pyveu Documentation

Release 0.1.0

Frank Sauerburger

Sep 09, 2020

First steps

CHAPTER 1

Installation

There are two recommended ways of installing the Python package. Pick the one which is more convenient for you.

1.1 Install with pip

You can install the Python package pyveu and all its dependencies using pip. Run the following command in a terminal.

```
$ pip install pyveu
```

1.2 Install from the repository

As an alternative, it is possible to clone the development repository and install the package via the setup.py script.

```
$ git clone https://gitlab.sauerburger.com/frank/pyveu
$ cd pyveu
$ python setup.py install
```

This method is especially well suited if you want to install a development version on a different branch.

CHAPTER 2

Getting started

When taking measurements, one usually has to keep track of three things: The actual value of the measurement, the unit in which the quantity is measured and an uncertainty which quantifies how precise the measurement is. The package pyveu was created to simplify the work with real-life quantities.

The main class of the package is the *pyveu.Quantity*. This section is devoted to show typical ways in which the *pyveu.Quantity* objects are used.

2.1 Create quantities

There are several ways to create a *pyveu.Quantity*.

2.1.1 String expressions

The easiest is probably to class the constructor with a string expression. The expression can consist of three parts:

1. The first part specifies the value of the quantity. This part is mandatory. The value expression can be any int, float in regular or scientific notation.

2. The second part specifies the uncertainty. This part is optional. If it is specified, it must start with "+-" followed by a positive number.

3. The last portion specifies the unit. This part is optional. If it is omitted, the quantities is a pure scalar quantity without unit. The unit is specified by products and ratios of unit symbols (such as `m` for meter or `s` for seconds). Unit symbols can be raised to a power by suffixing it with `^n` where `n` is the (positive or negative) exponent.

   Parenthesis cannot be used.

   Often, a unit is build by a ratio with a product in the denominator. To avoid cluttering the string with many `/` characters, it is possible to omit the `*` character in the product of the denominator. A space between two unit symbols multiplies the two units and has precedence of the ratio. This means the string `1 / m^2 s` is equivalent to `m^-2 * s^-1`.

The following example creates quantities from valid expressions.

```
>>> from pyveu import Quantity
>>> Quantity("42")
<Quantity: 42>
```

```
>>> Quantity("137 +- 1")
<Quantity: 137 +- 1>
```

```
>>> Quantity("1.6 km")
<Quantity: 1600 m>
```

```
>>> Quantity("-8.22 +- 0.32 m / s")
<Quantity: (-8.22 +- 0.32) m s^(-1)>
```

```
>>> Quantity("3.2e-3 +- 34e-4 K / W")
<Quantity: (0.0032 +- 0.0034) m^(-2) kg^(-1) s^3 K>
```

2.1.2 Programmatically

When quantities are created programmatically, for example when reading a file, it looks quite painful to build a string with value, error, and unit. The constructor of the class *pyveu.Quantity* accepts all these arguments separately. The first three positional arguments are

1. The value of the quantity as an int or float,
2. The error of the quantities as an int or float; or `None`,
3. The unit of the quantities as a unit vector; or `None`.

The first two arguments should be self-explanatory. The third argument, the unit vector, specifies the unit of the quantity. Unit vectors are how units are handled internally. When working with the default SI units, a unit vector is an 8-dimension vector, or simply a list with eight items. The items specify the exponents of the SI base units. It's best to *copy* the unit vector from existing *pyveu.Unit* objects.

```
>>> import pyveu.si as si
>>> myunit = si.kelvin / si.watt
>>> myunit.unit_vector()
array([-2., -1.,  3.,  0.,  1.,  0.,  0.,  0.])
```

The following quantity is equivalent to the one created from the string `3.2e-3 +- 34e-4 K / W`.

```
>>> Quantity(3.2e-3, 34e-4, myunit.unit_vector())
<Quantity: (0.0032 +- 0.0034) m^(-2) kg^(-1) s^3 K>
```

2.1.3 Named quantities

Quantities can be annotated with a `label`, `symbol`, and a `latex` alternative symbol. The names of the quantities are used when the quantity is printed. The information can also be used by a user program or a third-party library. For example, if a named quantity is passed to a plotting library, the label could be used as the axis label.

`label` The label is a human-readable description of the quantity. `Current`, `Fine structure constant`, `Heat conductivity` are examples of good labels.

`symbol` The symbol is the mathematical symbol used for the quantity. `I`, `a` or `p` are examples of good symbols.

`latex` alternative Sometimes, the common mathematical symbol needs latex typesetting. In that case, you can specify the `latex` alternative, which is used instead of the symbol if the output medium is latex document. `\\alpha` or `p_i` are examples of good latex alternative symbols. If latex typesetting is not needed, this should be `None`.

We can name a quantity by passing the names via keyword arguments to the constructor. This work when using expression strings or when creating a unit programmatically.

```
>>> Quantity("42", label="The answer")
<Quantity The answer: 42>
```

```
>>> Quantity("0.00729", label="Fine structure constant",
... symbol="a", latex="\\alpha")
<Quantity Fine structure constant: a = 0.00729>
```

```
>>> Quantity("-8.22 +- 0.32 m / s", symbol="v")
<Quantity: v = (-8.22 +- 0.32) m s^(-1)>
```

2.2 Arithmetics with quantities

Calculating with quantities is as easy as doing it with simple numbers in Python. The *pyveu.Quantity* overload the operators `+`, `-`, `*`, `/` and `**`. Arithmetic operations are carried out for the value of the quantity, for the unit. The uncertainties of the quantities are propagated to the result.

```
>>> duration = Quantity("28 +- 1 min")
>>> distance = Quantity("1000 +- 50 m")
>>> speed = distance / duration
>>> speed
<Quantity: (0.595238 +- 0.0365745) m s^(-1) | depends=[9, 10]>
```

The package also brings common mathematical methods, such as *pyveu.exp()* and *pyveu.log()*. Calling these methods will also propagate the uncertainty.

```
>>> from pyveu import log
>>> log(Quantity("431 +- 13"))
<Quantity: 6.06611 +- 0.0301624 | depends=[12]>
```

The `depnds` part of the printout indicates that the quantity depends on another quantity (here the quantity with id 12). This information is required to keep track of the correlations between quantities. What are correlations between quantities? When you call `Quantity()`, you create a new, statistically independent quantity. The difference between independent and correlated quantities will become apparent with an example.

Assume you have two statically independent measurements of a physical quantity, you should call `Quantity()` twice. The quantity object `a` and `b` are independent.

```
>>> a = Quantity("1 +- 0.1")
>>> b = Quantity("1 +- 0.1")
```

When we add them, we see that the relative uncertainty gets reduced because the up and down fluctuations of both quantities cancel on average.

```
>>> a + b
<Quantity: 2 +- 0.141421 | depends=[14, 15]>
```

On the other hand, we can also create two quantities `c` and `d` whose uncertainties are correlated.

```
>>> parent = Quantity("1 +- 0.1")
>>> c = 1 * parent
>>> d = 1 * parent
```

Adding `c` and `d` is identical to `2 * parent`, so the result should be the same.

```
>>> c + d
<Quantity: 2 +- 0.2 | depends=[17]>
>>> 2 * parent
<Quantity: 2 +- 0.2 | depends=[17]>
```

Notice that when adding the quantities, the error is simply scaled by the factor two. The up and down fluctuations cannot cancel, because they are always in the same direction.

Internally, correlations are stored as dependencies along with the derivates. The derivative tells the system to what extend an uncertainty propagates to a derived quantity. Using the chain rule it is easy to compute the derivative of a new derived quantity. This technique is known as [automatic differentiation](https://en.wikipedia.org/wiki/Automatic_differentiation) or auto-diff.

2.3 Retrieve the result

The properties of a quantity can be access programmatically with the *pyveu.Quantity.value()*, *pyveu.Quantity.error()* and *pyveu.Quantity.unit_vector()* methods.

```
>>> height = Quantity("178 +- 2 cm")
>>> height.value()
1.78
>>> height.error()
0.02
>>> height.unit_vector()
array([1., 0., 0., 0., 0., 0., 0., 0.])
```

By default, these methods return the value in multiples of the base units, here meter and not centimeter. To get the values in a different unit, pass a unit object or a unit expression to the *pyveu.Quantity.value()* and *pyveu.Quantity.error()* methods.

```
>>> height = Quantity("178 +- 2 cm")
>>> height.value("mm")
1780.0
>>> height.error("mm")
20.0
```

Quantities come with the *pyveu.Quantity.str()* method, which generates a string representation with sensible rounding.

```
>>> height = Quantity("178.12345 +- 2.12345 cm")
>>> height.str()
'(1.78 +- 0.02) m'
```

As with *pyveu.Quantity.value()* and *pyveu.Quantity.error()*, *pyveu.Quantity.str()* accepts an argument to print the quantity in another unit. Unlike the other two methods, the unit passed to *pyveu.Quantity.str()* does not need to be a scalar multiple of the quantity. The print out will automatically append any missing arguments.

```
>>> speed = Quantity("10 +- 0.1 m / s")
>>> speed.str("km / hr")
'(36.0 +- 0.4) km / hr'
>>> speed.str("km")  # missing 1/second added automatically
'(0.01000 +- 0.00010) km * 1 / s'
```

The default set of units contains the speed of light, Planck's constant and electron volts. This makes it easy to convert between natural and SI units.

```
>>> electron_mass = Quantity("0.5 MeV / c^2")
>>> electron_mass
<Quantity: 8.91331e-31 kg>
>>> red_h_c = Quantity("1 h c") / (2 * math.pi)  # Note: hbar = 100 * bar
>>> red_h_c.str("MeV fm")
'197.32698045930246 MeV * fm'
```

CHAPTER 3

API reference

3.1 Named

class pyveu.**Named**(*label*, *symbol*, *latex*)

The class Named defines the base class for all classes in this package, which have a label, a mathematical symbol and optionally and alternative latex representation.

The label should be used to describe an object with a verbose, human readable string. In case of a physical quantity, the label can be 'Current', 'Voltage of the photo diode' or similar strings. The label can be retrieved with the label() method.

The symbol should be used to decorate object with a mathematical symbol. If the named object stores time information, an intuitively understandable symbol is *t*. The symbol is used when the named object is printed on the console. The symbol should not use latex syntax.

The latex property stores an alternative representation of the symbol with latex support. Dollar signs should not be included in the string. For example, the latex symbol of the dielectric constant in matter is commonly set to *epsilon_r*.

The base class Named does not define methods to modify the properties. This means, unless a derived class uses the ModifiableNamed mix-in, the name properties are read-only.

label()

Returns the verbose label of the named object. The label is read-only.

latex()

Returns the latex symbol. The latex symbol property is read-only.

lors()

Returns the latex symbol if it is not None. Otherwise the symbol is returned. This method is read-only. The name of the method stands for latex-or-symbol.

symbol()

Returns the non-latex symbol. The symbol property is read-only.

3.2 UnitSystem

class pyveu.**UnitSystem**(*name*, *n_base*, *n_dimensionless=0*)

This class represents a unit system. This means it holds the definition of all base units. The set of base units spans a vector space (multiplication of base units is the vector space addition). All derived units are vectors in the vector space, this means they can be represented as a linear combination of the base vectors.

The class also functions as a factory for units belonging to this unit system. The unit system keeps a registry of all units created within this system. The registry can be used to parse unit strings. Similarly, the unit system creates and registers prefixes which can be prepended to all units.

A unit is considered dimensionless, if it is a linear combination of dimensionless base units. This means there can not be a linear combination of base vectors considered as dimensionless, which is not pure linear combination of dimensionless vector.

base_representation(*unit_vector*, *lors=False*)

Creates a string representation of the given unit vector using a product of powers of the base units.

If the lors arguemnt is set to true, the method uses lors() of the base units. Otherwise symbol() is used. If there are undefined base units at the time of execution, these base units are represented by [base#i] where i is the index of the base unit.

create_base_unit(*index*, *label*, *symbol*, *latex=None*, *register=False*)

Creates a unit for the base unit identified with the given index. The created unit is returned. The created unit is not registered unless the argument 'register' is set to True.

Please note that base units can not be scaled by a factor. Unlike create_unit, this method does not register the unit. Therefore, unregistered base units do not take part in the unit string parsing.

This mechanism is useful if a base unit has a prefix, such as kilogram. The base class is kilogram. However, in order that prefixes work as expected (i.e. not create a double prefix Mkg, Mega-kilogram), one can register a non-base unit gram. Only gram is be considered during string parsing with all possible prefixes (e.g. kg, mg, etc.). For all other base units, one can set 'register' to True, or manually register them with the 'register_unit()' method.

Since the index argument is used as a list index, it is possible to give negative values. This makes adding dimensionless units more convenient. The first dimensionless base unit can be created with index=-1, the second with index=-2, etc.

Creating a base unit is final. Once a base unit has been created, create_base_unit() will raise an exception, if the same index is used again.

create_prefix(*factor*, *label=None*, *symbol=None*, *latex=None*)

Creates and registers a new prefix built from the given factor. The newly created prefix is returned.

If registering this prefix causes ambiguities, an exception is raised. An example of a ambiguity is, if one tries to add the 'm' (Milli) prefix, and there are already units with the symbols 'min' (minutes) and 'in' (inch).

create_unit(*factor*, *unit_vector*, *label=None*, *symbol=None*, *latex=None*)

Creates and registers a new unit constructed from the given unit_vector and the factor. The newly created unit is returned.

Please note that unlike create_base_unit(), all units created with this method are registered.

If registering this unit causes ambiguities, an exception is raised. An example of an ambiguity is adding a unit with symbol 'h' (hour) if Planck's quantum with symbol 'h' is already registered. An example of a more subtle ambiguity is, if one tries to add a unit with the symbol 'min' (minutes) and there is already a unit 'in' (inch) and the prefix 'm' (Milli).

dimensionless(*unit*)
Check whether the unit vector of the given unit is a pure linear combination of dimensionless base units. If so, return True, otherwise False.

parse_unit(*expression*)
Try to parser the given string to construct a unit from the string. The new unit is returned on success. In case of an error, an exception is raised.

While parsing the string, the method searches all combinations of registered prefixes and registered units to identify the individual tokens. The method only tries to parse a single unit, optionally combined with a prefix.

register_prefix(*prefix*, *label*, *symbol*, *latex=None*)
Registers the given prefix. Registering a prefix means, that it is added to internal registry and will be considered when parsing a unit string. This method can be used to register anonymous prefixes.

The same note about ambiguities for create_prefix() applies here.

register_unit(*unit*, *label*, *symbol*, *latex=None*)
Registers the given unit. Registering a unit means, that it is added to internal registry and will be considered when parsing a unit string. This method can be used to register anonymous units. The method returns the created unit.

The same note about ambiguities for create_prefix() applies here.

3.3 Prefix

class pyveu.**Prefix**(*factor*, *label*, *symbol*, *latex*, *unit_system*)
The Prefix class represents a string with which units can be prepended in order to scale them. A popular example are the SI prefixes such as Kilo, Mega or Giga. A prefix can be created via a unit system. The unit system adds all prefixes to its internal registry. Registered prefixes are considered when parsing unit strings.

The prefix class inherits the label, symbol and latex property of the Named class. Furthermore, the class inherits the properties of the SystemAffiliated class and is therefore tied to a particular unit system.

In addition to the inherited properties, this class stores a factor which is used to scale a unit. For example, the factor of the prefix Kilo is 1000.

Multiplications and divisions are overloaded for units. Multiplications and divisions of a prefix and a number return a number in most of the cases. The only exception is the case when prefix is multiplied by a number from the left, e.g., 10 * kilo. In that case, the result is an anonymous prefix. Its history is a product with the two factors 10 and kilo. These derived prefixes are not automatically added to the registry. If you want to add an anonymous prefix to the registry, use the register_prefix() method of the unit system.

factor()
Returns the factor of the prefix. This method is read-only.

history_str(*latex=False*)
Returns a string representation of the history. If the optional parameter latex=True, a latex version of the string is created. If the history is None, returns None. This method completely includes scalar factors in the history.

3.4 Unit

class pyveu.**Unit**(*factor*, *unit_vector*, *label*, *symbol*, *latex*, *unit_system*)
The unit class represents physical units, such as Ampere or Newton. A unit is created by a unit system. A unit

is permanently tied to the creating unit system. It is not necessary to create units with all possible prefixes. Prefixes are automatically handled once they are registered with a unit system.

A unit inherits all the properties from Named and SystemAffiliated. Additionally, a factor and a unit vector is stored. The unit vector stores the exponents of the base units. Assume a unit system with three base units A, B and C. A unit with a unit vector of [0, 2, 1] corresponds to A^0 * B^2 * C^1. The factor can be used to generated arbitrarily scaled derived units. For example, if you set the factor to 60, the unit represent minutes, if it has the same unit vector as seconds. Please note that it is not necessary to register units with prefixes. Prefixes are handles by the unit system.

Multiplications, divisions and powers are overloaded for units. These operations create a new unit object. The resulting objects are anonymous, i.e. their label and symbol properties are None. Furthermore, these derived units are not automatically added to the registry. If you want to add a anonymous unit to the registry, use the register_unit() method of the unit system.

The properties of a unit can not be changed.

base_representation(*latex=False*, *suppress_factor=False*)
Returns the unit using only base units. If the optional argument latex is True, this method returns a latex version. By default the returned string includes the factor of the unit. If suppress_factor is True, the returned string excludes the factor.

static create_with_history(*factor*, *unit_vector*, *unit_system*)
For internal usage only.

Alternative method to create a unit. Instead of assembling a new Unit from scratch, it is created by multiplying, dividing and exponentiating base units. The return value is an anonymous unit with a minimal history of base units.

If the final unit is dimensionless (i.e. unit_vector = [0, 0, . . .]), the factor is multiplied by the first base unit to the 0-th power.

dimensionless()
Consults the unit system to see, whether this unit is dimensionless, and turns the outcome. If this is true, the unit can be used in mathematical functions such as Sine or the exponential function.

factor()
Returns the factor of the units. This method provides read-only access to the factor.

history_str(*latex=False*)
Returns a string representation of the history. If the optional parameter latex=True, a latex version of the string is created. If the history is None, returns None. This method completely ignores the factor of the unit.

str(*latex=False*)
If the unit is unnamed returns the factor and the history of the unit. If the unit is named, returns the symbol. If the optional parameter latex=True, latex version of the string is created.

unit_vector()
Returns the unit vector of the unit. The stores the exponents of the base units. The unit (neglecting the factor of the unit) is the product of all base units raised to the powers stored in the unit vector. The i-th value in the unit vector specifies the power of the i-th base unit.

Unit vectors are stored as numpy arrays. This method returns a copy of the numpy array.

This method provides read-only access.

3.5 Quantity

class pyveu.**Quantity**(*value*, *error=None*, *unit=None*, *label=None*, *symbol=None*, *latex=None*, *unit_system=None*)

The Quantity stores a single value annotated with an uncertainty and a unit. The Quantity class is the working horse of the pyveu package.

Quantity objects support arithmetic operations with other Quantities, Units and prefixes. Each operation generates a new object. Quantities keep track of its dependencies and how it has been constructed. This information is used to keep track of correlations between quantities and to propagate uncertainties.

Arithmetic operations involving quantities consider the units of all participants. This means, that for example an addition of two quantities is only possible, if they have the same unit. Multiplications yield a new quantity object with the product of both units.

The error propagation is calculated when the arithmetic operation is executed. Modifying one of the participating quantities does not modify the resulting quantity.

Internally, quantity objects store the value in base units. The unit is stored as a unit vector which specifies the exponents of the base units. In order to propagates the uncertainty each quantity needs to keep a list of all quantities it was initially derived from. Additionally, quantities store the uncertainties of these initial quantities and the partial derivatives of the quantity with respect to the initial quantities.

Initial quantities are creates with the Quantity() constructor. All quantities creates in this way are considered to be statistically independent. Quantities created by arithmetic operations are called dependent quantities, since their uncertainty depends on the initial independent quantities. Dependent quantities store the uncertainty of all the independent quantities it depends on, and the partial derivatives with respect to all independent quantities it depends on. The id() method is used to identify independent quantities.

In-place arithmetic operations generate a new object, such that id() returns a different value. The returned object is a dependent quantity. This distinction is important in the following example >>> a = Quantity("10 += 1") >>> b = a + 3 # dependent quantity >>> a *= 2 # A has a different id() >>> c = a + 6

Quantity b depends on quantity a. However, when the quantity a is multiplied in-place, a new quantity a is created, which depends on the initial quantity a. The third quantity c, which is derived from the new a, also becomes a quantity depending on the initial quantity a. When one combines b and c in an arithmetic operations, the framework knows that the both variables depend on the same initial quantity. >>> 2 * a - c 0 +- 0

If the values had been modified in-place, leaving the id() unchanged, the resulting object would have an ambiguous value of the initial uncertainty of a.

Quantity objects inherit all the properties from Named and SystemAffiliatedMixin. This means, Quantity object can be annotated with a label, symbol and latex alternative symbol. It is possible to modify the names using the label(), symbol() and latex() method. Furthermore, Quantity objects are tied to a unit system.

The constructor has an optional argument unit_system. If it is omitted or set to None, the new Quantity object is tied to the unit system returned by pyveu.get_default_unit_system().

Various methods accept the 'to' argument. This argument changes the unit on which the return value is based. For example, consider a quantity for the speed of a tennis ball. >>> v = Quantity("60 m/s")

By default, the value() method returns the value in base units. >>> v.value() 60

If the 'to' argument is passed to the value method, one can retrieve the value in different units. >>> v.value(to="km/h") 216.0

dimensionless()

Check whether the unit vector of this quantity is dimensionless(). Dimensionless quantities can be used in mathematical function such as Sine of the exponential function.

error(*to=None*)

Returns the uncertainty of the quantity object. If the optional argument 'to' is omitted, the returned error is in base units. If a string or a unit object is given, the unit is returned in this unit. If the given unit is not a scalar multiple of this quantity, an exception is raised.

The return value is the square root of variance().

static parse(*expression*, *unit_system=None*)

Parses the expression and returns the triple: value, error and unit vector. The optional parameter determines the unit system. If it is omitted (or set to None), the default unit system is used.

This is a static method. It should be called using the class. >>> Quantity.parse("125.09 +- 0.24 GeV") (125.09, 0.24, np.array(...))

The syntax of the expression is intended to follow mathematical intuition. An expression consists of three parts. The first part specifies the value of a quantity. This part is mandatory. The value must be specified as an integer or a float. Syntax of the value is similar to the float syntax in python.

The second part is optional and specifies the uncertainty of the value. If this part is included it must start with the string '+-' followed by another integer of float. The error must be positive.

The final part specifies the unit. The method returns a unit vector of (0, 0, ..., 0) if this part is omitted. The unit part consists of an arbitrary number of unit specifiers. A unit specifier is a string consisting a registered unit and optionally a registered prefix. unit specifies can be raised to a power by appending '^' and the exponent. The exponent must be an integer or a float. If unit specifies are separated by spaces, they are multiplied.

Unlike the regular mathematical notation, a slash '/' begins a denominator sequence. All space-separated unit specifiers following the slash are part of the denominator. Another slash '/' continues the denominator. To switch back to the numerator, use the star '*'. The following examples illustrate the denominator/numerator convention. The right hand side in the following examples follow the regular mathematical conventions.

a / b c = a / (b * c) a / b / c = a / (b * c) a / b * c = (a * c) / b a c / b = (a * c) / b a * c / b = (a * c) / b

To summarize: Everything to the right of a slash '/' before a star '*' is in the denominator, independently of the number of slashes. Everything to the right of a star '/' before a slash '/' is in the numerator, independently of the number of stars.

Full expression strings look like this.

"10.1 +- 0.3 mm / s" "210 +- 2 m s^-1" "3e8 kg / m^2" "-42" "1e-3 +- 43e-5" "312 +- 21.1 kg m / s^2"

Please note that the use of parentheses is not possible.

static parse_unit(*unit_part*, *unit_system=None*)

Parse the unit part of Quantity expression. See Quantity.parse() for more information about the syntax. The method returns a unit object. The history of the unit object reflects the structure of given string.

If the optional argument unit_system is omitted, the default unit system is used.

qid()

Returns an identifier of the quantity. The identifier is unique for the lifetime of the python process. The identifier is an integer. The qid is used to track the dependencies between quantities.

round(*to=None*, *significant_digits=1.2*, *exponent=None*)

Returns the triple: rounded value as string, rounded error as string, common exponent as integer. This information can be used to construct string representations of the quantities.

The value string is rounded to the last significant digit of the rounded error. The optional keywords significant_digits determines how the error should be rounded. If significant_digits is set to an integer, this

corresponds to the number of significant digits of the error. If significant_digits is a float, the error is rounded to ceil(significant_digits) or floor(significant_digits), depending on the value of the error. For example, if significant_digits is 1.2, errors between 0.10 and 0.20 are rounded to two significant_digits, errors between 0.2 and 1.0 are rounded to one significant digit. The border between these two ranges is determined by the decimal part of significant_digits.

Both, the value and the error, need to be multiplied with 10^(common_exponent). This means the returned information can be used to construct a sting of the form

(rounded_value +- rounded_error) * 10^(common_exponent).

If the optional argument 'to' is omitted, the returned error is in base units. If a string or a unit object is given, the unit is returned in this unit. If the given unit is not a scalar multiple of this quantity, an exception is raised.

If the optional argument 'exponent' is given, the common exponent is fixed to the given value.

str(*to=1*, *significant_digits=1.2*)
Returns a string containing the value, error and the unit of the quantity. The error is rounded to 1 significant digit (two between 1.0 and 2.0), the value is rounded to the last significant digit of the error. By default, the value and error are returned in base units. If the optional argument 'to' is given, it is used to determine the desired unit. 'to' can be a string or a unit object. If the given unit is not a scalar multiple of this quantity, all the remaining terms are added to the unit string. Any factor included in 'to' is ignored.

See round() for more information about significant_digits.

unit_vector()
Returns the unit vector.

value(*to=None*)
Returns the value of the quantity object. If the optional argument 'to' is omitted, the returned value is in base units. If a string or a unit object is given, the unit is returned in this unit. If the given unit is not a scalar multiple of this quantity, an exception is raised.

variance(*to=None*)
Returns the variance of the quantity object. If the optional argument 'to' is omitted, the returned error is in base units. If a string or a unit object is given, the unit is returned in this unit. If the given unit is not a scalar multiple of this quantity, an exception is raised.

3.6 Mathematical functions

pyveu.**exp**(*quantity*)
Calculates the exponential of the given quantity and propagates the uncertainty. The quantity has to be dimensionless.

pyveu.**log**(*quantity*)
Calculates the natural logarithm of the given quantity and propagates the uncertainty. The quantity has to be dimensionless.

pyveu.**log2**(*quantity*)
Calculates the logarithm to the base 2 of the given quantity and propagates the uncertainty. The quantity has to be dimensionless.

pyveu.**log10**(*quantity*)
Calculates the logarithm to the base 10 of the given quantity and propagates the uncertainty. The quantity has to be dimensionless.

pyveu.**pow**(*base*, *exponent*)
Calculates the power the given quantity and propagates the uncertainty. The exponent has to be dimensionless. See Quantity's power operator.

pyveu.**sqrt**(*quantity*)
Calculates the square root of the given quantity and propagates the uncertainty. The quantity has to be dimensionless.

pyveu.**sin**(*quantity*)
Calculates the sine of the given quantity and propagates the uncertainty. The quantity has to be dimensionless.

pyveu.**cos**(*quantity*)
Calculates the cosine of the given quantity and propagates the uncertainty. The quantity has to be dimensionless.

pyveu.**tan**(*quantity*)
Calculates the tangent of the given quantity and propagates the uncertainty. The quantity has to be dimensionless.

pyveu.**sinh**(*quantity*)
Calculates the hyperblic sine of the given quantity and propagates the uncertainty. The quantity has to be dimensionless.

pyveu.**cosh**(*quantity*)
Calculates the hyperbolic cosine of the given quantity and propagates the uncertainty. The quantity has to be dimensionless.

pyveu.**tanh**(*quantity*)
Calculates the hyperbolic tangent of the given quantity and propagates the uncertainty. The quantity has to be dimensionless.

pyveu.**asin**(*quantity*)
Calculates the inverse sine of the given quantity and propagates the uncertainty. The quantity has to be dimensionless.

pyveu.**acos**(*quantity*)
Calculates the inverse cosine of the given quantity and propagates the uncertainty. The quantity has to be dimensionless.

pyveu.**atan**(*quantity*)
Calculates the inverse tangent of the given quantity and propagates the uncertainty. The quantity has to be dimensionless.

pyveu.**asinh**(*quantity*)
Calculates the inverse hyperbolic sine of the given quantity and propagates the uncertainty. The quantity has to be dimensionless.

pyveu.**acosh**(*quantity*)
Calculates the inverse hyperbolic cosine of the given quantity and propagates the uncertainty. The quantity has to be dimensionless.

pyveu.**atanh**(*quantity*)
Calculates the inverse hyperbolic tangent of the given quantity and propagates the uncertainty. The quantity has to be dimensionless.

3.7 Exceptions

class pyveu.**PyVeuException**
Base class from which all package-specific exceptions will be derived.

class pyveu.**DifferentUnitSystem**
This exception will be raises if an arithmetic operation is requested between two objects tied to different unit systems.

class pyveu.**UncertaintyIllDefined**
Raises when the error propagation cannot be performed for the given values.

class pyveu.**BaseUnitExists**
This exception will be raises if one tries to overwrite a base unit.

class pyveu.**SymbolCollision**
This exception is raised when one tried to add a unit/prefix which causes ambiguities in the unit system.

class pyveu.**UnitNotFound**
This exception is raised when parse_unit() is not able to find a unit (and prefix) to match the given expression.

The python package pyveu (Value Error Unit) handles real-life experimental data which includes uncertainties and physical units. The package implements arithmetic operations and many mathematical functions for physical quantities. Gaussian error propagation is used to calculate the uncertainty of derived quantities.

The package is built with the day-to-day requirements of people working a laboratory kept in mind. The package offers an imperative programming style, which means that the operations are evaluated when they are typed interactively in python, giving researchers the freedom and flexibility they need.

CHAPTER 4

Quickstart

Install the package using pip

```
$ pip install pyveu
```

The working horse of the package is the *pyveu.Quantity* class. It can be used to convert units, for example, it can convert meter per second into kilometer per hour.

```
>>> from pyveu import Quantity
>>> speed = Quantity("32 +- 3 m / s")
>>> speed.str("km / hr")
'(115 +- 11) km / hr'
```

Quantities from a measurement usually come with a measurement uncertainty. The class *pyveu.Quantity* propagates the uncertainty automatically.

```
>>> time = Quantity("3.23 +- 0.1 min")
>>> distance = speed * time
>>> distance.str("km")
'(6.2 +- 0.6) km'
```

CHAPTER 5

Links

- GitLab Repository
- Documentation
- pyveu on PyPi

Index

T

U

V