modelparameters Documentation

Release 1.0

Johan Hake, Henrik Finsberg

May 01, 2019

Programmers reference

Modelparameters is a lightweight library to declare nested parameters in Python, and can be used as a module for providing parameter structure for physical models. It is mainly developed as a tool for gotran, but certainly not limited to that.

```
from modelparameters import ScalarParam
# Define the gravitational acceleration
g = ScalarParam(9.81, name="g", unit="m/s**2",
                description="gravitational acceleration")
# Define the time (in milliseconds)
t = ScalarParam(100.0, unit="ms",
                description="Time spent")
# Compute the distance travelled in free fall
s = 0.5 * g * t**2
print(s)
```

Output:

```
49050.0  [-∞, ∞]
```

CHAPTER 1

Demos

Will come later!

CHAPTER 2

Installation

You can install modelparameters either with pip:

```
pip install modelparameters
```

or if you want the latest features, you can install from source:

```
pip install git+https://finsberg@bitbucket.org/finsberg/modelparameters.git
```

CHAPTER 3

Source code

Modelparmeters is orginally developed by Johan Hake, and the original source code can be found in his repoistory. The current maintained version can be foud here.

CHAPTER 4

Contributors

The main contributors are

- Henrik Finsberg (henriknf@simula.no)
- Johan Hake (hake.dev@gmail.com)

CHAPTER 5

License

GNU GPLv3

5.1 modelparameters package

5.1.1 Submodules

5.1.2 modelparameters.codegeneration module

modelparameters.codegeneration.**latex_unit**(*unit*)
Return sympified and LaTeX-formatted string describing given unit. E.g.: >>> LatexCodeGenerator.format_unit("m/s**2") 'mathrm{frac{m}{s^{2}}}'

modelparameters.codegeneration.**ccode**(*expr*, *assign_to=None*, *float_precision='double'*)
Return a C-code representation of a sympy expression

modelparameters.codegeneration.**cppcode**(*expr*, *assign_to=None*, *float_precision='double'*)
Return a C++-code representation of a sympy expression

modelparameters.codegeneration.**pythoncode**(*expr*, *assign_to=None*, *namespace='math'*)
Return a Python-code representation of a sympy expression

modelparameters.codegeneration.**sympycode**(*expr*, *assign_to=None*)

modelparameters.codegeneration.**matlabcode**(*expr*, *assign_to=None*)

modelparameters.codegeneration.**juliacode**(*expr*, *assign_to=None*)

modelparameters.codegeneration.**latex**(*expr*, ***settings*)
Convert the given expression to LaTeX representation.

```
>>> from sympy import latex, pi, sin, asin, Integral, Matrix, Rational
>>> from sympy.abc import x, y, mu, r, tau
```

```
>>> print(latex((2*tau)**Rational(7,2)))
8 \sqrt{2} \tau^{\frac{7}{2}}
```

Not using a print statement for printing, results in double backslashes for latex commands since that's the way Python escapes backslashes in strings.

```
>>> latex((2*tau)**Rational(7,2))
'8 \\sqrt{2} \\tau^{\\frac{7}{2}}'
```

order: Any of the supported monomial orderings (currently "lex", "grlex", or "grevlex"), "old", and "none". This parameter does nothing for Mul objects. Setting order to "old" uses the compatibility ordering for Add defined in Printer. For very large expressions, set the 'order' keyword to 'none' if speed is a concern.

mode: Specifies how the generated code will be delimited. 'mode' can be one of 'plain', 'inline', 'equation' or 'equation*'. If 'mode' is set to 'plain', then the resulting code will not be delimited at all (this is the default). If 'mode' is set to 'inline' then inline LaTeX $ $ will be used. If 'mode' is set to 'equation' or 'equation*', the resulting code will be enclosed in the 'equation' or 'equation*' environment (remember to import 'amsmath' for 'equation*'), unless the 'itex' option is set. In the latter case, the `$$ $$` syntax is used.

```
>>> print(latex((2*mu)**Rational(7,2), mode='plain'))
8 \sqrt{2} \mu^{\frac{7}{2}}
```

```
>>> print(latex((2*tau)**Rational(7,2), mode='inline'))
$8 \sqrt{2} \tau^{7 / 2}$
```

```
>>> print(latex((2*mu)**Rational(7,2), mode='equation*'))
\begin{equation*}8 \sqrt{2} \mu^{\frac{7}{2}}\end{equation*}
```

```
>>> print(latex((2*mu)**Rational(7,2), mode='equation'))
\begin{equation}8 \sqrt{2} \mu^{\frac{7}{2}}\end{equation}
```

itex: Specifies if itex-specific syntax is used, including emitting `$$ $$`.

```
>>> print(latex((2*mu)**Rational(7,2), mode='equation', itex=True))
$$8 \sqrt{2} \mu^{\frac{7}{2}}$$
```

fold_frac_powers: Emit "^{p/q}" instead of "^{frac{p}{q}}" for fractional powers.

```
>>> print(latex((2*tau)**Rational(7,2), fold_frac_powers=True))
8 \sqrt{2} \tau^{7/2}
```

fold_func_brackets: Fold function brackets where applicable.

```
>>> print(latex((2*tau)**sin(Rational(7,2))))
\left(2 \tau\right)^{\sin{\left (\frac{7}{2} \right )}}
>>> print(latex((2*tau)**sin(Rational(7,2)), fold_func_brackets = True))
\left(2 \tau\right)^{\sin {\frac{7}{2}}}
```

fold_short_frac: Emit "p / q" instead of "frac{p}{q}" when the denominator is simple enough (at most two terms and no powers). The default value is *True* for inline mode, False otherwise.

```
>>> print(latex(3*x**2/y))
\frac{3 x^{2}}{y}
>>> print(latex(3*x**2/y, fold_short_frac=True))
3 x^{2} / y
```

long_frac_ratio: The allowed ratio of the width of the numerator to the width of the denominator before we start breaking off long fractions. The default value is 2.

```
>>> print(latex(Integral(r, r)/2/pi, long_frac_ratio=2))
\frac{\int r\, dr}{2 \pi}
>>> print(latex(Integral(r, r)/2/pi, long_frac_ratio=0))
\frac{1}{2 \pi} \int r\, dr
```

mul_symbol: The symbol to use for multiplication. Can be one of None, "ldot", "dot", or "times".

```
>>> print(latex((2*tau)**sin(Rational(7,2)), mul_symbol="times"))
\left(2 \times \tau\right)^{\sin{\left (\frac{7}{2} \right )}}
```

inv_trig_style: How inverse trig functions should be displayed. Can be one of "abbreviated", "full", or "power". Defaults to "abbreviated".

```
>>> print(latex(asin(Rational(7,2))))
\operatorname{asin}{\left (\frac{7}{2} \right )}
>>> print(latex(asin(Rational(7,2)), inv_trig_style="full"))
\arcsin{\left (\frac{7}{2} \right )}
>>> print(latex(asin(Rational(7,2)), inv_trig_style="power"))
\sin^{-1}{\left (\frac{7}{2} \right )}
```

mat_str: Which matrix environment string to emit. "smallmatrix", "matrix", "array", etc. Defaults to "smallmatrix" for inline mode, "matrix" for matrices of no more than 10 columns, and "array" otherwise.

```
>>> print(latex(Matrix(2, 1, [x, y])))
\left[\begin{matrix}x\\y\end{matrix}\right]
```

```
>>> print(latex(Matrix(2, 1, [x, y]), mat_str = "array"))
\left[\begin{array}{c}x\\y\end{array}\right]
```

mat_delim: The delimiter to wrap around matrices. Can be one of "[", "(", or the empty string. Defaults to "[".

```
>>> print(latex(Matrix(2, 1, [x, y]), mat_delim="("))
\left(\begin{matrix}x\\y\end{matrix}\right)
```

symbol_names: Dictionary of symbols and the custom strings they should be emitted as.

```
>>> print(latex(x**2, symbol_names={x:'x_i'}))
x_i^{2}
```

`latex` also supports the builtin container types list, tuple, and dictionary.

```
>>> print(latex([2/x, y], mode='inline'))
$\left [ 2 / x, \quad y\right ]$
```

`modelparameters.codegeneration.`**`octavecode`**(*expr*, *assign_to=None*)

5.1.3 modelparameters.commands module

`modelparameters.commands.`**`get_output`**(*cmd*, *inp=None*, *cwd=None*, *env=None*)

`modelparameters.commands.`**`get_status_output`**(*cmd*, *inp=None*, *cwd=None*, *env=None*)

`modelparameters.commands.`**`get_status_output_errors`**(*cmd*, *inp=None*, *cwd=None*, *env=None*)

5.1.4 modelparameters.config module

modelparameters.config.**float_format**()

5.1.5 modelparameters.logger module

class modelparameters.logger.**Logger**(*name*)
Bases: object

add_log_indent(*increment=1*)
Add to indentation level.

add_logfile(*filename=None*, *mode='a'*)

begin_log(**message*)
Begin task: write message and increase indentation level.

debug(**message*)
Write debug message.

end_log()
End task: write a newline and decrease indentation level.

error(**message*, ***kwargs*)
Write error message and raise an exception.

flush_logger()
Flush the log handler

get_log_handler()
Get handler for logging.

get_log_level()
Get log level.

get_logfile_handler(*filename*)

get_logger()
Return message logger.

info(**message*)
Write info message.

info_blue(**message*)
Write info message in blue.

info_green(**message*)
Write info message in green.

info_red(**message*)
Write info message in red.

log(*level*, **message*)
Write a log message on given log level

pop_log_level()
Pop log level from the level stack, reverting to before the last push_level.

push_log_level(*level*)
Push a log level on the level stack.

remove_logfile(*filename*)

set_default_exception(*exception*)

set_log_handler(*handler*)
Replace handler for logging.

To add additional handlers instead of replacing the existing, use log.get_logger().addHandler(myhandler).

See the logging module for more details.

set_log_indent(*level*)
Set indentation level.

set_log_level(*level*)
Set log level.

set_log_prefix(*prefix*)
Set prefix for log messages.

set_raise_error(*value*)

suppress_logging()
Suppress all logging

type_error(**message*, ***kwargs*)
Write error message and raise a type error exception.

value_error(**message*, ***kwargs*)
Write error message and raise a value error exception.

warning(**message*)
Write warning message.

wrap_log_message(*message*, *symbol='*'*)

5.1.6 modelparameters.parameterdict module

Contains the ParameterDict class, useful for defining recursive dictionaries of parameters and using attribute syntax for later access.

class modelparameters.parameterdict.**Param**(*value*, *name=''*, *description=''*, ***kwargs*)
Bases: object

A simple type checking class for a single value

check(*value*)
Check the value using the type and any range check

convert_to(*unit*)
Convert parameter to a different unit than the current one.

Parameters **unit** (*str*) – The new unit

Returns Return the same prameter with the new unit

Return type *Param*

Example

```
>>> p_s = ScalarParam(1.0, unit="s")
>>> p_ms = p_s.convert_to('ms')
>>> print('value = {}, unit = {}'.format(p_ms.value), p_ms.unit))
value = 1000.0, unit = 'milliseconds'
```

copy(*include_checkarg=True*, *include_name=True*, *include_description=True*)
Return a copy of the parameter

Parameters

- **include_checkarg** (*bool*) – If include checkargs in new Param
- **include_name** (*bool*) – If include name in new Param
- **include_description** (*bool*) – If include description in new Param

description

format_data(*value=None*, *not_in=False*, *str_length=0*)
Print a nice formated version of the value and its range

Parameters

- **value** (*same as Param.value_type (optional)*) – A value to be used in the formating. If not passed stored value is used.
- **not_in** (*bool (optional)*) – If True return a not in version of the value
- **str_length** (*int (optional)*) – Used to pad the str with blanks

format_width()
Return the width of the formated str of value

getvalue()
Return the value

name

repr(*include_checkarg=True*, *include_name=True*, *include_description=True*)
Returns an executable version of the Param including optional arguments

Parameters

- **include_checkarg** (*bool*) – If include checkargs in new Param
- **include_name** (*bool*) – If include name in new Param
- **include_description** (*bool*) – If include description in new Param

setvalue(*value*, *check=True*)
Try to set the value using the check

update(*value*)

value
Return the value

class modelparameters.parameterdict.**ScalarParam**(*value*, *ge=None*, *le=None*, *gt=None*, *lt=None*, *unit='1'*, *name=''*, *description=''*)
Bases: *modelparameters.parameters.Param*

A simple type and range checking class for a scalar value

copy(*include_checkarg=True*, *include_name=True*, *include_description=True*, *include_unit=True*)
Return a copy of the parameter

Parameters

- **include_checkarg** (*bool*) – If include checkargs in new Param
- **include_name** (*bool*) – If include name in new Param
- **include_description** (*bool*) – If include description in new Param
- **include_unit** (*bool*) – If include unit in new Param

get_sym()

name

repr(*include_checkarg=True*, *include_name=True*, *include_description=True*, *include_unit=True*)
Returns an executable version of the Param including optional arguments

Parameters

- **include_checkarg** (*bool*) – If include checkargs in new Param
- **include_name** (*bool*) – If include name in new Param
- **include_description** (*bool*) – If include description in new Param
- **include_unit** (*bool*) – If include unit in new Param

sym

unit
Return the unit

update(*param*)
Update parameter with value of new parameter. Take into account unit conversion if applicable.

Parameters **param** (*ScalarParameter or scalar*) – The parameter with the new value

class modelparameters.parameterdict.**OptionParam**(*value*, *options*, *name=''*, *description=''*)
Bases: *modelparameters.parameters.Param*

A simple type and options checking class for a single value

repr(*include_checkarg=True*, *include_name=True*, *include_description=True*)
Returns an executable version of the Param including optional arguments

Parameters

- **include_checkarg** (*bool*) – If include checkargs in new Param
- **include_name** (*bool*) – If include name in new Param
- **include_description** (*bool*) – If include description in new Param

class modelparameters.parameterdict.**ConstParam**(*value*, *name=''*, *description=''*)
Bases: *modelparameters.parameters.Param*

A Constant parameter which prevent any change of values

class modelparameters.parameterdict.**ArrayParam**(*value*, *size=None*, *ge=None*, *le=None*, *gt=None*, *lt=None*, *unit='1'*, *name=''*, *description=''*)
Bases: *modelparameters.parameters.ScalarParam*

A numpy Array based parameter

resize(*newsize*)
Change the size of the Array

setvalue(*value*)
Set value of ArrayParameter

value
Return the value

class modelparameters.parameterdict.**SlaveParam**(*expr*, *unit='1'*, *name=''*, *description=''*)
Bases: *modelparameters.parameters.ScalarParam*

A slave parameter defined by other parameters

expr
Return the stored expression

format_data(*value=None*, *not_in=False*, *str_length=0*)
Print a nice formated version of the value and its range

getvalue()
Return a computed value of the Parameters

setvalue(*value*)
A setvalue method which always fails

value
Return a computed value of the Parameters

class modelparameters.parameterdict.**ParameterDict**(***params*)
Bases: dict

A dictionary with attribute-style access, that maps attribute access to the real dictionary.

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

copy(*to_dict=False*)
Make a deep copy of self, including recursive copying of parameter subsets.

Parameters **to_dict** (*bool (optional)*) – Return a dict with items representing the values of the Parameters

format_data(*indent=None*)
Make a recursive indented pretty-print string of self and parameter subsets.

fromkeys(**args*)
Create a new dictionary with keys from iterable and values set to value.

iterparameterdicts()
Iterate over all ParameterDicts

Parameters **recurse** (*bool (optional)*) – If True each encountered ParameterDict will also be entered

iterparams(*recurse=False*)
Iterate over all Param

Parameters **recurse** (*bool (optional)*) – If True each encountered ParameterDict will be entered

optstr()
Return a string with option set

An option string can be sent to a script using a parameter dict to set its parameters from command line options

parse_args(*options=None*, *usage=''*)
Parse a list of options. use sys.argv as default

Parameters **options** (*list of str (optional)*) – List of options. By default sys.argv[1:] is used. This argument is mostly for debugging.

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

update(*other*)
A recursive update that handles parameter subsets correctly unlike dict.update.

5.1.7 modelparameters.parameters module

modelparameters.parameters.**store_symbol_parameter**(*param*)
Store a symbol parameter

modelparameters.parameters.**value_namespace**(*expr*, *include_derivatives=False*)
Create a value name space for the included symbols in the expression

modelparameters.parameters.**symbols_from_expr**(*expr*, *include_numbers=False*, *include_derivatives=False*)
Returns a set of all symbols of an expression

Parameters

- **expr** (*sympy expression*) – A sympy expression containing sympy.Symbols or sympy.AppliedUndef functions.
- **include_numbers** (*bool*) – If True numbers will also be returned
- **include_derivatives** (*bool*) – If True derivatives will be returned instead of its variables

modelparameters.parameters.**symbol_to_param**(*sym*)
Take a symbol or expression of symbols and returns the corresponding Parameters

modelparameters.parameters.**pythoncode**(*expr*, *assign_to=None*, *namespace='math'*)
Return a Python-code representation of a sympy expression

modelparameters.parameters.**sympycode**(*expr*, *assign_to=None*)

modelparameters.parameters.**float_format**()

class modelparameters.parameters.**Logger**(*name*)
Bases: object

add_log_indent(*increment=1*)
Add to indentation level.

add_logfile(*filename=None*, *mode='a'*)

begin_log(**message*)
Begin task: write message and increase indentation level.

debug(**message*)
Write debug message.

end_log()
End task: write a newline and decrease indentation level.

error(**message*, ***kwargs*)
Write error message and raise an exception.

flush_logger()
Flush the log handler

get_log_handler()
Get handler for logging.

get_log_level()
Get log level.

get_logfile_handler(*filename*)

get_logger()
Return message logger.

info(**message*)
Write info message.

info_blue(**message*)
Write info message in blue.

info_green(**message*)
Write info message in green.

info_red(**message*)
Write info message in red.

log(*level*, **message*)
Write a log message on given log level

pop_log_level()
Pop log level from the level stack, reverting to before the last push_level.

push_log_level(*level*)
Push a log level on the level stack.

remove_logfile(*filename*)

set_default_exception(*exception*)

set_log_handler(*handler*)
Replace handler for logging.

To add additional handlers instead of replacing the existing, use log.get_logger().addHandler(myhandler).

See the logging module for more details.

set_log_indent(*level*)
Set indentation level.

set_log_level(*level*)
Set log level.

set_log_prefix(*prefix*)
Set prefix for log messages.

set_raise_error(*value*)

suppress_logging()
Suppress all logging

type_error(**message*, ***kwargs*)
Write error message and raise a type error exception.

value_error(**message*, ***kwargs*)
Write error message and raise a value error exception.

warning(**message*)
Write warning message.

wrap_log_message(*message*, *symbol='*'*)

modelparameters.parameters.**check_arg**(*arg*, *argtypes*, *num=-1*, *context=None*, *itemtypes=None*, *ge=None*, *le=None*, *gt=None*, *lt=None*)
Type check for positional arguments

Parameters

- **arg** (*any*) – The argument to be checked
- **argtypes** (*type, tuple*) – The type of which arg should be
- **num** (*int (optional)*) – The argument positional number
- **context** (*type, function/method (optional)*) – The context of the check. If context is a class the check is assumed to be during creation. If a function/method the contex is assumed to be a call to that function/method
- **itemtypes** (*type (optional)*) – If given argtypes must be a tuple or list and itemtypes forces each item to be a certain type
- **ge** (*scalar (optional)*) – Greater than or equal, range control of argument
- **le** (*scalar (optional)*) – Lesser than or equal, range control of argument
- **gt** (*scalar (optional)*) – Greater than, range control of argument
- **lt** (*scalar (optional)*) – Lesser than, range control of argument

modelparameters.parameters.**check_kwarg**(*kwarg*, *name*, *argtypes*, *context=None*, *itemtypes=None*, *ge=None*, *le=None*, *gt=None*, *lt=None*)
Type check for keyword arguments

Parameters

- **kwarg** (*any*) – The keyword argument to be checked
- **name** (*str*) – The name of the keyword argument
- **argtypes** (*type, tuple*) – The type of which arg should be
- **context** (*type, function/method (optional)*) – The context of the check. If context is a class the check is assumed to be during creation. If a function/method the contex is assumed to be a call to that function/method
- **itemtypes** (*type (optional)*) – If given argtypes must be a tuple or list and itemtypes forces each item to be a certain type
- **ge** (*scalar (optional)*) – Greater than or equal, range control of argument
- **le** (*scalar (optional)*) – Lesser than or equal, range control of argument
- **gt** (*scalar (optional)*) – Greater than, range control of argument
- **lt** (*scalar (optional)*) – Lesser than, range control of argument

modelparameters.parameters.**value_formatter**(*value*, *width=0*)
Return a formated string of a value

Parameters

- **value** (*any*) – The value which is formatted
- **width** (*int*) – A min str length value

class modelparameters.parameters.**Range**(*ge=None*, *le=None*, *gt=None*, *lt=None*)
Bases: object

A simple class for helping checking a given value is within a certain range

format(*value*, *width=0*)
Return a formated range check of the value

Parameters

- **value** (*scalar*) – A value to be used in checking range
- **width** (*int*) – A min str length value

format_in(*value*, *width=0*)
Return a formated range check

Parameters

- **value** (*scalar*) – A value to be used in checking range
- **width** (*int*) – A min str length value

format_not_in(*value*, *width=0*)
Return a formated range check

Parameters

- **value** (*scalar*) – A value to be used in checking range
- **width** (*int*) – A min str length value

modelparameters.parameters.**tuplewrap**(*arg*)
Wrap the argument to a tuple if it is not a tuple

class modelparameters.parameters.**Timer**(*task*)
Bases: object

Timer class

classmethod timings()
Return all registered timings

class modelparameters.parameters.**Param**(*value*, *name=''*, *description=''*, ***kwargs*)
Bases: object

A simple type checking class for a single value

check(*value*)
Check the value using the type and any range check

convert_to(*unit*)
Convert parameter to a different unit than the current one.

Parameters **unit** (*str*) – The new unit

Returns Return the same prameter with the new unit

Return type *Param*

Example

```
>>> p_s = ScalarParam(1.0, unit="s")
>>> p_ms = p_s.convert_to('ms')
>>> print('value = {}, unit = {}'.format(p_ms.value), p_ms.unit))
value = 1000.0, unit = 'milliseconds'
```

copy (*include_checkarg=True*, *include_name=True*, *include_description=True*)
Return a copy of the parameter

Parameters

- **include_checkarg** (*bool*) – If include checkargs in new Param
- **include_name** (*bool*) – If include name in new Param
- **include_description** (*bool*) – If include description in new Param

description

format_data (*value=None*, *not_in=False*, *str_length=0*)
Print a nice formated version of the value and its range

Parameters

- **value** (*same as Param.value_type (optional)*) – A value to be used in the formating. If not passed stored value is used.
- **not_in** (*bool (optional)*) – If True return a not in version of the value
- **str_length** (*int (optional)*) – Used to pad the str with blanks

format_width ()
Return the width of the formated str of value

getvalue ()
Return the value

name

repr (*include_checkarg=True*, *include_name=True*, *include_description=True*)
Returns an executable version of the Param including optional arguments

Parameters

- **include_checkarg** (*bool*) – If include checkargs in new Param
- **include_name** (*bool*) – If include name in new Param
- **include_description** (*bool*) – If include description in new Param

setvalue (*value*, *check=True*)
Try to set the value using the check

update (*value*)

value
Return the value

class modelparameters.parameters.**OptionParam** (*value*, *options*, *name=''*, *description=''*)
Bases: *modelparameters.parameters.Param*

A simple type and options checking class for a single value

repr (*include_checkarg=True*, *include_name=True*, *include_description=True*)
Returns an executable version of the Param including optional arguments

Parameters

- **include_checkarg** (*bool*) – If include checkargs in new Param
- **include_name** (*bool*) – If include name in new Param
- **include_description** (*bool*) – If include description in new Param

class modelparameters.parameters.**ConstParam**(*value*, *name=''*, *description=''*)
Bases: *modelparameters.parameters.Param*

A Constant parameter which prevent any change of values

class modelparameters.parameters.**TypelessParam**(*value*, *name=''*, *description=''*)
Bases: *modelparameters.parameters.Param*

A Typeless parameter allowing any change of value, including type changes

class modelparameters.parameters.**ScalarParam**(*value*, *ge=None*, *le=None*, *gt=None*, *lt=None*, *unit='1'*, *name=''*, *description=''*)
Bases: *modelparameters.parameters.Param*

A simple type and range checking class for a scalar value

copy(*include_checkarg=True*, *include_name=True*, *include_description=True*, *include_unit=True*)
Return a copy of the parameter

Parameters

- **include_checkarg** (*bool*) – If include checkargs in new Param
- **include_name** (*bool*) – If include name in new Param
- **include_description** (*bool*) – If include description in new Param
- **include_unit** (*bool*) – If include unit in new Param

get_sym()

name

repr(*include_checkarg=True*, *include_name=True*, *include_description=True*, *include_unit=True*)
Returns an executable version of the Param including optional arguments

Parameters

- **include_checkarg** (*bool*) – If include checkargs in new Param
- **include_name** (*bool*) – If include name in new Param
- **include_description** (*bool*) – If include description in new Param
- **include_unit** (*bool*) – If include unit in new Param

sym

unit
Return the unit

update(*param*)
Update parameter with value of new parameter. Take into account unit conversion if applicable.

Parameters **param** (*ScalarParameter or scalar*) – The parameter with the new value

class modelparameters.parameters.**ArrayParam**(*value*, *size=None*, *ge=None*, *le=None*, *gt=None*, *lt=None*, *unit='1'*, *name=''*, *description=''*)
Bases: *modelparameters.parameters.ScalarParam*

A numpy Array based parameter

resize(*newsize*)
Change the size of the Array

setvalue(*value*)
Set value of ArrayParameter

value
Return the value

class modelparameters.parameters.**SlaveParam**(*expr*, *unit='1'*, *name=''*, *description=''*)
Bases: *modelparameters.parameters.ScalarParam*

A slave parameter defined by other parameters

expr
Return the stored expression

format_data(*value=None*, *not_in=False*, *str_length=0*)
Print a nice formated version of the value and its range

getvalue()
Return a computed value of the Parameters

setvalue(*value*)
A setvalue method which always fails

value
Return a computed value of the Parameters

modelparameters.parameters.**eval_param_expr**(*expr*, *param_ns=None*, *include_derivatives=False*, *ns=None*)
Eval an expression of symbols of ScalarParam

Parameters

- **expr** (*expression of ParamSymbols*) – The expression to be evaulated
- **param_ns** (*dict (optional)*) – A namespace containing the parameters for which the expr should be evaluated with.
- **include_derivatives** (*bool (optional)*) – If True not only symbols are evaulated but also derivatives
- **ns** (*dict (optional)*) – A namespace in which the expression will be evaluated in

5.1.8 modelparameters.sympytools module

modelparameters.sympytools.**check_arg**(*arg*, *argtypes*, *num=-1*, *context=None*, *itemtypes=None*, *ge=None*, *le=None*, *gt=None*, *lt=None*)
Type check for positional arguments

Parameters

- **arg** (*any*) – The argument to be checked
- **argtypes** (*type, tuple*) – The type of which arg should be
- **num** (*int (optional)*) – The argument positional number
- **context** (*type, function/method (optional)*) – The context of the check. If context is a class the check is assumed to be during creation. If a function/method the contex is assumed to be a call to that function/method

- **itemtypes** (*type (optional)*) – If given argtypes must be a tuple or list and itemtypes forces each item to be a certain type
- **ge** (*scalar (optional)*) – Greater than or equal, range control of argument
- **le** (*scalar (optional)*) – Lesser than or equal, range control of argument
- **gt** (*scalar (optional)*) – Greater than, range control of argument
- **lt** (*scalar (optional)*) – Lesser than, range control of argument

modelparameters.sympytools.**deprecated**(*func*)
This is a decorator which can be used to mark functions as deprecated. It will result in a warning being emitted when the function is used.

modelparameters.sympytools.**Conditional**(*cond*, *true_value*, *false_value*)
Declares a conditional

Parameters

- **cond** (*A conditional*) – The conditional which should be evaluated
- **true_value** (*Any model expression*) – Model expression for a true evaluation of the conditional
- **false_value** (*Any model expression*) – Model expression for a false evaluation of the conditional

modelparameters.sympytools.**ContinuousConditional**(*cond*, *true_value*, *false_value*, *sigma=1.0*)
Declares a continuous conditional. Instead of a either or result the true and false values are weighted with a sigmoidal function which either evaluates to 0 or 1 instead of the true or false.

Parameters

- **cond** (*An InEquality conditional*) – An InEquality conditional which should be evaluated
- **true_value** (*Any model expression*) – Model expression for a true evaluation of the conditional
- **false_value** (*Any model expression*) – Model expression for a false evaluation of the conditional
- **sigma** (*float (optional)*) – Determines the sharpness of the sigmoidal function

modelparameters.sympytools.**store_symbol_parameter**(*param*)
Store a symbol parameter

modelparameters.sympytools.**symbol_to_params**(*sym*)

modelparameters.sympytools.**symbol_to_param**(*sym*)
Take a symbol or expression of symbols and returns the corresponding Parameters

modelparameters.sympytools.**symbols_from_expr**(*expr*, *include_numbers=False*, *include_derivatives=False*)
Returns a set of all symbols of an expression

Parameters

- **expr** (*sympy expression*) – A sympy expression containing sympy.Symbols or sympy.AppliedUndef functions.
- **include_numbers** (*bool*) – If True numbers will also be returned

- **include_derivatives** (*bool*) – If True derivatives will be returned instead of its variables

modelparameters.sympytools.**iter_symbol_params_from_expr**(*expr*)
Return an iterator over sp.Symbols from expr

modelparameters.sympytools.**symbol_params_from_expr**(*expr*)
Return a list of Symbols from expr

modelparameters.sympytools.**symbol_param_value_namespace**(*expr*)
Create a value name space for the included symbols in the expression

modelparameters.sympytools.**value_namespace**(*expr*, *include_derivatives=False*)
Create a value name space for the included symbols in the expression

modelparameters.sympytools.**add_pair_to_subs**(*subs*, *old*, *new*)
Add a pair of old and new symbols to subs. If a subs with old as a key already excist it will be removed before insertion.

5.1.9 modelparameters.utils module

modelparameters.utils.**ClassType**
alias of builtins.type

class modelparameters.utils.**Logger**(*name*)
Bases: object

add_log_indent(*increment=1*)
Add to indentation level.

add_logfile(*filename=None*, *mode='a'*)

begin_log(**message*)
Begin task: write message and increase indentation level.

debug(**message*)
Write debug message.

end_log()
End task: write a newline and decrease indentation level.

error(**message*, ***kwargs*)
Write error message and raise an exception.

flush_logger()
Flush the log handler

get_log_handler()
Get handler for logging.

get_log_level()
Get log level.

get_logfile_handler(*filename*)

get_logger()
Return message logger.

info(**message*)
Write info message.

info_blue(**message*)
Write info message in blue.

info_green(**message*)
Write info message in green.

info_red(**message*)
Write info message in red.

log(*level*, **message*)
Write a log message on given log level

pop_log_level()
Pop log level from the level stack, reverting to before the last push_level.

push_log_level(*level*)
Push a log level on the level stack.

remove_logfile(*filename*)

set_default_exception(*exception*)

set_log_handler(*handler*)
Replace handler for logging.

To add additional handlers instead of replacing the existing, use log.get_logger().addHandler(myhandler).

See the logging module for more details.

set_log_indent(*level*)
Set indentation level.

set_log_level(*level*)
Set log level.

set_log_prefix(*prefix*)
Set prefix for log messages.

set_raise_error(*value*)

suppress_logging()
Suppress all logging

type_error(**message*, ***kwargs*)
Write error message and raise a type error exception.

value_error(**message*, ***kwargs*)
Write error message and raise a value error exception.

warning(**message*)
Write warning message.

wrap_log_message(*message*, *symbol='*'*)

modelparameters.utils.**float_format**()

modelparameters.utils.**reduce**(*function*, *sequence*[, *initial*]) → value
Apply a function of two arguments cumulatively to the items of a sequence, from left to right, so as to reduce the sequence to a single value. For example, reduce(lambda x, y: x+y, [1, 2, 3, 4, 5]) calculates ((((1+2)+3)+4)+5). If initial is present, it is placed before the items of the sequence in the calculation, and serves as a default when the sequence is empty.

modelparameters.utils.**rjust**(*s*, **args*, ***kwargs*)

modelparameters.utils.**VALUE_JUST**(*s*, **args*, ***kwargs*)

modelparameters.utils.**param2value**(*param*)

modelparameters.utils.**value_formatter**(*value*, *width=0*)
Return a formated string of a value

Parameters

- **value** (*any*) – The value which is formatted
- **width** (*int*) – A min str length value

class modelparameters.utils.**Range**(*ge=None*, *le=None*, *gt=None*, *lt=None*)
Bases: object

A simple class for helping checking a given value is within a certain range

format(*value*, *width=0*)
Return a formated range check of the value

Parameters

- **value** (*scalar*) – A value to be used in checking range
- **width** (*int*) – A min str length value

format_in(*value*, *width=0*)
Return a formated range check

Parameters

- **value** (*scalar*) – A value to be used in checking range
- **width** (*int*) – A min str length value

format_not_in(*value*, *width=0*)
Return a formated range check

Parameters

- **value** (*scalar*) – A value to be used in checking range
- **width** (*int*) – A min str length value

modelparameters.utils.**format_time**(*time*)
Return a formated version of the time argument

Parameters **time** (*float*) – Time given in sections

class modelparameters.utils.**Timer**(*task*)
Bases: object

Timer class

classmethod timings()
Return all registered timings

modelparameters.utils.**list_timings**()
List all registered timings

modelparameters.utils.**clear_timings**()
Clear all registered timings

modelparameters.utils.**tic**()
Start timing

modelparameters.utils.**toc**()
Return timing since last toc/tic

modelparameters.utils.**is_iterable**(*obj*)
Test for iterable

Parameters **obj** (*any*) – Object which is beeing tested

modelparameters.utils.**add_iterable**(*iterable*, *initial=None*)
Sum the content of an iterable

modelparameters.utils.**camel_capitalize**(*name*)
Camel capitalize a str

modelparameters.utils.**tuplewrap**(*arg*)
Wrap the argument to a tuple if it is not a tuple

modelparameters.utils.**listwrap**(*arg*)
Wrap the argument to a list if it is not a list

modelparameters.utils.**check_arginlist**(*arg*, *lst*, *name='arg'*)
Check that arg is in lst

modelparameters.utils.**check_arg**(*arg*, *argtypes*, *num=-1*, *context=None*, *itemtypes=None*, *ge=None*, *le=None*, *gt=None*, *lt=None*)
Type check for positional arguments

Parameters

- **arg** (*any*) – The argument to be checked
- **argtypes** (*type, tuple*) – The type of which arg should be
- **num** (*int (optional)*) – The argument positional number
- **context** (*type, function/method (optional)*) – The context of the check. If context is a class the check is assumed to be during creation. If a function/method the contex is assumed to be a call to that function/method
- **itemtypes** (*type (optional)*) – If given argtypes must be a tuple or list and itemtypes forces each item to be a certain type
- **ge** (*scalar (optional)*) – Greater than or equal, range control of argument
- **le** (*scalar (optional)*) – Lesser than or equal, range control of argument
- **gt** (*scalar (optional)*) – Greater than, range control of argument
- **lt** (*scalar (optional)*) – Lesser than, range control of argument

modelparameters.utils.**check_kwarg**(*kwarg*, *name*, *argtypes*, *context=None*, *itemtypes=None*, *ge=None*, *le=None*, *gt=None*, *lt=None*)
Type check for keyword arguments

Parameters

- **kwarg** (*any*) – The keyword argument to be checked
- **name** (*str*) – The name of the keyword argument
- **argtypes** (*type, tuple*) – The type of which arg should be
- **context** (*type, function/method (optional)*) – The context of the check. If context is a class the check is assumed to be during creation. If a function/method the contex is assumed to be a call to that function/method
- **itemtypes** (*type (optional)*) – If given argtypes must be a tuple or list and itemtypes forces each item to be a certain type
- **ge** (*scalar (optional)*) – Greater than or equal, range control of argument

- **le** (*scalar (optional)*) – Lesser than or equal, range control of argument
- **gt** (*scalar (optional)*) – Greater than, range control of argument
- **lt** (*scalar (optional)*) – Lesser than, range control of argument

modelparameters.utils.**quote_join**(*list_of_str*)
Join a list of strings with quotes and commans

modelparameters.utils.**deprecated**(*func*)
This is a decorator which can be used to mark functions as deprecated. It will result in a warning being emitted when the function is used.

5.1.10 Module contents

CHAPTER 6

Indices and tables

- genindex
- modindex
- search

Python Module Index

m

Index

I

J

L

M

N

O

P

Q

R

S

T

U

V

W