Python Parameters Documentation

Release 1.9.0

Matthew Wardrop

April 27, 2015

CONTENTS

CHAPTER

ONE

INTRODUCTION

Keeping track of parameters during simulations can often be cumbersome, especially if those parameters are time- or co- dependent, or if unit conversions are necessary. The ParamPy (*parampy*) module solves this problem by acting as a central repository of all parameters, their non-dimensionalisation, their interdependencies, their units and their bounds. While providing all of this functionality, ParamPy also attempts to maintain minimal overhead so that it is suitable for use in problems requiring iteration (such as numerical integration).

Properly enumerated, the python-parameters module can:

- Act as a central location for the storage and retrieval of model parameters.
- Keep track of parameter units, and perform unit conversions where that makes sense.
- Perform non-dimensionalisation of parameters in a consistent fashion to allow for simple model simulations.
- Allow parameters to inter-depend upon one another, and ensure that parameters do not depend upon one another in unresolvable ways.
- Allow parameters to depend on as-yet undeclared runtime values (such as integration time).
- Keep track of limits for parameter values, to ensure parameters do not escape pre-defined parameter ranges.
- Provide this functionality with minimal overhead to allow for speedy simulations.

As of version 1.9.0 (the version at time of publication); simple parameter storage and retrieval is only a factor of $\approx$10 slower than a python variable set and read; though speeds decrease depending upon how many of the more sophisticated features are used (such as parameter bounding).

All features are documented, and most are unittested.

CHAPTER

TWO

INSTALLATION

In most cases, installing this module is as easy as:

```
python2 setup.py install
```

If you run Arch Linux, you can instead run:

```
makepkg -i
```

CHAPTER

THREE

QUICK START

In this section, the syntax of ParamPy for each major feature is shown, along with simple examples of use within the python interpreter. For the most part, the syntax should straightforward to use. If in doubt, refer to the API documentation in a subsequent chapter.

While the Parameters module is extremely flexible, allowing arbitrary unit definition and non-dimensionalisation scalings, in most cases it should not be necessary to take advantage of this flexibility. Otherwise, there are two main classes that are likely to be immediately useful to user:

- `SIQuantity` (a subclass of `Quantity`)
- `Parameters`

`Quantity` instances are the fundamental representation of physical quantities in ParamPy, and the `SIQuantity` subclass recognises all SI units as well as some addition units the author found useful. For more information, see the API documentation in the next chapter. Instances of the Parameters class do the heavy lifting of managing potentially interdependent parameters and their non-dimensionalisation (among other things). Both of these classes can be imported from the parameters module using:

```
>>> from parampy import SIQuantity, Parameters
```

We will give a quick overview of their functionality in the following sections.

3.1 SIQuantity

To create a SIQuantity object, we initialise it with a value and some valid units (see the "Supported Units" chapter). The value specified can be any integer, float or numpy array (or types conforming to standard python numeric operations). For example:

```
>>> q = SIQuantity(3, 'm*s')
>>> q
3 m*s
>>> r = SIQuantity(2.14, 'GN*nm')
>>> r
2.14 GN*nm
```

Arithmetic with SIQuantity instances is very simple, and support is provided for unit conversion, addition, subtraction, multiplication, division, raising to some power and equality tests. In cases where the operation does not make sense, an exception is raised with details as to why. Examples are now provided for each of these operations in turn.

```
>>> SIQuantity(3,'m/s^2')('in/ns^2') # Unit Conversion
1.1811023622e-16 in/ns^2
```

```
>>> SIQuantity(3,'m')+SIQuantity(2,'mm') # Addition
3.002 m

>>> SIQuantity(3,'kJ')-SIQuantity(2,'cal') # Subtraction
2.9916264 kJ

>>> SIQuantity(3.,'mT') / SIQuantity(10,'s') # Division
0.3 mT/s

>>> 2*SIQuantity(3.,'mT') # Scaling
6.0 mT

>>> SIQuantity(2.6,'A')**2 # Squaring
6.76 A^2

>>> SIQuantity(1,'W*h') == SIQuantity(1e-9,'GW*h')
True

>>> SIQuantity(3,'m') + SIQuantity(2,'s') # Nonsense addition
UnitConversionError: Invalid conversion. Units 's' and 'm' do not match.
```

In each of the above examples, the returned object is another SIQuantity object, which can also undergo further arithmetic operation. For convenience, it is not necessary to manually instantiate all physical quantities into SIQuantity objects; provided that at least one of the operands involved in the calcuation is already an SIQuantity object. For example, the addition example above could be simplified to:

```
>>> SIQuantity(3,'m')+(2,'mm') # Addition
3.002 m

>>> (3,'m')+SIQuantity(2,'mm') # Addition
3.002 m
```

Be careful when using this contraction that order of operations does not cause the tuple to perform some undefined operation with something else. For example:

```
>>> 2*(3,'m')+SIQuantity(2,'mm') # This becomes (3,'m',3,'m') + SIQuantity(2,'mm'), and fails
```

The Quantity class, which is the base class for the SIQuantity class is quite flexible, and if you want to implement a non-SI system, or add custom units, refer to the API documentation.

3.2 Parameters

The Parameters class handles the heavy lifting of managing named parameters, their values, relationships and conversions; using Quantity objects to represent physical quantities. Initialising a Parameters object is done using:

```
>>> p = Parameters()
```

There are three optional arguments that can be passed to this initialiser:

- default_scaled=True: Whether parameters should return non-dimensionalised values instead of Quantity objects. By default, this is True. This can be negated at runtime.
- constants=False: Whether to populate the parameter namespaces with various physical constants, all of which are prefixed with "c_" (e.g. c_hbar). By default, this is False.
- dispenser=None: A custom UnitDispenser to use instead of the standard SI one. Most users will not need to touch this.

Unlike the `SIQuantity` class, the `Parameters` class has a lot of power, and a lot of subtlety. If you are planning to use advanced features of the Parameters class (such as parameters depending on other parameters), it would probably be best to skip directly to the API documentation for the `Parameters` class, where everything is enumerated in detail. In the next few subsections, only the basic functionality of this class will be explored.

3.2.1 Parameter Definition

Defining a parameter (all of the following are equivalent except when defining parameters that are functions of others):

```
>>> p(x=(1.3,'kg*ns'))           # Method 1
>>> p << {'x': (1.3,'kg*ns')}    # Method 2
>>> p['x'] = (1.3,'kg*ns')       # Method 3
>>> p.x = (1.3,'kg*ns')          # Method 4
```

The first and second methods can be extended to add multiple parameters at once:

```
>>> p(x=1.3, y=2.1, z=4.1)
>>> p << {'x': 1.3, 'y': 2.1, 'z': 4.1}
```

Parameters can also be dependent on one another, by defining them as a function or symbolic expression (as a string).

```
>>> p(x=lamda y,z: y**2 + z**2)                  # Method 1
>>> p << {'x': 'y^2 + z^2'}                  # Method 2
>>> p['x'] = lamda y,z: 'y^2 + z^2'          # Method 3
>>> p.x = lamda y,z: y**2 + z**2             # Method 4
```

Methods 1 & 3 evaluate the function **before** setting it as the value of the parameter, whereas methods 2 & 4 cause the function to change with any future change in the underlying parameters. An exception will be raised if an attempt is made to cause parameters to circularly depend on one another.

Parameter names can be any legal python variable name (except those starting with an underscore). Parameter values can be any numeric type (including complex), any function/lambda object (or a string representing a mathematical expression which is then converted to a function), any tuple of preceeding types with a unit, and any Quantity object.

To see a list of units stored in a `Parameters` instance, use:

```
>>> p.show()
```

To remove a parameter definition, simply use the forget method:

```
>>> p.forget('x','y','z',...)
```

For more details, including how to make parameter dependencies invertible, to enforce bounds on parameters, and how to load and save parameters from a file, please refer to the API documentation.

3.2.2 Parameter Extraction

Extracting a parameter (all of the following methods are equivalent):

```
>>> p('x')      # Method 1
>>> p.x         # Method 2
>>> p['x']      # Method 3
```

By default, all returned units are non-dimensionalised first. To extract the physical quantity as a `Quantity` object, simply prepend the variable name with an underscore.

```
>>> p('_x')
>>> p._x
>>> p['_x']
```

There is also a special fourth method for inverting the default behaviour and returning a Quantity object by default (and is otherwise like method 1).

```
>>> p._('x') # Method 4
```

Methods 1 & 4 can be used to extract more than one parameter at once, in which case they are returned as a dictionary of values:

```
>>> p('x','y','z')
{'x': <value>, 'y': <value>, 'z': <value>}
>>> p._('x','y','z')
{'x': <quantity>, 'y': <quantity>, 'z': <quantity>}
```

Additionally, methods 1 & 4 can also temporarily override parameters as they extract the parameters, which is useful for testing how parameters behave in diferrent parameter contexts.

```
>>> p('x','y','z', k=<value>, l=<value>)
{'x': <value>, 'y': <value>, 'z': <value>}
>>> p._('x','y','z', k=<value>, l=<value>)
{'x': <quantity>, 'y': <quantity>, 'z': <quantity>}
```

For more information about extracting parameters, including parameter ranges, plotting, and more; please refer to the API documentation.

3.2.3 Unit Conversion

It is possible to use a `Parameters` object for other tasks, such as unit conversion. The syntax for this is:

```
>>> p.convert( <value>, input=<input units>, output=<output units>, value=<True/False> )
```

where *input* is the units that Parameters should assume the object has, *output* is the desired output units, and *value* (with default True) specifies whether or not you are only interested in its numerical value, or whether a Quantity object should be returned. If not specified, or equal to None, the input and output units are assumed to refer to the non-dimensional quantities used by the Parameters object.

For example, the following command converts 1 ns to a non-dimensional quantity:

```
>>> p.convert(1, input='ns')
1e-9
```

The following command converts 4 J to a value with units of calories:

```
>>> p.convert(4, input='J', output='cal')
0.9553835865099838
```

The following command converts 5 /day to a Quantity with units of years:

```
>>> p.convert(5, input='/day', output='/year', value=False)
1826.21095 /year
```

Note that this utility still works for Quantity units as well, in which case the ‘input‘ argument is ignored, and read from the Quantity object.

```
>>> p.convert(SIQuantity(1,'km'),output='m')
1000.0
```

CHAPTER

FOUR

API DOCUMENTATION

In this section, documentation is provided at a per-method level, allowing you to write easily write scripts and problems which conform to the API. Future updates with same major version should maintain backwards compatibility with this API. If you find a problem or inconsistency in this documentation, that is a bug; and the author would like to know about it.

The ordering of the API is from most often used to least often used classes (for an expected average user); and so we begin with the `Parameters` class, then the `Quantity` class, before moving to less used and more foundational classes. Not documented here are classes and methods which the author deems to be for internal use only.

4.1 The Parameters Class

class `parampy.parameters.`**`Parameters`**

Bases: `object`

Parameters(dispenser=None, default_scaled=True, constants=False)

`Parameters` is the main class in the `parameters` package, and acts to organise and manage potentially interdependent physical quantities. The functionality of this class includes:

- Non-dimensionalisation of parameters
- Managing parameters which are functions of other parameters
- Performing unit conversions of parameters
- Iterating over ranges of parameters
- Putting bounds on parameters

A lot of the functionality of the `Parameters` class is handled in "magic" methods, meaning that you will not necessarily find what you want to do by looking at the method documentation alone. In the rest of the class documentation, we cover the use of Parameters magic methods, alluding to the public methods where appropriate.

Parameters

- **dispenser** (*UnitDispenser or None*) – Should provide None or a custom unit dispenser. If None is provided, Parameters will instantiate an SIUnitDispenser; which hosts the standard units relative to an SI basis.
- **default_scaled** (*bool*) – `True` when Parameters should by default return scaled non-dimensional parameters. `False` otherwise.
- **constants** (*bool*) – `True` when Parameters should import the physical constants when the internal `UnitDispenser` is of type `SIUnitDispenser`. `False` otherwise.

Initialising a Parameters Instance: Initialising a `Parameters` instance with the default configuration is simple:

```
>>> p = Parameters()
```

If you want to use your own `UnitDispenser`, use united parameters by default, and preload the `Parameters` instance with a set of physical constants, you could use:

```
>>> u = SIUnitDispenser()
>>> p = Parameters(dispenser=u, default_scaled=False, constants=True)
```

Seeing what is stored in a Parameters instance: To see an overview of the parameter object; simply use:

```
>>> p.show() # See show documentation for more.
```

To get a list of parameters stored, use:

```
>>> list(p)
```

To check if a parameter is stored:

```
>>> is_stored = 'x' in p # Will be True if 'x' is in p
```

Parameter Definition: There are several ways to define parameters, each of which is shown below:

```
>>> p(x=1)
>>> p << {'x':1}
>>> p['x'] = 1
>>> p.x = 1
```

The first two methods generalise to defining multiple parameters at once, whereas the second two do not. The first and third set parameters *after* first interpreting the value given (in a manner to be described below), whereas the others do not.

Note: The last method will even when the parameter name clashes with a method of `Parameters`, which you will be warned about when you define such a parameter. It will not be possible, however, to retrieve such a parameter using attribute notation later. Fortunately, "interpretation" makes no difference when retrieving parameters, and so the other methods will work as expected.

Let us begin with the properties that all of these methods share.

- Parameter names must start with a letter; and may contain any number of additional letters, numbers, and underscores.
- **Parameter values may be:**
  - any numeric type (include `complex`).
  - a string that represents a symbolic expression.
  - a function with zero or more arguments, each of which being the name of a parameter.
  - a two-tuple of any of the above types with a valid unit (see the Supported Units section of the Parameters documentation for supported SI units in a `SIUnitDispenser`).
  - a `Quantity` instance

Here are some examples of valid parameter-value combinations:

```
>>> p.x=1.42
>>> p.alpha = 'sin(x)'
>>> p.beta2 = lambda x, alpha: math.sqrt(x+alpha)
```

```
>>> p.y = (32.1, '{mu}eV')
>>> p.q_z_ = ('x+2*beta2', 'ms')
```

When a unit is not specified, the parameter is assumed to be a scaled version of the parameter. If the unit has never been set, then a 'constant' unit is assumed; otherwise the older unit is assumed.

You can specify the units of a parameter, without specifying its value:

```
>>> p & {'x':'ms'}
```

Note: If you do this when the parameter has already been set, the parameter value will be adjusted to maintain a constant physical value. If a parameter has been set, and you try to change the units to another which is dimensionally incompatible (e.g. 'm' -> 'ms'), this will result in a `UnitConversionError` being raised. You should first ask the `Parameters` instance to "forget" this value, as described shortly.

Defining multiple parameters at once is easy with the first two methods:

```
>>> p(x=1,y=(2,'s'),z=lambda x,y: x+y)
>>> p << {'x':1,'y':(2,'s'),'z': lambda x,y: x+y }
```

Note, though, that these two statements are **not** equivalent, as described in the next section.

Interpreted vs Non-interpreted Parameter Definition: In the previous section, a distinction was made between methods of defining parameters that first "interpret" the value and those that do not. In particular: `p(?=...)` and `p[?] = ...` were said to first interpret the value, whereas `p << {?: ...}` and `p.? = ...` did not.

The difference amounts to whether or not functions are first evaluated before being saved as a parameter value. For example:

```
>>> p['x'] = lambda y: y**2 # Evaluated immediately
>>> p.is_function('y')
False
>>> p.x = lambda y: y**2 # Evaluated only when x is determined
>>> p.is_function('x')
True
```

Succinctly, interpreted parameter definitions set the parameter to the value of the function at time of definition; whereas non-interpreted parameter definitions cause the parameter to be evaluated in whatever context it finds itself in the future.

Parameter Extraction: Just as there are several ways to define parameters, there are several ways to extract them. They are:

```
>>> p('x')
>>> p.x
>>> p['x']
```

Each of the above returns either a single non-dimensional value or a `Quantity` depending upon whether default_scaled was passed as True or False respectively in the parameter instance. The first of these allows for multiple parameters to be extracted, whereas the second two do not. Additionally, the first of these also allows for temporarily overriding the parameter context. For example:

```
>>> p('x','y',z=(10,'ms'))
```

The above example returns a dictionary which contains a mapping from 'x' and 'y' to their respective values; evaluated in the context of 'z' being overridden temporarily with a value of 10 ms. At the end of this evaluation, 'z' maintains whatever value it had before. This only affects the value of requested variables if it is a function of those variables.

Note: As in most functions in this class that accept multiple parameter arguments, a dictionary is returned if there are two or more parameters; otherwise the value of the single requested parameter is returned. You can force a dictionary response by encapsulating this parameter in a list. For example:

```
>>> p(['x'])
```

As mentioned, if the `Parameter` instance was instantiated with `default_scaled` set to `True`, then all of the parameters requested by name will return a non-dimensionalised number; or if it was `False`, a `Quantity` object. To invert this default, simply add a '_' to the variable name. For example:

```
>>> p('_x')
>>> p._x
>>> p['_x']
```

As a shorthand for extracting multiple parameters, you can use the following special method:

```
>>> p._('x','y',z=(10,'ms'))
```

This is identical in function to:

```
>>> p('_x','_y',z=(10,'ms'))
```

Although redundant, this special method can also set parameter values just as calling the `Parameter` instance does:

```
>>> p._(x=10)
```

Parameter Interdependencies: As already discussed, parameters within a `Parameters` instance can depend on values of other parameters by setting their value to be a function or symbolic expression. In this section we describe how to best take advantage of this functionality.

Firstly, the function may either be a full function, method or lambda expression.

Secondly, the function must take as arguments only parameter names which exist in the `Parameters` instance at the time that it is executed. Thus, "non-interpreted" parameter definitions can be set to include variables that are only defined at runtime; such as the current time in a numerical integration. Beyond that, there are no restrictions, except that the function must return a value that is understood (this can both a numeric quantity, or a united quantity tuple, or a `Quantity` object).

Thirdly, by using a variable name prepended with an underscore in the function declaration, you can access both non-dimensional and dimensional quantities in your function.

Fourthly, you can cause your function to be invertible by adding a reference to the variable name you are defining to the end of the list of function arguments, with a default value. When that variable is specified, you should return an ordered list/tuple of updated parameters for the other parameters in the function declaration. If there is only one other, it is sufficient to simply return it. For example:

```
>>> p.x = lambda y,x=None: y**2 if x is None else x**0.5

>>> p << {'y': 1, 'z': 1, 'x': lambda y,z,x=None: y+z if x is None else [y,x-y]}
>>> p('x')
2.0
>>> p('z',x=10)
9.0
```

Lastly, parameter relationships can chained and as deep as you like. For example, x could depend on y, which could depend on z, and so on. The `Parameters` will ensure that there are no dependency loops.

Removing Parameters: To remove a parameter, simply use the forget method:

```
>>> p.forget('x','y',...)
```

As many parameters as you like can be specified in one function call.

Parameter Units and Scaling: Custom units can be added by using the `unit_add()` method, and scaling used in the non-dimensionalisation process can be extracted for a given unit using the `unit_scaling()` method.

You can customise the scaling used in the non-dimensionalisation process by changing the scaling of the various different dimensions using the `scaling()` method. By default, units are scaled relative to the SI fundamental units. The current scaling for a dimension can also be extracted using this function.

Unit Conversion: Please see the documentation for the `convert()` method.

Parameter Bounds: Please see the documentation for the `bounds()` and `set_bounds()` methods.

Parameter Ranges: It is often useful to iterate over a range of parameter values. You can generate such ranges using the `range()` method. You can also have the iteration process handled for you, as documented in the `range_iterator()` method.

Physical Constants: If `constants` was `True` when the `Parameters` instance was initialised, and a `SIUnitDispenser` is being used (as it is by default), then the parameter list is prepopulated with a list of physical constants. Please see the "Physical Constants" section of the *python-parameters* documentation.

Loading and Saving Parameter Sets: To load a parameter set into a Parameters instance, use the classmethod `load()`. See its documentation for more information.

To save your existing parameters, use:

p >> "filename.py"

Note that parameters that are dependent on other parameters will not survive this transition, and will be saved as static values.

Parameter Contexts: Parameters objects support Python's "with" syntax. Upon exiting a "with" environment, any changes made will be reset to before re-entering the parent environment.

```
>>> with p:
>>>     p(x=1)
>>> p('x') # Returns value of x before entering the with environment.
```

asscaled(***kwargs*)

Parameters **kwargs** – A dictionary of parameter values

Returns Number (normally float, but could be complex, etc)

A utility function to return what the scaled value of a parameter would be if it were overridden with *kwargs*. It is the logical partner of *asvalue(**kwargs)*. For example:

```
>>> p.asscaled(x=(1, 'm'))
1
```

Note that this is equivalent to `p.convert((1, 'm'))`. If multiple parameter values are specified, a dictionary of values is returned:

```
>>> p.asscaled(x=1, y=2)
{'x': 1, 'y': 2}
```

asvalue(***kwargs*)

Parameters **kwargs** – A dictionary of parameter values

Returns Number (normally float, but could be complex, etc)

A utility function to return what the united value of a parameter would be if it were overridden with *kwargs*. It is the logical partner of *asscaled(**kwargs)*. For example:

```
>>> p.asvalue(x=(1, 'nm'))
1
```

If no units are passed, then inout value is assumed to be non-dimensional, and units are lifted from the underlying parameter. For example:

```
>>> p.x = (1,'ms')
>>> p.asvalue(x=1)
1000
```

If multiple parameter values are specified, a dictionary of values is returned: >>> p.asvalue(x=1, y=2) {'x': 1000, 'y': 2}

bounds (**params*, ***bounds*)

Parameters

- **params** – Sequence of parameter names for which to query the bounds.
- **bounds** (*dict*) – Dictionary specifying parameter bounds to be applied.

Returns Dictionary of `Bound` objects, or, if only one parameter is specified, then just one `Bound` object. If no bounds exist for a parameter, `None` is returned.

This method allows you to simply query and set bounds on parameters. For more advanced bounds setting, you should use `set_bounds()`. For each parameter, valid bounds are specified by a two-tuple, or a list of two-tuples. If one of the extremum values is None, it is set to -infinity or +infinity, depending upon whether it is the upper or lower bound.

For example:

```
>>> p.bounds(x=(0,100)) # Bounds x between 0 and 100 inclusive.

>>> p.bounds(y=[(0,50),(150,200)]) # Bounds y between 0-50 or between 150-200.

>>> p.bounds('x') # Returns the Bounds object associated with x

>>> p.bounds('x',x=(0,None)) # Sets the bounds on x to be [0,+inf], and then returns the :cl
```

Note that multiple parameters can be queried and set at the same time.

> **Warning:** Using parameter bounds greatly increases the amount of computation done during parameter retrieval. It is recommended that you do not use parameter bounds in contexts which require minimal runtime, such as numerical integration.

cache (***kwargs*)

Parameters **kwargs** (`dict`) – Dictionary of boolean values

A utility function to toggle caching of particular parameters. When cache is enabled, if a parameter function has been called before with the same parameter values, then it returns the old value. Note that only one set of parameters is remembered, and so this caching is designed only for situations where the parameter is not expected to change at every call.

Example:

```
>>> p.cache(x=True, y=False)
```

This will enable caching for *x* and disable it for *y*.

convert (*self*, *quantity*, *input=None*, *ouput=None*, *value=True*)

Parameters

- **quantity** (`Quantity`, *Quantity* tuple representation or any pythonic numeric type (including numpy arrays).) – The quantity to be converted.
- **input** (`None`, `str`, or `Units`) – The units of the inputed quantity (ignored if input type is `Quantity`).
- **output** (`None`, `str`, or `Units`) – The units to convert toward.
- **value** (`bool`) – Whether the function should return only the value (rather than the full `Quantity` object).

Returns Pythonic number if `value` is `True`, and `Quantity` otherwise.

forget (**params*)

Parameters **params** (`tuple` of `str`) – List of parameter names to forget.

Returns A reference to the parent `Parameters` instance.

This is the way to remove parameters that are stored inside a `Parameters` instance.

Example:

```
>>> p.forget('x','y','z')
```

This removes parameters *x*, *y* and *z* from the parameter list.

is_constant (**args*, ***params*)

Parameters

- **param** (*str*) – The param for which to test constancy.
- **wrt** (*tuple*) – A sequence of parameter names, with respect to which *param* should be constant.
- **params** (*dict*) – Dictionary of parameter value overrides.

Returns `True` if first listed parameter is independent of all subsequent parameters. `False` otherwise.

This method is useful for simplifying parameters in contexts where parameters are going to be often polled, by checking to see if they can be statically cached in your application. For example:

```
>>> p.x = "3*t"
>>> p.is_constant('x','t')
False
>>> p.is_constant('x','t',x=1)
True
```

is_function (*param*, ***params*)

Parameters

- **param** (*str*) – Name of parameter
- **params** (*dict*) – Dictionary of parameter value overrides.

Returns `True` if specified parameter is a function. `False` otherwise.

This method checks if a given parameter is a function, which usually implies it is dependent on other parameters. For example:

```
>>> p.x = 3
>>> p.is_function('x')
False
>>> p.is_function('x',x='3*t')
True
```

is_resolvable(**args*, ***params*)

Parameters

- **args** (*tuple*) – Sequence of parameter names.
- **params** (*dict*) – Dictionary of parameter value overrides.

Returns `True` if each of the requested parameters can be

successfully evaluated. `False` otherwise.

This method actually goes through the process of evaluating the parameter, so if you need its value, it is probably better to use a try-except block in your code around the usual parameter extraction code.

Example:

```
>>> p.x = 1
>>> p.y = 'x*z'
>>> p.is_resolvable('x')
True
>>> p.is_resolvable('x','y')
False
>>> p.is_resolvable('x','y',z=1)
True
```

classmethod load(*cls*, *filename*, ***kwargs*)

Parameters

- **filename** (*str*) – Filename from which to load parameters.
- **kwargs** (*dict*) – Dictionary of arguments to pass to `Parameters` constructor.

Returns A new `Parameters` instance preloaded with the configuration contained in `filename`.

This is the method you should use to load a saved `Parameters` configuration. For example:

```
>>> p.load('params.py')
```

The file being loaded should be a valid Python file, with one or more of the following variables available in the global namespace:

- `parameters` : a dictionary of parameter values with names as keys.
- `parameters_cache` : a dictionary of boolean values with names as keys (and where True indicates that the parameter should be cached, see `cache()`).
- `parameters_units` : a dictionary of parameter units with names as keys (only necessary to specify units for parameters which do not have a value attached to them, but for which it is useful to have default units)
- `dimension_scalings` : a dictionary of scalings with dimensions as keys (for valid scalings, see `scaling()`).

- `units_custom` : a list of dictionaries which contain the kwargs necessary to construct the custom unit (seel `add_unit()`).

optimise (*param*, **wrt*, ***params*)

This method returns either a function or string depending on whether the input `param` consisted of more than a single parameter. For symbolic expressions, this can greatly speed up parameter retrieval. A similar mechanism is used internally to make parameter lookups fast. Additionally, it can pre-evaluate parameters that do not depend upon a parameter listed in *wrt*, subject to the parameter overrides of *params*. If *wrt* is not provided, only the functionalisation of the string representation of an expression is performed.

Parameters

- **param** – Any parameter specification that is accepted by parameter retrieval.
- **type** – object
- **wrt** (*tuple*) – Parameters for which all dependees should be preserved as variables.
- **params** (*dict*) – Parameter overrides to use for the check as to whether a parameter is dependent on one of the parameters in *wrt*.

Returns A python function that can be passed to a Parameters instance for Parameters retrieval, or a string if the `param` consisted of a single parameter name.

Raises ExpressionOptimisationError

For example:

```
>>> p.optimise('sin(x)*exp(-t)')
< function with arguments x and t >

>>> p.optimise('sin(x)*exp(-t)','t',x=1)
<function with argument t, with x evaluated to 1>
```

plot (**params*, ***ranges*)

Parameters

- **params** (*tuple*) – A sequence of parameter expressions to plot.
- **ranges** (*dict*) – A dictionary of parameter ranges and overrides.

If Matplotlib is installed, this method provides a simple way to debug whether your parameter values are working as expected. The format of the ranges can be anything accepted by `range()`, but must only have one varying independent parameter.

For example:

```
>>> p.plot('x', x="sin(w*t)", w=2, t=(0,'2*3.142/w',50))
```

The above example will generate a sinusoidal curve x vs. t plot at 50 equidistant intervals between 0 and the end of the first period.

range (**args*, ***ranges*)

Parameters

- **args** (*tuple*) – A sequence of parameters (or parameter expressions).
- **ranges** (*dict*) – A dictionary of overrides and range specifications.

Returns A sequence of parameter values if there is a single parameter requested and it is not enclosed in a list, and dictionary of values otherwise.

This method provides a solution to the common problem of iterating over parameter ranges, or investigating how one parameter changes as a function of another. It has a similar syntax for parameter extraction by calling the `Parameter` instance, and indeed provides a superset of the functionality. It is kept separate for peformance considerations.

One can think about the syntax of this method as overriding the parameter value with a sequence of values, rather than a specific value. For example:

```
>>> p << {'y':lamdba x:x**2}
>>> p.range( 'y', x = [0,1,2,3] )
[0,1.,4.,9.]
```

In this simple example, we see that we can iterate over a provided array of values. Arrays may be input as lists or numpy ndarrays; and returned arrays are typically numpy arrays.

The values for parameter overrides can also be provided in a more abstract notation; such that the range will be generated when the function is called. Parameters accepts ranges in the following forms:

- (<*start*>, <*stop*>, <*count*>) : which will generate a linear array from <start> to <stop> with <count> values.
- (<*start*>, <*stop*>, <*count*>, ..., <*sampler*>) : which is as above, but where the <sampler> is expected to generate the array. <sampler> can be a string (either 'linear','log','invlog' for linear, logarithmic, or inverse logarithmic distributions respectively); or a function which takes arguments <start>, <stop>, <count> and any other arguments from "...". Note that when you specify your own function, the <start>, <stop> and <count> variables need not be interpreted as their name suggests.

Example:

```
>>> p.range( 'y', x = (0,10,2) )
[0.,100.]
```

It is also possible to determine multiple parameters at once.

```
>>> p.range( 'x', 'y', x=(0,10,2) )
{'x':[0.,10.], 'y':[0.,100.]}
```

If multiple overrides are provided, they must either be constant or have the same length.

```
>>> p.range('x', x=(0,10,2), z=1 ) # This is OKAY

>>> p.range('x', x=(0,10,2), z=[3,4]) # This is also OKAY

>>> p.range( 'x', x=(0,10,2), z=[1,2,3] ) # This is NOT okay.
```

ranges_iterator(*ranges*, *params={}*, *masks=None*, *function=None*, *function_kwargs={}*, *nprocs=None*, *ranges_eval=None*, *progress=True*)

This method is shorthand for:

```
>>> RangesIterator(parameters=self, ranges=ranges, params=params, masks=masks, function=func
```

The `RangesIterator` object allows you to iterate over nested parameter ranges, which is useful when you want to sweep out a multidimensional parameter space; for example, when you want to make a 2D plot of some function. In some sense, this is a generalisation of the Python `map()` function.

For more information, please refer to the `RangesIterator` documentation.

scaling(**args*, ***kwargs*)

Parameters

- **args** (*tuple of str*) – A (possibly empty) sequence of dimensions to query.

- **kwargs** (*dict of str/Units*) – A (possibly empty) specification of scales for various dimensions.

Returns The scaling associated with dimensions listed in `args`.

It is usually the case in numerical applications that one is only interested in dealing with non-dimensionalised quantities, rather than quantities with units. By default, Parameters instances will return non-dimensional quantities unless negated by the underscore (as described elsewhere). This method sets the reference scales which determine the non-dimensionalisation. By default, the SI fundamental units are used, as defined in `SIUnitDispenser`.

For example:

```
>>> p.scaling(length='m') # This will set the base length unit to 'm'
>>> p( (1,'m') )
1.
>>> p.scaling(time='ns') # This will set base time unit to 'ns'
>>> p( (1,'s') )
1e9
```

Obviously, this does not affect units returned in united form.

```
>>> p._( (1,'s') )
1 s
```

set_bounds (*bounds_dict*, *error=True*, *clip=False*, *inclusive=True*)

Parameters

- **bounds_dict** (*dict*) – A dictionary with parameters as keys with a valid bounding specification (described below).
- **error** (*bool*) – `True` if a parameter found to be outside specified bounds throw an error; or if `clip` is `True`, whether a warning should be generated. `False` otherwise.
- **clip** (*bool*) – `True` if the relevant parameter should be clipped to the nearest bound edge (assumes inclusive is True). `False` otherwise.
- **inclusive** (*bool*) – `True` if the upper and lower bounds should be included in range. `False` otherwise.

This method provides you with greater control than the shorthand methods described in the class documentation. As a reminder the shorthand methods looked like:

```
>>> p['x'] = (0,100)
```

If one of the extremum values is None, it is set to -infinity or +infinity, depending upon whether it is the upper or lower bound. If a disjointed bound is necessary, you can use:

```
>>> p['x'] = [ (None,10), (15,None) ]
```

This method is much more flexible; for example:

```
>>> p.set_bounds( {'x':(0,100)}, error=True, clip=True, inclusive=True )
```

Will warp 'x' to the closer of 0 or 100 if outside of the range [0,100], reporting a warning in the process.

> **Warning:** Using parameter bounds greatly increases the amount of computation done during parameter retrieval. It is recommended that you do not use parameter bounds in contexts which require minimal runtime, such as numerical integration.

show(*self*)

This method simply prints a formatted table of parameters stored in the `Parameters` instance. It shows the parameter and its dependencies, its united value, and its non-dimensionalised value.

To use it, simply run:

```
>>> p.show()
```

unit_add(**args*, ***kwargs*)

Parameters

- **args** (*tuple of mixed type*) – A length 1 sequence of Unit objects, or args to pass to the `Unit` constructor.
- **kwargs** (*dict of mixed type*) – Keyword arguments to pass to the `Unit` constructor.

This method allows you to add custom units to the `UnitDispenser` object which generates `Units` objects on demand. These additional units can override existing units, or add entirely new units. The `args` and `kwargs` passed to this function are essentially passed directly to the `Unit` constructor:

```
>>> Unit(*args, **kwargs)
```

Unless: - `args` contains a single `Unit` instance, it is directly added the the `UnitDispenser` used by `Parameters`. - `args` contains a single `dict` instance, in which case the unit `Unit(**args[0])` is add to the `UnitDispenser`.

For more information regarding the parameters which can passed to the `Unit` constructor, see documentation for `Unit`.

unit_scaling(**params*)

Parameters **params** (*tuple*) – A sequence of params for which to query internal non-dimensionalisation scaling.

Returns A dictionary of unit scalings, or, if `params` is of length 1, a single numeric scaling factor.

This method returns the internal non-dimensionalisation factor used to scale united values. For example, using the default `SIUnitDispenser`:

```
>>> p.unit_scaling('m')
1.0
>>> p.unit_scaling('km')
0.001
>>> p.unit_scaling('eV')
6.241509647120417e+18
```

The results should be interpreted as: non-dim_value = united_value / unit_scaling .

units(**params*)

Parameters

- **params** – A sequence of parameters for which to extract the default units.
- **type** – tuple of str

Returns An dictionary of `Units`, or, if only one param has been

requested and not wrapped in a list, a single `Units` object.

This method returns the default units associated with a particular parameter. For example:

```
>>> p << {'x': (1,'ms'), 'y': (1,'m')}
>>> p.units('x')
ms
>>> p.units(['x'])
{'x': ms}
>>> p.units('x','y')
{'x': ms, 'y': m}
```

class parampy.parameters.**Bounds**

Bases: object

Bounds(param, units, bounds, error=True, clip=False, inclusive=True)

Parameters

- **param** (*str*) – Name of param to which bound applies.
- **units** (str or Units) – Units of parameter.
- **bounds** (*List of bounds.*) – The bound specification. See Parameters.bounds().
- **error** (*bool*) – A boolean flag named error.
- **clip** (*bool*) – A boolean flag named clip.
- **inclusive** (*bool*) – A boolean flag named inclusive.

The Bounds object is used by a Parameters instance to store the properties of a parameter bound. This is the object type returned by Parameters.bounds(). The above parameters can be extracted using attributes (if b is an instance of Bounds):

```
>>> b.param
>>> b.units
>>> ...
```

4.2 The Quantity Class

class parampy.quantities.**Quantity**

Bases: object

Quantity (value,units=None,absolute=False,dispenser=None)

A Quantity object represents a physical quantity; that is, one with both a value and dimensions. It is able to convert between different united representations; and keeps track of units in basic arithmetic.

Parameters

- **value** (*Numeric*) – The value of the physical quantity in units of 'units'. Can be any python object conforming to standard numeric operations.
- **units** (*str or Units*) – A representation of the units of the object. See documentation of 'Units' for more information.
- **absolute** (*bool*) – Whether this quantity represents an absolute quantity (a quantity with an absolute reference scale) or a relative one (such as a temperature delta).
- **dispenser** (*UnitDispenser*) – The unit dispenser object from which unit objects are drawn. If not specified, a new UnitDispenser object is created.

Instantiate a Quantity object: To create a new Quantity object simply pass the value, unit representation and an instance of UnitDispenser to the Quantity constructor. It is important that the units provided are recognised by the UnitDispenser.

```
>>> q = Quantity(1, 'ms', dispenser=SIUnitDispenser())
```

If no dispenser is provided, it internally defaults to the empty UnitDispenser. A subclass named `SIQuantity` is also available which defaults to the SIUnitDispenser; a unit dispenser prepopulated with SI units. In the rest of this documentation we use `SIQuantity` for brevity.

Note: Instantiating a *UnitDispenser* instance will be the slowest part of instantiating a *Quantity*; so in circumstances where performance is required, providing an already instantiated *UnitDispenser* instance is a good idea.

Accessing value and units separately: You can extract the value and units of a Quantity object separately by accessing the 'value' and 'units' attributes of the object.

```
>>> SIQuantity(1,'ms').value
1
>>> SIQuantity(1,'ms').units
ms
```

Unit conversion: You can convert a Quantity object to another Quantity object with different units provided the dimensions agree.

```
>>> SIQuantity(1,'m')('km')
0.001 km
>>> SIQuantity(1,'g/ns')('kg/s')
1000000.0 kg/s
```

You can also do a unit conversion and keep only the value, using the right shift operator:

```
>>> SIQuantity(1,'m') >> 'km'
0.001
```

Representing in standard basis: As a special case of the unit conversion described above, you can represent any Quantity in the basis defined by the UnitDispenser, using the `basis()` method. The SIUnitDispenser's basis is the fundamental SI units.

```
>>> SIQuantity(1,'km').basis()
1000 m
>>> SIQuantity(1,'J').basis()
1.0 m^2*kg/s^2
```

Arithmetic with Quantity objects: Quantity objects support basic arithmetic with the following operations: - Addition and subtraction with another Quantity object of the same dimensions (final units taken from first argument) - Multiplication and division with another Quantity object, or with a scalar numeric value. - Absolute values. - Arbitrary powers

For convenience, Quantity objects will recognise two-tuples as a prototype for a Quantity, allowing for shorthand in numeric operations.

For example:

```
>>> SIQuantity(1,'ms') + (2,'s')
2001.0 ms
>>> (3,'km') - SIQuantity(2,'m')
2.998 km
>>> 2 * SIQuantity(10,'m')
20 m
```

```
>>> SIQuantity(10,'m') * SIQuantity(2,'s')
20 s*m
>>> SIQuantity(10,'m') / SIQuantity(3,'s') # Be careful with integer divison
3 m/s # Answer is wrong because SIQuantity does not change value types.
>> abs(SIQuantity(-5,'m'))
5 m
>> SIQuantity(5,'m*s')**2
25 s^2*m^2
```

Boolean logic with Quantity objects: Quantity objects support the following boolean operators: - Testing for equality and inequality - Testing relative size using less than and greater than

As for arithmetic, two-tuples are automatically converted to Quantity objects for comparison.

```
>>> SIQuantity(2,'m') == (200,'cm')
True
>>> SIQuantity(2,'m') != (200,'cm')
False
>>> SIQuantity(2,'ns') < SIQuantity(1000,'as')
False
>>> SIQuantity(2,'ns') > SIQuantity(1000,'as')
True
```

absolute
A boolean specifying whether this unit requires an absolute reference frame. For example, many temperature scales do.

You can update this value using:

```
>>> quantity.value = <value>
```

basis()

Returns Quantity object with current value expressed in the basis units of the UnitDispenser.

For example, for the `SIUnitDispenser`:

```
>>> SIQuantity(1,'J').basis()
1.0 kg*m^2/s^2
```

dispenser
The *UnitDispenser* instance used to generate units from string representations.

You can update this reference using:

```
>>> quantity.dispenser = <UnitDispenser instance>
```

units
The units of the quantity.

You can update the units (maintaining the current value) using:

```
>>> quantity.units = <Units object or string representation>
```

value
The value (or magnitude) of the quantity in the current units.

You can update this value using:

```
>>> quantity.value = <value>
```

class parampy.definitions.**SIQuantity**
Bases: parampy.quantities.Quantity

A subclass of Quantity which has a fallback default dispenser of an SIUnitDispenser rather than an empty UnitDispenser. See documentation of Quantity for more information.

4.3 The Units Module

class parampy.units.**Unit**
Bases: object

Unit (name,abbr=None,rel=1.0,prefixable=True,plural=None,dimensions={},base_unit=None)

Unit is the fundamental object which represents a single unit in units specification. For example, in "m*s", each of "m" and "s" would have their own associated Unit object.

Parameters

- **name** (*str or list of str*) – A string or list of strings that are used as full names for this unit. If more than one is provided, the first is used as the default representation.
- **abbr** (*str or list of str*) – A string or list of strings that will be used to represent the abbreviated unit. If more than one is provided, the first is used as the default representation.
- **rel** (*float*) – The size of this unit compared to some fixed arbitrary basis.
- **prefixable** (*bool*) – Whether or not the unit can be prefixed (e.g. milli, micro, etc).
- **plural** (*str*) – When written in full, use this string as the plural unit. Note that when this is not provided, where implemented, an 's' is appended to the unit.
- **dimensions** (*dict*) – A dictionary specification of the dimensions of this unit.
- **base_unit** (*Unit*) – A Unit object corresponding to the base unit (if this unit is prefixed), or None otherwise.

To instantiate a Unit object, you can use something like: >>> u = Unit('metre',abbr='m',rel=1.0,dimensions={'length':1})

Note that the keys for the dimensions can be whatsoever you please, and that you can invent your own dimensions. For example, you could create a unit with dimensions of "intelligence". The UnitDispenser instance will pick up this dimension and handle unit conversions/etc, where that makes sense. The values of the dimensions dictionary indicate the power to which the dimension specified in the key should be taken.

abbr
The default abbreviation for this unit.

To set the name(s) for this unit, you can use:

```
>>> u.abbr = "m"
```

Or, if you want to provide multiple names as an alias.

```
>>> u.abbr = ["m","<abbr>"]
```

abbrs
abbrs(self)

Returns The full list or tuple of abbreviations specified for this unit.

dimensions
The dictionary of dimensions specified for this unit.

To set the dimensions for this unit, you can use:

```
>>> u.dimensions = {'length':1}
```

For more about the dimensions format, refer to comments in the general documentation for this class.

name
The default full name for this unit.

To set the name(s) for this unit, you can use:

```
>>> u.name = "meter"
```

Or, if you want to provide multiple names as an alias.

```
>>> u.name = ["meter","metre"]
```

names
The full list or tuple of names specified for this unit.

set_dimensions(***dimensions*)

Parameters **dimensions** (*dict*) – Dictionary of dimensions.

Returns A reference to the unit object.

This is a shorthand convenience method that allows you to construct units in a slightly simpler form:

```
>>> u = u = Unit('metre',abbr='m',rel=1.0).set_dimensions(length=1)
```

class parampy.units.**UnitDispenser**
Bases: object

UnitDispenser()

A `Unit` instance is an object that manages a collection of `Unit` objects.

One can subclass the UnitDispenser object and implement the UnitsDispenser.[init_prefixies,init_units] methods to prepopulate the UnitDispenser object with units. This is done for the SI unit system with the `SIUnitDispenser` class.

Most of the functionality of the UnitDispenser class is clear from its methods. The functionality which is not clear from reading the documentation of the methods is introduced here.

Creating a UnitDispenser instance:

```
>>> ud = UnitDispenser()
```

Generating Units objects: To generate a `Units` object which represents a particular combination of individual `Unit` objects, you can call this class with a string representing the units. For example:

```
>>> ud('m*s')
m*s
```

This is shorthand for calling:

```
>>> Units('m*s',dispenser=ud)
m*s
```

In addition to being shorter, the Units object is cached in the UnitDispenser object for future recall. See the documentation for `Units` for more.

For non-compound units, you can also use:

```
>>> ud.km
```

Which is shorthand for:

```
>>> ud('km')
```

add(*unit*, *check=True*)

Parameters

- **unit** (*Unit*) – The unit to be added to the UnitDispenser.
- **check** (*bool*) – Whether to check that there is a basis unit for every dimension used by the `Unit` object.

Returns A reference to self, for chaining.

This method adds the supplied `Unit` object to the dispenser's catalog of known units. If the unit object has a dimension of type never seen before by the dispenser, and the unit has no other dimensions, it is made the default "basis" for that dimension. If the unit has other dimensions as well, then a warning is printed stating that no basis element is known for that dimension.

To add a unit (or units), you can use:

```
>>> ud.add( Unit('metre',abbr='m',rel=1.0).set_dimensions(length=1,mass=-2) ).add( .... )
```

You can also use the shorthand:

```
>>> ud + Unit('metre',abbr='m',rel=1.0).set_dimensions(length=1,mass=-2) + .....
```

Warning: When adding a unit, the dispenser will replace any existing unit by the same name; and that if Unit.prefixable is True, then it will add the unit as well as all possible prefixed versions of that unit.

add_conversion_map(*unit_from*, *unit_to*, *mapping*, *absolute=True*, *context=None*)

Parameters

- **unit_from** – The units from which to convert.
- **unit_to** – The units to which to convert.
- **mapping** (*callable*) – A callable that takes a single value in units *unit_from* and returns a value with units *unit_to*.
- **absolute** (*bool*) – Whether the map provided is for units with an absolute scale or a relative one (i.e. fahrenheit -> celcius depends on whether stored values are absolute or delta values).
- **context** (*str*) – The context in which to provide this conversion.

This method adds a potentially non-linear mapping between units that can be used to simplify computations in different physical contexts. For example, it may be useful to use store data in 'decibels', and later convert this to a linear value. This can be handled automatically.

Using:

```
>>> ud.add_scaling( 'dB', '', lambda v: 10**(v/10.), absolute=True, context=None )
```

Will allow you to do:

```
>>> SIQuantity( 1, 'dB' )('')
```

add_scaling(*dim_from*, *dim_to*, *scaling*, *context=None*)

Parameters

- **dim_from** (*dict*) – A dictionary of dimensions from which scaling will be provided to *dim_to*.
- **dim_to** (*dict*) – A dictionary of dimensions to which scaling will be provided from *dim_from*.
- **scaling** (*float*) – The scaling to be acquired.
- **context** (*string*) – The context to which this scaling belongs.

This method adds a scaling between units of different dimensions that can be used to simplify computations in different physical contexts. For example, in many condensed matter physics, energies are often discussed as frequencies (and vice versa).

Using:

```
>>> ud.add_scaling( ud.joule.dimensions, ud.hertz.dimensions, 1./2/hbar/pi, context='condens
```

Will allow you to do:

```
>>> SIQuantity( 2, 'J' )('GHz', context='condensed_matter')
```

basis(***kwargs*)

Parameters **kwargs** (*dict*) – A dictionary of `Unit` or `str` objects with dimension names as keys.

This method is used to both extract and set the "basis" units for this dispenser instance. By default, the basis is the set of Unit objects first set with dimensions of {<dimension> : 1}. The only role the basis is plays is to set a standard canonical basis of units in which to express physical quantities.

To examine the current basis, use:

```
>>> ud.basis()
```

To update or modify the basis, use:

```
>>> ud.basis(length='cm', time='ns')
```

For more about how this useful, see the documentation for `Quantity.basis`.

conversion_map(*unit_from*, *unit_to*, *absolute=True*, *context=None*)

Parameters

- **unit_from** – The units from which to convert.
- **unit_to** – The units to which to convert.
- **absolute** (*bool*) – Whether the map found should be for an absolute conversion or a relative one (i.e. fahrenheit -> celcius depends on whether stored values are absolute or delta values).
- **context** (*str*) – The context in which to provide this conversion.

This method returns a function that when called upon the value associated with a quantity of units *unit_from* will return the appropriate value for a new quantity with units *unit_to*. This method only returns such a function for definitions provided to *UnitDispenser.add_conversion_map*.

dimensions

A list of known dimensions. For example:

```
>>> ud.dimensions = ['length', 'mass', 'time']
```

get(*unit*)

Parameters **unit** (*str*) – A string representation of the unit of interest.

Returns `Unit` object associated with a the string representation.

Raises `UnitInvalidError` if no unit can be found that matches.

has(*identifier*)

Parameters **identifier** (*str*) – A string representation of the unit of interest.

Returns `True` if the `UnitDispenser` object recognises the string representation; and `False` otherwise.

For example:

```
>>> ud.has('metre')
```

has_conversion_map(*unit_from*, *unit_to*, *absolute=True*, *context=None*)

Parameters

- **unit_from** – The units from which to convert.
- **unit_to** – The units to which to convert.
- **absolute** (*bool*) – Whether the map found is an an absolute conversion or a relative one (i.e. fahrenheit -> celcius depends on whether stored values are absolute or delta values).
- **context** (*str*) – The context in which to provide this conversion.

This method returns *True* if there exists a conversion between the provided units, as specified using *UnitDispenser.add_conversion_map*.

init_prefixes()
A stub to allow subclasses to populate themselves.

init_units()
A stub to allow subclasses to populate themselves.

is_scalable(*dim_from*, *dim_to*, *context=None*)

Parameters

- **dim_from** (*dict*) – A dictionary of dimensions from which scaling to *dim_to* will be tested.
- **dim_to** (*dict*) – A dictionary of dimensions to which scaling from *dim_from* will be tested.
- **context** (*str*) – The context in which to test this scaling.

This method returns *True* if there is a special scaling specified between the dimensions provided.

list()

Returns A list of strings representing the units recognised by the UnitDispenser.

For example:

```
>>> ud.list()
['m','cm','mm',...]
```

Note that no particular order is guaranteed for this list.

scale(*dim_from*, *dim_to*, *context=None*)

Parameters

- **dim_from** (*dict*) – A dictionary of dimensions from which to scale.
- **dim_to** (*dict*) – A dictionary of dimensions to which to scale.
- **context** (*str*) – The context in which to provide this scaling.

This method returns a float corresponding to a scaling from the dimensions provided, as specified using *UnitDispenser.add_scaling*.

class parampy.units.**Units**

Bases: object

Units(units=None,dispenser=None)

The Units object is the highest level class in python-parameters which deals only with units; and is the class used directly by Quantity. It handles drawing appropriate units from a UnitDispenser, and performing Unit arithmetic. This arithmetic is mostly useful when used behind the scenes by a Quantity instance.

Parameters

- **units** – A representation of the units in some form. See below for details.
- **dispenser** (*UnitDispenser*) – A reference to the UnitDispenser from which to draw units for this object.

Representation of units:

The unit representation passed to the Units constructor can be:

- A Units object (in which case a copy is returned)
- A Unit object (in which case it is upgraded to a Units object)
- A string representing a units object
- A dictionary of Unit-power relationships

A valid string representation is a string which consists of a series of unit tokens (described below) separated by either a "* (for multiplication) or a "/" (for division). Each unit token consists of a unit string representation recognisable by a dispenser (for example: "ms" or "millisecond") followed by an optional power; which is indicated by a caret "^" and a floating point number (including integers). Importantly fractions are not supported in this specification.

For example, here are some valid units string representations:

- "kg*m*s^-2"
- "ms/nm*kg^2"
- "/nm"

A valid dictionary of unit-power relationships is a mapping from Unit object keys (or their string representation) to a numeric power. For example:

```
>>> Units(units={"ms":1,"kg":0.5}, dispenser=ud)
```

Units arithemetic: Units objects support the following arithmetic operations: - Multiplication - Division - Arbitrary powers

For example:

```
>>> ud = SIUnitDispenser()
>>> u1 = Units('J', dispenser=ud)
>>> u2 = Units('nm*s', dispenser=ud)
>>> u1*ud
J*s*nm
>>> u1/ud
J/s/nm
>>> u2**2
s^2*nm^2
```

If you multiply or divide a unit object by an object other than a *Units* instance, a *Quantity* object will be returned. For example:

```
>>> 3 * ud.nm
3 nm

>>> ud.nm**2 / 2
0.5 nm^2
```

Units equality: Units objects can also recognise when they are equal to other Units objects.

For example (with u1 and u2 as defined above):

```
>>> u1 == u2
False
>>> u1 != u2
True
```

There are a few useful methods though.

To determine the numerical scaling factor between two units, you can use: >>> units.scale(other_units) This will raise an exception if the other units have different dimensions.

The numerical scaling factor of this unit relative to the unit basis of the unit dispenser is given by: >>> units.rel

The dimensions of the units object is given by: >>> units.dimensions

The unit equivalent unit in the basis of the UnitDispenser is given by: >>> units.basis

And you can see the dependence of Units on the underlying Unit objects directly by using: >>> units.units

basis()

Returns A new `Units` object representing the units in the `UnitDispenser` basis that correspond to the same dimensions as this object.

For example:

```
>>> Units('g*ns',dispenser=SIUnitDispenser()).basis()
'kg*s'
```

dimensions

A dictionary representing the dimensions of the units described by this object.

```
>>> u.dimensions
{'length': 1, 'time': 1}
```

rel

The relative size of this unit (compared to other units from the same unit dispenser, and with respect to an arbitrary fixed basis).

```
>>> u.rel = 1.2
```

scale(*other*)

Parameters **other** (*Units*) – Unit with which to compare.

Returns A float such that when multiplying the value of a physical quantity with this objects units it would return the value of the same physical quantity represented in 'other's units.

Raises UnitConversionError if "other" does not have the same dimensions as this object.

This method returns the scaling between the units of this object and that of another. This method is used by the `Quantity` object in order to provide unit conversion.

units

A dictionary representation of the units of this object.

```
>>> u.units
{ms: 1, kg: -1}
```

class parampy.definitions.**SIUnitDispenser**

Bases: parampy.units.UnitDispenser

A subclass of `UnitDispenser` which prepopulates the unit dispenser with SI units and some common other units. For a complete list of supported units, please see the "Supported Units" chapter of the python-parameters documentation.

init_prefixes()

This method is called by the `UnitDispenser` constructor, at which point this method populates the dispenser object with the SI prefixes. See the "Supported Units" chapter of the python-parameters documentation for a list of supported prefixes.

init_units()

This method is called by the `UnitDispenser` constructor, at which point this method populates the dispenser object with the SI units (and some other common units). See the "Supported Units" chapter of the python-parameters documentation for a list of supported units.

4.4 Exceptions

exception parampy.errors.**ExpressionOptimisationError**

Bases: parampy.errors.ParametersException

exception parampy.errors.**ParameterBoundsUncheckedWarning**

Bases: exceptions.UserWarning

exception parampy.errors.**ParameterInconsistentWarning**

Bases: exceptions.UserWarning

exception parampy.errors.**ParameterInvalidError**

Bases: parampy.errors.ParametersException

exception parampy.errors.**ParameterNameWarning**

Bases: exceptions.UserWarning

exception parampy.errors.**ParameterNotInvertableError**

Bases: parampy.errors.ParametersException

exception parampy.errors.**ParameterOutsideBoundsError**

Bases: parampy.errors.ParametersException

exception parampy.errors.**ParameterOutsideBoundsWarning**
Bases: exceptions.UserWarning

exception parampy.errors.**ParameterOverSpecifiedError**
Bases: parampy.errors.ParametersException

exception parampy.errors.**ParameterRecursionError**
Bases: parampy.errors.ParametersException

exception parampy.errors.**ParameterValueError**
Bases: parampy.errors.ParametersException

exception parampy.errors.**ParametersException**
Bases: exceptions.Exception

exception parampy.errors.**QuantityCoercionError**
Bases: parampy.errors.ParametersException

exception parampy.errors.**QuantityValueError**
Bases: parampy.errors.ParametersException

exception parampy.errors.**ScalingDimensionInvalidError**
Bases: parampy.errors.ParametersException

exception parampy.errors.**ScalingUnitInvalidError**
Bases: parampy.errors.ParametersException

exception parampy.errors.**ScalingValueError**
Bases: parampy.errors.ParametersException

exception parampy.errors.**SymbolicEvaluationError**
Bases: parampy.errors.ParametersException

exception parampy.errors.**UnitConversionError**
Bases: parampy.errors.ParametersException

exception parampy.errors.**UnitInvalidError**
Bases: parampy.errors.ParametersException

4.5 Useful Utility Classes

class parampy.iteration.**RangesIterator**(*parameters*, *ranges*, *params={}*, *masks=None*, *function=None*, *function_kwargs={}*, *nprocs=None*, *ranges_eval=None*, *progress=True*)
Bases: object

RangesIterator(parameters, ranges, params={}, masks=None, function=None, function_kwargs={}, nprocs=None, ranges_eval=None, progress=True)

RangesIterator is a python iterable object, which allows one to easily iterate over a potentially multidimensional space of parameters. It also has inbuilt multithreading support which is used when a function is supplied in order to fully take advantage of the available computational resources.

Parameters

- **parameters** (*Parameters*) – A reference to a Parameters instance.
- **ranges** (*dict or list of dict*) – The ranges to iterate over.
- **params** (*dict*) – A background context of parameters upon which the ranges should be superimposed.

- **masks** (*list of callable objects*) – A list of boolean functions which indicate when a particular index should be computed.
- **function** (*callable*) – An (optional) function to call for every resulting parameter configuration.
- **function_kwargs** (*dict*) – An (optional) dictionary of kwargs to pass to the above function at every call.
- **nprocs** (*None or int*) – The number of processes to spawn at any one time (for multithreading support).
- **ranges_eval** (*numpy.ndarray*) – An (optional) previously computed ranges_eval to use in this enumeration.
- **progress** (*bool or callable*) – *True* if progress should be shown, and *False* otherwise. This can also be a callable object taking arguments *total*, *completed* and *start_time*, which are the total number of indices to compute, the number completed computations, and the start time computed using *datetime.datetime.now()*.

Constructing a RangesIterator instance: In its simplest form, initialising a `RangesIterator` looks like:

```
>>> RangesIterator(p, ranges)
```

Where *p* is a Parameters instance, and *ranges* is a valid specification of the ranges to be iterated over.

Valid ranges specification: `ranges` can be any valid range specification (as defined in `Parameters.range()`), or any list of valid range specifications. If a list of specifications is provided, then the cartesian product of range specifications is taken. For example:

```
>>> ranges = [{'x':(0,10,11)},{'y':(0,10,11)}]
```

If passed to RangesIterator, this will lead to iteration over the integer values of x in [0,10], and then for each x, to iterate over the integer values of y in [0,10]; forming the cartesian product [0,10]x[0,10].

Remember that multiple parameters can be set at once, so that the following is also valid: >>> ranges = [{'x':(0,10,11),'k':(1,2,11)},{'y':(0,10,11)}]

Iterating over a RangesIterator instance: To iterate over the possible parameter configurations, you use the regular iteration sytax:

```
>>> for result in iterator:
                # Do something here
```

In each iteration of the above loop, result will be a two-tuple of the indices of the current iteration in the cartesian product and one of the following: - If `function()` is not specified, then a dictionary of the parameters (including except where overwritten those in `params`) corresponding to the cartesian indices. - If `function()` is specified, the value of the function evaluated in the parameter context, and with the kwargs arguments in `function_kwargs`.

Specifying a function to be evaluated: If specified, the `function()` value must be a callable type, such as a *function* or *instancemethod* (or any object implementing `__call__()`). The function must take at least the keyword value of `params`. The following are examples of valid specifications:

```
>>> def test(params):
                pass
>>> RangesIterator(..., function=test, ...)

>>> class Test(object):
                def test(self,params):
```

```
                        pass
>>> RangesIterator(..., function=Test().test, ...)

>>> RangesIterator(..., function=lambda params: None, ...)
```

The function can return any type, which will then be returned to you via the iteration process, as specified above.

If `function_kwargs` is specified, then the function provided will be evaluated with the additional kwargs present in that dictionary.

Multithreading: By default, if `function` is provided, `RangesIterator` will spawn up to *N* parallel subprocesses to evaluate the function in different parameter contexts (where *N* is detailed below). If this is undesired, because your function is not threadsafe or for other reasons, you can specify `nprocs` as a *0* or a *1*; in which case multithreading is disabled.

Otherwise, `nprocs` is to be the number of processes to use (with a default value of the number of processors of the machine). If specified as a negative number, then the iteration process will use that many fewer than the total number of processors on your machine.

Masking: If you do not want the parameters or evaluated function at all possible cartesian products of the input ranges, then it is possible to use boolean masking functions. This is useful, for example, when wanting continue a previously started sweep.

Masks should be callable objects with a signature of: `<mask_name>(indices, ranges=None, params={})`

For example:

```
>>> def deny_mask(indices, ranges=None, params={}):
                return False # Deny all indices.
```

At runtime, the current indices in question are passed as indices, along with the range specifications and current parameter context.

`function`
A reference to the callable to be called at each iteration, which is most likely a function or method (or None).

You can change the callable using:

```
>>> iterator.function = <callable>
```

`function_kwargs`
A reference to the dictionary of kwargs to passed to the callable at each iteration.

You can change the kwargs using:

```
>>> iterator.function_kwargs = <kwargs dict>
```

`masks`
The list of mask callables used to filter which indices in the cartesian product of ranges are to be considered.

You can change the masks using:

```
>>> iterator.masks = <list of masks>
```

`nprocs`
The number of processes to use (if positive), the number of CPUs to leave free (if negative), or None to default to using all CPUs.

You can change nprocs using:

```
>>> iterator.nprocs = <integer or None>
```

p

A reference to a Parameters instance.

You can update this reference using:

```
>>> iterator.p = <Parameters Instance>
```

params

A reference to the parameter context used by the iterator.

You can change the parameter context using:

```
>>> iterator.params = <parameters dictionary>
```

Note that this will reset `ranges_eval()`.

progress

A boolean indicating whether progress should be shown, or a callable object which takes arguments:

- *total*: The total number of computations to be performed.
- *completed*: The number of computations completed.
- *start_time*: When the computation started (as a *datetime.datetime* object).

You can change progress using:

```
>>> iterator.progress = <bool or callable>
```

ranges

A reference to the list of ranges stored by the iterator.

You can change the ranges used by the iterator using:

```
>>> iterator.ranges = <Ranges specification>
```

Note that this will reset `ranges_eval()`.

ranges_eval

A previously evaluated ranges_eval from ranges_expand which can used to shortcut the range evaluation phase, and to ensure consistency when the range sampler is stochastic. If not specified when constructing the iterator, it will be generated when access it via this property, and then cached for later use.

You can override ranges_eval using:

```
>>> iterator.ranges_eval = <Previously generated ranges_eval>
```

ranges_expand()

Returns A two-tuple of a structured numpy.ndarray with keys that are the parameters being iterated over and values being their current non-dimensional value, and the list of indices to consider as filtered by masks.

For example:

```
>>> iterator = RangesIterator(parameters=p, ranges=[{'x':(0,1,2)},{'y':(3,4,2)}])
>>> iterator.ranges_expand()
(array([[(0.0, 3.0), (0.0, 4.0)],
        [(1.0, 3.0), (1.0, 4.0)]],
       dtype=[('x', '<f8'), ('y', '<f8')]), [(0, 0), (0, 1), (1, 0), (1, 1)])
```

CHAPTER

FIVE

SUPPORTED UNITS

This section documents the units and prefixes supported by `SIQuantity`, `SIUnitDispenser` or subclasses thereof.

Except where otherwise noted by an * in the unit lists shown below, all units can be prefixed with the following prefixes:

Larger Prefixes	kilo	mega	giga	tera	peta	exa	zepto	yotta
Abbreviation	k	M	G	T	P	E	Z	Y
Scaling (10^x)	3	6	9	12	15	18	21	24

Smaller Prefixes	milli	micro	nano	pico	femto	atto	zepto	yocto
Abbreviation	m	{mu},μ	n	p	f	a	z	y
Scaling (10^x)	-3	-6	-9	-12	-15	-18	-21	-24

The following units are split into groups according to the domains in which they are relevant. Units marked with an asterisk are not prefixable.

5.1 Fundamental Units

Unit Names	Abbreviations	Dimensions
constant	non-dim	-
metre, meter	m	L
second	s	T
gram	g	M
ampere	A	I
kelvin	K	Θ
mole	mol	N
candela	cd	J
dollar*	$	$

5.2 Lengths

Unit Names	Abbreviations	Dimensions
mile	mi	L
yard	yd	L
foot	ft	L
inch	in	L
centimetre, centimeter	cm	L
point	pt	L
angstrom	Å	L
astronomical unit	au	L
lightyear	ly	L

5.3 Volumes

Unit Names	Abbreviations	Dimensions
litre,liter	L	L^3
gallon	gal	L^3
quart	qt	L^3

5.4 Times

Unit Names	Abbreviations	Dimensions
year*	-	T
day*	-	T
hour*	-	T
minute*	min	T
hertz	Hz	T^{-1}

5.5 Force & Pressure

Unit Names	Abbreviations	Dimensions
newton	N	MLT^{-2}
atm	-	$ML^{-1}T^{-2}$
bar	-	$ML^{-1}T^{-2}$
pascal	Pa	$ML^{-1}T^{-2}$
mmHg	mmHg	$ML^{-1}T^{-2}$
psi	-	$ML^{-1}T^{-2}$

5.6 Energy & Power

Unit Names	Abbreviations	Dimensions
joule	J	ML^2T^{-2}
calorie	cal	ML^2T^{-2}
electronvolt	eV	ML^2T^{-2}
watt	W	ML^2T^{-3}

5.7 Electromagnetism

Unit Names	Abbreviations	Dimensions
coulomb	C	IT
farad	F	$T^4I^2L^{-2}M^{-1}$
henry	H	$ML^2T^{-2}I^{-2}$
volt	V	$ML^2I^{-1}T^{-3}$
ohm	Ω	$ML^2I^{-2}T^{-3}$
siemens	mho	$M^{-1}L^{-2}T^3I^2$
tesla	T	$MI^{-1}T^{-2}$
gauss	G	$MI^{-1}T^{-2}$
weber	Wb	$L^2MT^{-2}I^{-1}$

p

Q

R

S

U

V