The DimPy physical quantity package for Python

David Bate

Summer 2008

Terminology

There are three tiers of dimensional quantity in DimPy:

Terminology

There are three tiers of dimensional quantity in DimPy:

- A `Dimension` stores the exponent of each SI unit in a `Quantity` or `Unit`.

Terminology

There are three tiers of dimensional quantity in DimPy:

- A `Dimension` stores the exponent of each SI unit in a `Quantity` or `Unit`.
- A `Unit` contains a `Dimension`, a unit name ("meter") and a unit symbol ("m"). Meter, mile, second are units.

Terminology

There are three tiers of dimensional quantity in DimPy:

- A `Dimension` stores the exponent of each SI unit in a `Quantity` or `Unit`.
- A `Unit` contains a `Dimension`, a unit name ("meter") and a unit symbol ("m"). Meter, mile, second are units.
- A `Quantity` contains a `Unit` and a scalar multiple. Variables such as `my_height` and `mass_of_moon` would be `Quantity` instances.

Creating quantities and new units

In general, users do not need to explicitly construct a `Dimension`, `Unit` or `Quantity`. Instead, new `Units` and `Quantities` can be constructed from existing ones:

Creating quantities and new units

In general, users do not need to explicitly construct a `Dimension`, `Unit` or `Quantity`. Instead, new `Units` and `Quantities` can be constructed from existing ones:

```
>>> my_height = 1.8*meter
```

Creating quantities and new units

In general, users do not need to explicitly construct a `Dimension`, `Unit` or `Quantity`. Instead, new `Units` and `Quantities` can be constructed from existing ones:

```
>>> my_height = 1.8*meter
```

```
>>> mass_of_moon = 7.36e22*kilogram
```

Creating quantities and new units

In general, users do not need to explicitly construct a `Dimension`, `Unit` or `Quantity`. Instead, new `Units` and `Quantities` can be constructed from existing ones:

```
>>> my_height = 1.8*meter
```

```
>>> mass_of_moon = 7.36e22*kilogram
```

DimPy will also check that standard operations are valid:

Creating quantities and new units

In general, users do not need to explicitly construct a `Dimension`, `Unit` or `Quantity`. Instead, new `Units` and `Quantities` can be constructed from existing ones:

```
>>> my_height = 1.8*meter
```

```
>>> mass_of_moon = 7.36e22*kilogram
```

DimPy will also check that standard operations are valid:

```
>>> my_height + mass_of_moon
DimensionMismatchError: Addition,
     dimensions were (m) (kg)
```

Creating quantities and new units

In general, users do not need to explicitly construct a `Dimension`, `Unit` or `Quantity`. Instead, new `Units` and `Quantities` can be constructed from existing ones:

```
>>> my_height = 1.8*meter
```

```
>>> mass_of_moon = 7.36e22*kilogram
```

DimPy will also check that standard operations are valid:

```
>>> my_height + mass_of_moon
DimensionMismatchError: Addition,
     dimensions were (m) (kg)
```

It is also possible to define new units from existing ones:

Creating quantities and new units

In general, users do not need to explicitly construct a `Dimension`, `Unit` or `Quantity`. Instead, new `Units` and `Quantities` can be constructed from existing ones:

```
>>> my_height = 1.8*meter
```

```
>>> mass_of_moon = 7.36e22*kilogram
```

DimPy will also check that standard operations are valid:

```
>>> my_height + mass_of_moon
DimensionMismatchError: Addition,
     dimensions were (m) (kg)
```

It is also possible to define new units from existing ones:

```
>>> Nm = newton*meter; Nm
m N
```

Quantity methods

To view a quantity in different units, the `Quantity.in_unit` method is used, returning a string with the required value. Alternatively the % operator may be used:

Quantity methods

To view a quantity in different units, the `Quantity.in_unit` method is used, returning a string with the required value. Alternatively the % operator may be used:

```
>>> my_height = 1.8*meter
```

Quantity methods

To view a quantity in different units, the `Quantity.in_unit` method is used, returning a string with the required value. Alternatively the % operator may be used:

```
>>> my_height = 1.8*meter

>>> my_height.in_unit(foot)
'5.90551181102 ft'
```

Quantity methods

To view a quantity in different units, the `Quantity.in_unit` method is used, returning a string with the required value. Alternatively the % operator may be used:

```
>>> my_height = 1.8*meter

>>> my_height.in_unit(foot)
'5.90551181102 ft'

>>> my_height % inch
'70.8661417323 inch'
```

Quantity methods

To view a quantity in different units, the `Quantity.in_unit` method is used, returning a string with the required value. Alternatively the % operator may be used:

```
>>> my_height = 1.8*meter

>>> my_height.in_unit(foot)
'5.90551181102 ft'

>>> my_height % inch
'70.8661417323 inch'
```

Quantity functions such as `is_scalar_type`, `have_same_dimensions` and `is_dimensionless` are also available to compare `Quantity` instances.

Non-standard units

Not every quantity one can consider is the product of SI units.

Non-standard units

Not every quantity one can consider is the product of SI units.

Suppose a building contractor must construct "three houses per week", then we need a way to create a unit based upon a string and the ability for these to interact with numbers and quantities.

Non-standard units

Not every quantity one can consider is the product of SI units.

Suppose a building contractor must construct “three houses per week”, then we need a way to create a unit based upon a string and the ability for these to interact with numbers and quantities.

To create a unit from a string the `Flydim` class is used:

Non-standard units

Not every quantity one can consider is the product of SI units.

Suppose a building contractor must construct "three houses per week", then we need a way to create a unit based upon a string and the ability for these to interact with numbers and quantities.

To create a unit from a string the `Flydim` class is used:

```
>>> house = Flydim('house')
>>> flat = Flydim('flat')
```

Non-standard units

Not every quantity one can consider is the product of SI units.

Suppose a building contractor must construct "three houses per week", then we need a way to create a unit based upon a string and the ability for these to interact with numbers and quantities.

To create a unit from a string the `Flydim` class is used:

```
>>> house = Flydim('house')
>>> flat = Flydim('flat')

>>> house*flat
house flat
```

Non-standard units

Not every quantity one can consider is the product of SI units.

Suppose a building contractor must construct "three houses per week", then we need a way to create a unit based upon a string and the ability for these to interact with numbers and quantities.

To create a unit from a string the `Flydim` class is used:

```
>>> house = Flydim('house')
>>> flat = Flydim('flat')

>>> house*flat
house flat

>>> house/flat
house flat^-1
```

A `Flyquant` contains a `Quantity` and a `Flydim`, and are usually constructed from already existing `Flydims` and `Quantities`:

A `Flyquant` contains a `Quantity` and a `Flydim`, and are usually constructed from already existing `Flydims` and `Quantities`:

```
>>> house = Flydim('house')
>>> street = 200*house
>>> length_of_house = 10*meter
```

A `Flyquant` contains a `Quantity` and a `Flydim`, and are usually constructed from already existing `Flydims` and `Quantities`:

```
>>> house = Flydim('house')
>>> street = 200*house
>>> length_of_house = 10*meter

>>> length_of_street = street*(length_of_house/house)
```

A `Flyquant` contains a `Quantity` and a `Flydim`, and are usually constructed from already existing `Flydims` and `Quantities`:

```
>>> house = Flydim('house')
>>> street = 200*house
>>> length_of_house = 10*meter

>>> length_of_street = street*(length_of_house/house)

>>> length_of_street
2000.0 m
```

A `Flyquant` contains a `Quantity` and a `Flydim`, and are usually constructed from already existing `Flydims` and `Quantities`:

```
>>> house = Flydim('house')
>>> street = 200*house
>>> length_of_house = 10*meter

>>> length_of_street = street*(length_of_house/house)

>>> length_of_street
2000.0 m
```

Similar comparison functions exist for `Flyquants`.

Matrices containing physical quantities

If one were to populate a large `numpy.matrix` **A** with `Quantity` objects, operations performed on **A** would be computationally slow.

Matrices containing physical quantities

If one were to populate a large `numpy.matrix` **A** with `Quantity` objects, operations performed on **A** would be computationally slow.

A `QuantMatrix` is a `numpy.matrix` associated with two `Unit` vectors, where the dimension of an entry in the matrix is calculated from the outer product of the two vectors.

Matrices containing physical quantities

If one were to populate a large `numpy.matrix` **A** with `Quantity` objects, operations performed on **A** would be computationally slow.

A `QuantMatrix` is a `numpy.matrix` associated with two `Unit` vectors, where the dimension of an entry in the matrix is calculated from the outer product of the two vectors.

One should view a `QuantMatrix` as follows:

Matrices containing physical quantities

If one were to populate a large `numpy.matrix` **A** with `Quantity` objects, operations performed on **A** would be computationally slow.

A `QuantMatrix` is a `numpy.matrix` associated with two `Unit` vectors, where the dimension of an entry in the matrix is calculated from the outer product of the two vectors.

One should view a `QuantMatrix` as follows:

```
   kg  mol
m 1.0  2.0
s 3.0  4.0
```

Matrices containing physical quantities

If one were to populate a large `numpy.matrix` **A** with `Quantity` objects, operations performed on **A** would be computationally slow.

A `QuantMatrix` is a `numpy.matrix` associated with two `Unit` vectors, where the dimension of an entry in the matrix is calculated from the outer product of the two vectors.

One should view a `QuantMatrix` as follows:

```
   kg  mol
m 1.0  2.0
s 3.0  4.0
```

which represents the matrix:

```
1.0 m kg  2.0 m mol
3.0 s kg  4.0 s mol
```

To create a `QuantMatrix`, the base matrix must be a child of `numpy.ndarray` and the two dimension vectors must be lists containing `Dimension`, `Unit` or `Quantity` types.

To create a `QuantMatrix`, the base matrix must be a child of `numpy.ndarray` and the two dimension vectors must be lists containing `Dimension`, `Unit` or `Quantity` types.

DimPy will then calibrate the base matrix so that the matrix is displayed in SI units (and only `Dimension` types are stored):

To create a `QuantMatrix`, the base matrix must be a child of `numpy.ndarray` and the two dimension vectors must be lists containing `Dimension`, `Unit` or `Quantity` types.

DimPy will then calibrate the base matrix so that the matrix is displayed in SI units (and only `Dimension` types are stored):

```
>>> base_matrix = numpy.array([[1,2],[3,4]])
>>> vertical = [meter, second]
>>> horizontal = [mile, mole]
```

To create a `QuantMatrix`, the base matrix must be a child of `numpy.ndarray` and the two dimension vectors must be lists containing `Dimension`, `Unit` or `Quantity` types.

DimPy will then calibrate the base matrix so that the matrix is displayed in SI units (and only `Dimension` types are stored):

```
>>> base_matrix = numpy.array([[1,2],[3,4]])
>>> vertical = [meter, second]
>>> horizontal = [mile, mole]

>>> A = QuantMatrix(base_matrix, [vertical, horizontal])
      m     mol
m 1609.344  2.0
s 4828.032  4.0
```

To create a `QuantMatrix`, the base matrix must be a child of `numpy.ndarray` and the two dimension vectors must be lists containing `Dimension`, `Unit` or `Quantity` types.

DimPy will then calibrate the base matrix so that the matrix is displayed in SI units (and only `Dimension` types are stored):

```
>>> base_matrix = numpy.array([[1,2],[3,4]])
>>> vertical = [meter, second]
>>> horizontal = [mile, mole]

>>> A = QuantMatrix(base_matrix, [vertical, horizontal])
      m     mol
m 1609.344  2.0
s 4828.032  4.0
```

The base matrix or quantities can be changed after creation using attributes, but DimPy will check that the new values are compatible (i.e. that the size of the new matrix matches that of the old one).

As for a `Quantity`, DimPy will check that (where applicable) arithmetic operations are valid, but the requirements are more complicated.

As for a `Quantity`, DimPy will check that (where applicable) arithmetic operations are valid, but the requirements are more complicated.

Values are read from a `QuantMatrix` like a standard `numpy.ndarray`:

As for a `Quantity`, DimPy will check that (where applicable) arithmetic operations are valid, but the requirements are more complicated.

Values are read from a `QuantMatrix` like a standard `numpy.ndarray`:

```
>>> A[0,0]
1609.344 m^2
```

The shuffle function

`shuffle(qmatrix, shuffle_vector)` does not alter the value of a `QuantMatrix` but may be used to alter the appearance of a `QuantMatrix`.

The shuffle function

`shuffle(qmatrix, shuffle_vector)` does not alter the value of a `QuantMatrix` but may be used to alter the appearance of a `QuantMatrix`.

It multiplies each dimension in the horizontal dimensions by `shuffle_vector` and divides each vertical dimension by `shuffle_vector`:

The shuffle function

`shuffle(qmatrix, shuffle_vector)` does not alter the value of a `QuantMatrix` but may be used to alter the appearance of a `QuantMatrix`.

It multiplies each dimension in the horizontal dimensions by `shuffle_vector` and divides each vertical dimension by `shuffle_vector`:

```
>>> A
  m  mol
m 1   2
s 3   4
```

The shuffle function

`shuffle(qmatrix, shuffle_vector)` does not alter the value of a `QuantMatrix` but may be used to alter the appearance of a `QuantMatrix`.

It multiplies each dimension in the horizontal dimensions by `shuffle_vector` and divides each vertical dimension by `shuffle_vector`:

```
>>> A
  m  mol
m 1   2
s 3   4

>>> shuffle(A, meter/second); A
          m^2 s^-1  m s^-1 mol
    s         1          2
m^-1 s^2      3          4
```

The shuffle function

`shuffle(qmatrix, shuffle_vector)` does not alter the value of a `QuantMatrix` but may be used to alter the appearance of a `QuantMatrix`.

It multiplies each dimension in the horizontal dimensions by `shuffle_vector` and divides each vertical dimension by `shuffle_vector`:

```
>>> A
  m  mol
m 1   2
s 3   4

>>> shuffle(A, meter/second); A
          m^2 s^-1  m s^-1 mol
    s         1          2
m^-1 s^2      3          4
```

`shuffle_vector` may be a `Dimension`, `Unit` or `Quantity`.

Requesting Conversions

DimPy contains an infix parser which can also handle requests involving quantities. This can be accessed using the interactive session or the parse function.

Requesting Conversions

DimPy contains an infix parser which can also handle requests involving quantities. This can be accessed using the interactive session or the parse function.

Given an expression, DimPy will try to calculate its value and return an answer in SI units. A line is printed showing how the request was interpreted and the result:

Requesting Conversions

DimPy contains an infix parser which can also handle requests involving quantities. This can be accessed using the interactive session or the parse function.

Given an expression, DimPy will try to calculate its value and return an answer in SI units. A line is printed showing how the request was interpreted and the result:

```
---> 3 meters/(2 hours)*4 seconds
3*meter/(2*hour)*4*second = 0.00166666666667 m
```

The parser accepts two forms of multiplication and each will give a different interpretation.

The parser accepts two forms of multiplication and each will give a different interpretation.

The $*$ symbol behaves as the standard Python multiplication, so any expression appearing after it will begin on the numerator.

The parser accepts two forms of multiplication and each will give a different interpretation.

The $*$ symbol behaves as the standard Python multiplication, so any expression appearing after it will begin on the numerator.

Alternatively, a space may be inserted which will be interpreted as multiplication with much higher precedence. In general, the natural way to write the sentence dictates which should be used:

The parser accepts two forms of multiplication and each will give a different interpretation.

The * symbol behaves as the standard Python multiplication, so any expression appearing after it will begin on the numerator.

Alternatively, a space may be inserted which will be interpreted as multiplication with much higher precedence. In general, the natural way to write the sentence dictates which should be used:

```
---> 1.0/ten million
1.0/(ten*million) = 1e-07
```

The parser accepts two forms of multiplication and each will give a different interpretation.

The * symbol behaves as the standard Python multiplication, so any expression appearing after it will begin on the numerator.

Alternatively, a space may be inserted which will be interpreted as multiplication with much higher precedence. In general, the natural way to write the sentence dictates which should be used:

```
---> 1.0/ten million
1.0/(ten*million) = 1e-07

---> 1.0/ten*million
1.0/ten*million = 100000.0
```

Prefixes and Suffixes

A *word* is any string of non-whitespace characters, separated by whitespace. The parser will look for several sections in a word:

Prefixes and Suffixes

A *word* is any string of non-whitespace characters, separated by whitespace. The parser will look for several sections in a word:

- A scalar multiple at the start of a word

Prefixes and Suffixes

A *word* is any string of non-whitespace characters, separated by whitespace. The parser will look for several sections in a word:

- A scalar multiple at the start of a word
- It will then look for an SI prefix (e.g. ‘milli’)

Prefixes and Suffixes

A *word* is any string of non-whitespace characters, separated by whitespace. The parser will look for several sections in a word:

- A scalar multiple at the start of a word
- It will then look for an SI prefix (e.g. ‘milli’)
- Next some quantity (‘mile’)

Prefixes and Suffixes

A *word* is any string of non-whitespace characters, separated by whitespace. The parser will look for several sections in a word:

- A scalar multiple at the start of a word
- It will then look for an SI prefix (e.g. ‘milli’)
- Next some quantity (‘mile’)
- Then for an ‘s’, to see if the word is plural.

Prefixes and Suffixes

A *word* is any string of non-whitespace characters, separated by whitespace. The parser will look for several sections in a word:

- A scalar multiple at the start of a word
- It will then look for an SI prefix (e.g. ‘milli’)
- Next some quantity (‘mile’)
- Then for an ‘s’, to see if the word is plural.
- A number may then follow to represent an exponent. This exponent will act on the quantity and prefix, but not the scalar multiple.

Prefixes and Suffixes

A *word* is any string of non-whitespace characters, separated by whitespace. The parser will look for several sections in a word:

- A scalar multiple at the start of a word
- It will then look for an SI prefix (e.g. 'milli')
- Next some quantity ('mile')
- Then for an 's', to see if the word is plural.
- A number may then follow to represent an exponent. This exponent will act on the quantity and prefix, but not the scalar multiple.

Therefore, the most general word is of the form:

Prefixes and Suffixes

A *word* is any string of non-whitespace characters, separated by whitespace. The parser will look for several sections in a word:

- A scalar multiple at the start of a word
- It will then look for an SI prefix (e.g. 'milli')
- Next some quantity ('mile')
- Then for an 's', to see if the word is plural.
- A number may then follow to represent an exponent. This exponent will act on the quantity and prefix, but not the scalar multiple.

Therefore, the most general word is of the form:

```
---> 1e3millimeters2
1*10^3*(0.001*meter)^2 = 0.001 m^2
```

History

When in interactive mode, the `quantity_parser` module stores a history of recent queries. To recall a previous request enter # followed by the corresponding index anywhere in a request:

History

When in interactive mode, the `quantity_parser` module stores a history of recent queries. To recall a previous request enter # followed by the corresponding index anywhere in a request:

```
---> 3 meters in miles
3*meter = 0.00186411357671 * mile
```

History

When in interactive mode, the `quantity_parser` module stores a history of recent queries. To recall a previous request enter # followed by the corresponding index anywhere in a request:

```
---> 3 meters in miles
3*meter = 0.00186411357671 * mile

---> print_history()
0: 3 meters in miles
```

History

When in interactive mode, the `quantity_parser` module stores a history of recent queries. To recall a previous request enter # followed by the corresponding index anywhere in a request:

```
---> 3 meters in miles
3*meter = 0.00186411357671 * mile

---> print_history()
0: 3 meters in miles

---> #0*2
(3*meter)*2 = 6.0 m
```

Another way of storing values in an interative `quantity_parser` session is to define variables. To do this simply write an expression of the form:

Another way of storing values in an interative `quantity_parser` session is to define variables. To do this simply write an expression of the form:

```
---> new_variable = 3 meters
new_variable = 3*meter
```

Another way of storing values in an interative `quantity_parser` session is to define variables. To do this simply write an expression of the form:

```
---> new_variable = 3 meters
new_variable = 3*meter
```

To recall the value, use the variable name as for a regular variable:

Another way of storing values in an interative `quantity_parser` session is to define variables. To do this simply write an expression of the form:

```
---> new_variable = 3 meters
new_variable = 3*meter
```

To recall the value, use the variable name as for a regular variable:

```
---> new_variable*4
new_variable*4 = 12.0 m
```

Live demo