimp function in sympy.simplify

xreplace(*rule*, *hack2=False*)
Replace occurrences of objects within the expression.

Parameters **rule** (*dict-like*) – Expresses a replacement rule

Returns xreplace

Return type the result of the replacement

Examples

```
>>> from sympy import symbols, pi, exp
>>> x, y, z = symbols('x y z')
>>> (1 + x*y).xreplace({x: pi})
pi*y + 1
>>> (1 + x*y).xreplace({x: pi, y: 2})
1 + 2*pi
```

Replacements occur only if an entire node in the expression tree is matched:

```
>>> (x*y + z).xreplace({x*y: pi})
z + pi
>>> (x*y*z).xreplace({x*y: pi})
x*y*z
>>> (2*x).xreplace({2*x: y, x: z})
y
>>> (2*2*x).xreplace({2*x: y, x: z})
```

(continues on next page)

(continued from previous page)

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

xreplace doesn't differentiate between free and bound symbols. In the following, subs(x, y) would not change x since it is a bound symbol, but xreplace does:

```
>>> from sympy import Integral
>>> Integral(x, (x, 1, 2*x)).xreplace({x: y})
Integral(y, (y, 1, 2*y))
```

Trying to replace x with an expression raises an error:

```
>>> Integral(x, (x, 1, 2*x)).xreplace({x: 2*y}) # doctest: +SKIP
ValueError: Invalid limits given: ((2*y, 1, 4*y),)
```

See also:

replace() replacement capable of doing wildcard-like matching, parsing of match, and conditional replacements

subs() substitution of subexpressions as defined by the objects themselves.

CHAPTER

TWO

INDICES AND TABLES

- genindex
- search

INDEX

A

B

C

D

G

H

I

K

L

M

N

O

P

R

S

T

U

V

X