quantity Documentation

Release 0.11.4

Michael Amrhein

Jan 20, 2023

CONTENTS

This package provides classes and functions for unit-safe computations with quantities (including money).

Contents:

CHAPTER

ONE

INTRODUCTION

1.1 What is a Quantity?

"The value of a quantity is generally expressed as the product of a number and a unit. The unit is simply a particular example of the quantity concerned which is used as a reference, and the number is the ratio of the value of the quantity to the unit." (Bureau International des Poids et Mesures: The International System of Units, 8th edition, 2006)

Basic types of quantities are defined "by convention", they do not depend on other types of quantities, for example Length, Mass or Duration.

Derived types of quantities, on the opposite, are defined as products of other types of quantities raised by some exponent.

Examples:

- Volume = Length ** 3
- Velocity = Length ** 1 * Duration ** -1
- Acceleration = Length ** 1 * Duration ** -2
- Force = Mass ** 1 * Acceleration ** 1

Each type of quantity may have one special unit which is used as a reference for the definition of all other units, for example Metre, Kilogram and Second. The other units are then defined by their relation to the reference unit.

If a type of quantity is derived from types of quantities that all have a reference unit, then the reference unit of that type is defined by a formula that follows the formula defining the type of quantity.

Examples:

- Velocity -> Metre per Second = Metre ** 1 * Second ** -1
- Acceleration -> Metre per Second squared = Metre ** 1 * Second ** -2
- Force -> Newton = Kilogram ** 1 * Metre ** 1 * Second ** -2

1.1.1 "Systems of Measure"

There may be different systems which define quantities, their units and the relations between these units in a different way.

This is not directly supported by this module. For each type of quantity there can be only no or exactly one reference unit. But, if you have units from different systems for the same type of quantity, you can define these units and provide mechanisms to convert between them (see *Converters*).

1.2 Main Package Contents

1.2.1 The Basics: Quantity and Unit

The essential functionality of the package `quantity` is provided by the two classes `Quantity` and `Unit`.

A **basic** type of quantity is declared just by sub-classing `Quantity`. A **derived** type of quantity is declared by giving a definition based on more basic types of quantities. For details see *Defining a quantity class*.

1.2.2 Utility Functions and Classes

In addition, the package `quantity` provides some utility functions (see *Functions*), base classes for defining unit converters (see *Converter classes*) and some specific exceptions (see *Exceptions*).

1.2.3 Commonly Used Quantities

The module `quantity.predefined` provides definitions of commonly used quantities and units.

1.2.4 A Special Quantity: Money

Money is a special type of quantity. Its unit type is known as currency.

Money differs from physical quantities mainly in two aspects:

- Money amounts are discrete. For each currency there is a smallest fraction that can not be split further.
- The relation between different currencies is not fixed, instead, it varies over time.

The sub-package `quantity.money` provides classes and functions dealing with these specifics.

CHAPTER

TWO

QUANTITY

Unit-safe computations with quantities.

2.1 Usage

2.1.1 Defining a quantity class

A **basic** type of quantity is declared just by sub-classing *Quantity*:

```
>>> class Length(Quantity):
...     pass
...
```

But, as long as there is no unit defined for that class, you can not create any instance for the new quantity class:

```
>>> l = Length(1)
Traceback (most recent call last):
ValueError: A unit must be given.
```

If there is a reference unit, the simplest way to define it is giving a name and a symbol for it as keywords. The meta-class of *Quantity* will then create a unit automatically:

```
>>> class Mass(Quantity,
...            ref_unit_name='Kilogram',
...            ref_unit_symbol='kg'):
...     pass
...
>>> Mass.ref_unit
Unit('kg')
>>> class Length(Quantity,
...              ref_unit_name='Metre',
...              ref_unit_symbol='m'):
...     pass
...
>>> Length.ref_unit
Unit('m')
```

Now, this unit can be given to create a quantity:

```
>>> METRE = Length.ref_unit
>>> print(Length(15, METRE))
15 m
```

If no unit is given, the reference unit is used:

```
>>> print(Length(15))
15 m
```

Other units can be derived from the reference unit (or another unit), giving a definition by multiplying a scaling factor with that unit:

```
>>> a_thousandth = Decimal("0.001")
>>> KILOGRAM = Mass.ref_unit
>>> GRAM = Mass.new_unit('g', 'Gram', a_thousandth * KILOGRAM)
>>> MILLIMETRE = Length.new_unit('mm', 'Millimetre', a_thousandth * METRE)
>>> MILLIMETRE
Unit('mm')
>>> KILOMETRE = Length.new_unit('km', 'Kilometre', 1000 * METRE)
>>> KILOMETRE
Unit('km')
>>> CENTIMETRE = Length.new_unit('cm', 'Centimetre', 10 * MILLIMETRE)
>>> CENTIMETRE
Unit('cm')
```

Instead of a number a SI prefix can be used as scaling factor. SI prefixes are provided in a sub-module:

```
>>> from quantity.si_prefixes import *
>>> NANO.abbr, NANO.name, NANO.factor
('n', 'Nano', Decimal('0.000000001'))
```

```
>>> NANOMETRE = Length.new_unit('nm', 'Nanometre', NANO * METRE)
>>> NANOMETRE
Unit('nm')
```

Using one unit as a reference and defining all other units by giving a scaling factor is only possible if the units have the same scale. Otherwise, units can just be instantiated without giving a definition:

```
>>> class Temperature(Quantity):
...     pass
...
>>> CELSIUS = Temperature.new_unit('°C', 'Degree Celsius')
>>> FAHRENHEIT = Temperature.new_unit('°F', 'Degree Fahrenheit')
>>> KELVIN = Temperature.new_unit('K', 'Kelvin')
```

Derived types of quantities are declared by giving a definition based on more basic types of quantities:

```
>>> class Volume(Quantity,
...              define_as=Length ** 3,
...              ref_unit_name='Cubic Metre'):
...     pass
...
>>> class Duration(Quantity,
```

(continues on next page)

(continued from previous page)

```
...                 ref_unit_name='Second',
...                 ref_unit_symbol='s'):
...     pass
...
>>> class Velocity(Quantity,
...                 define_as=Length / Duration,
...                 ref_unit_name='Metre per Second'):
...     pass
...
```

If no symbol for the reference unit is given with the class declaration, a symbol is generated from the definition, as long as all types of quantities in that definition have a reference unit.

```
>>> Volume.ref_unit.symbol
'm³'
>>> Velocity.ref_unit.symbol
'm/s'
```

Other units have to be defined explicitly. This can be done either as shown above or by deriving them from units of the base quantities:

```
>>> CUBIC_CENTIMETRE = Volume.derive_unit_from(CENTIMETRE,
...                                            name='Cubic Centimetre')
>>> CUBIC_CENTIMETRE
Unit('cm³')
>>> HOUR = Duration.new_unit('h', 'Hour', 3600 * Duration.ref_unit)
>>> KILOMETRE_PER_HOUR = Velocity.derive_unit_from(KILOMETRE, HOUR)
>>> KILOMETRE_PER_HOUR
Unit('km/h')
```

In order to define a **quantized** quantity, the smallest possible fraction (in terms of the reference unit) can be given as *quantum*:

```
>>> class DataVolume(Quantity,
...                  ref_unit_name='Byte',
...                  ref_unit_symbol='B',
...                  quantum=Decimal('0.125')):
...     pass
...
>>> DataVolume.quantum
Decimal('0.125')
```

The method *quantum* can then be used to retrieve the smallest amount for a unit:

```
>>> BYTE = DataVolume.ref_unit
>>> BYTE.quantum
Decimal('0.125')
>>> KILOBYTE = DataVolume.new_unit('kB', 'Kilobyte', KILO * BYTE)
>>> KILOBYTE.quantum
Decimal('0.000125')
```

2.1.2 Instantiating quantities

The simplest way to create an instance of a *Quantity* subclass is to call the class giving an amount and a unit. If the unit is omitted, the quantity's reference unit is used (if one is defined):

```
>>> Length(15, MILLIMETRE)
Length(Decimal(15), Unit('mm'))
>>> Length(15)
Length(Decimal(15))
```

Alternatively, an amount and a unit can be multiplied:

```
>>> 17.5 * KILOMETRE
Length(Decimal('17.5'), Unit('km'))
```

Also, it's possible to create a *Quantity* instance from a string representation:

```
>>> Length('17.5 km')
Length(Decimal('17.5'), Unit('km'))
```

If a unit is given in addition, the resulting quantity is converted accordingly:

```
>>> Length('17 m', KILOMETRE)
Length(Decimal('0.017'), Unit('km'))
```

Instead of calling a subclass, the class *Quantity* can be used as a factory function ...:

```
>>> Quantity(15, MILLIMETRE)
Length(Decimal(15), Unit('mm'))
>>> Quantity('17.5 km')
Length(Decimal('17.5'), Unit('km'))
```

... as long as a unit is given:

```
>>> Quantity(17.5)
Traceback (most recent call last):
QuantityError: A unit must be given.
```

If the *Quantity* subclass defines a *quantum*, the amount of each instance is automatically rounded to this quantum:

```
>>> DataVolume('1/7', KILOBYTE)
DataVolume(Decimal('0.142875'), Unit('kB'))
```

2.1.3 Converting between units

A quantity can be converted to a quantity using a different unit by calling the method *Quantity.convert()*:

```
>>> l5cm = Length(Decimal(5), CENTIMETRE)
>>> l5cm.convert(MILLIMETRE)
Length(Decimal(50), Unit('mm'))
>>> l5cm.convert(KILOMETRE)
Length(Decimal('0.00005'), Unit('km'))
```

These kinds of conversion are automatically enabled for types of quantities with reference units. For other types of quantities there is no default way of converting between units.

```
>>> t27c = Temperature(Decimal(27), CELSIUS)
>>> t27c.convert(FAHRENHEIT)
Traceback (most recent call last):
UnitConversionError: Can't convert '°C' to '°F'.
```

Converters

For types of quantities that do not have a reference unit, one or more callables can be registered as converters:

```
>>> def fahrenheit2celsius(qty, to_unit):
...     if qty.unit is FAHRENHEIT and to_unit is CELSIUS:
...         return (qty.amount - 32) / Decimal('1.8')
...     return None
...
>>> def celsius2fahrenheit(qty, to_unit):
...     if qty.unit is CELSIUS and to_unit is FAHRENHEIT:
...         return qty.amount * Decimal('1.8') + 32
...     return None
...
>>> Temperature.register_converter(fahrenheit2celsius)
>>> Temperature.register_converter(celsius2fahrenheit)
>>> assert list(Temperature.registered_converters()) == \
...     [celsius2fahrenheit, fahrenheit2celsius]
...
```

For the signature of the callables used as converters see *Converter*.

```
>>> t27c.convert(FAHRENHEIT)
Temperature(Decimal('80.6'), Unit('°F'))
>>> t27c.convert(FAHRENHEIT).convert(CELSIUS)
Temperature(Decimal(27), Unit('°C'))
```

Alternatively, an instance of *TableConverter* can be created and registered as converter.

The example given above can be implemented as follows:

```
>>> tconv = TableConverter({(CELSIUS, FAHRENHEIT): (Decimal('1.8'), 32)})
>>> Temperature.register_converter(tconv)
>>> t27c = Temperature(Decimal(27), CELSIUS)
>>> t27c.convert(FAHRENHEIT)
Temperature(Decimal('80.6'), Unit('°F'))
```

It is suffient to define the conversion in one direction, because a reversed conversion is used automatically:

```
>>> t27c.convert(FAHRENHEIT).convert(CELSIUS)
Temperature(Decimal(27), Unit('°C'))
```

2.1.4 Unit-safe computations

Comparison

Quantities can be compared to other quantities using all comparison operators defined for numbers:

```
>>> Length(27) > Length(9)
True
>>> Length(27) >= Length(91)
False
>>> Length(27) < Length(9)
False
>>> Length(27) <= Length(91)
True
>>> Length(27) == Length(27)
True
>>> Length(27) != Length(91)
True
```

Different units are taken in to account automatically, as long as they are compatible, that means a conversion is available:

```
>>> Length(27, METRE) <= Length(91, CENTIMETRE)
False
>>> Temperature(20, CELSIUS) > Temperature(20, FAHRENHEIT)
True
>>> Temperature(20, CELSIUS) > Temperature(20, KELVIN)
Traceback (most recent call last):
UnitConversionError: Can't convert 'K' to '°C'.
```

Testing instances of different quantity types for equality always returns false:

```
>>> Length(20) == Duration(20)
False
>>> Length(20) != Duration(20)
True
```

All other comparison operators raise an *IncompatibleUnitsError* in this case.

Addition and subtraction

Quantities can be added to or subtracted from other quantities . . . :

```
>>> Length(27) + Length(9)
Length(Decimal(36))
>>> Length(27) - Length(91)
Length(Decimal(-64))
```

. . . as long as they are instances of the same quantity type:

```
>>> Length(27) + Duration(9)
Traceback (most recent call last):
IncompatibleUnitsError: Can't add a 'Length' and a 'Duration'
```

When quantities with different units are added or subtracted, the values are converted to the unit of the first, if possible ...:

```
>>> Length(27) + Length(12, CENTIMETRE)
Length(Decimal('27.12'))
>>> Length(12, CENTIMETRE) + Length(17, METRE)
Length(Decimal(1712), Unit('cm'))
>>> Temperature(20, CELSIUS) - Temperature(50, FAHRENHEIT)
Temperature(Decimal(10), Unit('°C'))
```

... but an exception is raised, if not:

```
>>> Temperature(20, CELSIUS) - Temperature(281, KELVIN)
Traceback (most recent call last):
UnitConversionError: Can't convert 'K' to '°C'.
```

Multiplication and division

Quantities can be multiplied or divided by scalars, preserving the unit:

```
>>> 7.5 * Length(3, CENTIMETRE)
Length(Decimal('22.5'), Unit('cm'))
>>> SECOND = Duration.ref_unit
>>> MINUTE = Duration.new_unit('min', 'Minute', Decimal(60) * SECOND)
>>> Duration(66, MINUTE) / 11
Duration(Decimal(6), Unit('min'))
```

Quantities can be multiplied or divided by other quantities ...:

```
>>> Length(15, METRE) / Duration(3, SECOND)
Velocity(Decimal(5))
```

... as long as the resulting type of quantity is defined ...:

```
>>> Duration(4, SECOND) * Length(7)
Traceback (most recent call last):
UndefinedResultError: Undefined result: Duration * Length
>>> Length(12, KILOMETRE) / Duration(2, MINUTE) / Duration(50, SECOND)
Traceback (most recent call last):
UndefinedResultError: Undefined result: Velocity / Duration
>>> class Acceleration(Quantity,
...                    define_as=Length / Duration ** 2,
...                    ref_unit_name='Metre per Second squared'):
...     pass
...
>>> Length(12, KILOMETRE) / Duration(2, MINUTE) / Duration(50, SECOND)
Acceleration(Decimal(2))
```

... or the result is a scalar:

```
>>> Duration(2, MINUTE) / Duration(50, SECOND)
Decimal('2.4')
```

When cascading operations, all intermediate results have to be defined:

```
>>> Length(6, KILOMETRE) * Length(13,  METRE) * Length(250, METRE)
Traceback (most recent call last):
UndefinedResultError: Undefined result: Length * Length
>>> class Area(Quantity,
...            define_as=Length ** 2,
...            ref_unit_name='Square Metre'):
...     pass
...
>>> Length(6, KILOMETRE) * Length(13,  METRE) * Length(250, METRE)
Volume(Decimal(19500000))
```

Exponentiation

Quantities can be raised by an exponent, as long as the exponent is an *Integral* number and the resulting quantity is defined:

```
>>> (5 * METRE) ** 2
Area(Decimal(25))
>>> (5 * METRE) ** 2.5
Traceback (most recent call last):
TypeError: unsupported operand type(s) for ** or pow(): 'Length' and
    'float'
>>> (5 * METRE) ** -2
Traceback (most recent call last):
UndefinedResultError: Undefined result: Length ** -2
```

2.1.5 Rounding

The amount of a quantity can be rounded by using the standard *round* function. It returns a copy of the quanitity, with its amount rounded accordingly:

```
>>> round(Length(Decimal('17.375'), MILLIMETRE), 1)
Length(Decimal('17.4'), Unit('mm'))
```

In any case the unit of the resulting quantity will be the same as the unit of the called quantity.

For more advanced cases of rounding the method *Quantity.quantize()* can round a quantity to any quantum according to any rounding mode:

```
>>> l = Length('1.7296 km')
>>> l.quantize(Length(1))
Length(Decimal('1.73', 3), Unit('km'))
>>> l.quantize(25 * METRE)
Length(Decimal('1.725'), Unit('km'))
>>> l.quantize(25 * METRE, ROUNDING.ROUND_UP)
Length(Decimal('1.75', 3), Unit('km'))
```

2.1.6 Apportioning

The method *Quantity.allocate()* can be used to apportion a quantity according to a sequence of ratios:

```
>>> m = Mass('10 kg')
>>> ratios = [38, 5, 2, 15]
>>> portions, remainder = m.allocate(ratios)
>>> portions
[Mass(Fraction(19, 3)),
 Mass(Fraction(5, 6)),
 Mass(Fraction(1, 3)),
 Mass(Decimal('2.5', 2))]
>>> remainder
Mass(Decimal(0, 2))
```

If the quantity is quantized, there can be rounding errors causing a remainder with an amount other than 0:

```
>>> b = 10 * KILOBYTE
>>> portions, remainder = b.allocate(ratios, disperse_rounding_error=False)
>>> portions
[DataVolume(Decimal('6.333375'), Unit('kB')),
 DataVolume(Decimal('0.833375'), Unit('kB')),
 DataVolume(Decimal('0.333375'), Unit('kB')),
 DataVolume(Decimal('2.5', 6), Unit('kB'))]
>>> remainder
DataVolume(Decimal('-0.000125'), Unit('kB'))
```

By default the remainder will be dispersed:

```
>>> portions, remainder = b.allocate(ratios)
>>> portions
[DataVolume(Decimal('6.333375'), Unit('kB')),
 DataVolume(Decimal('0.833375'), Unit('kB')),
 DataVolume(Decimal('0.33325', 6), Unit('kB')),
 DataVolume(Decimal('2.5', 6), Unit('kB'))]
>>> remainder
DataVolume(Decimal(0), Unit('kB'))
```

As well as of numbers, quantities can be used as ratios (as long as they have compatible units):

```
>>> CUBIC_METRE = Volume.ref_unit
>>> LITRE = Volume.new_unit('l', 'Litre', MILLI * CUBIC_METRE)
>>> l = 10 * LITRE
>>> ratios = [350 * GRAM, 500 * GRAM, 3 * KILOGRAM, 150 * GRAM]
>>> l.allocate(ratios)
([Volume(Decimal('0.875', 4), Unit('l')),
  Volume(Decimal('1.25', 3), Unit('l')),
  Volume(Decimal('7.5', 2), Unit('l')),
  Volume(Decimal('0.375', 4), Unit('l'))],
 Volume(Decimal(0, 4), Unit('l')))
```

2.1.7 Formatting as string

Quantity supports the standard *str* function. It returns a string representation of the quantity's amount followed by a blank and the quantity's units symbol.

In addition, *Quantity* supports the standard *format* function. The format specifier should use two keys: 'a' for the amount and 'u' for the unit, where 'a' can be followed by a valid format spec for numbers and 'u' by a valid format spec for strings. If no format specifier is given, '{a} {u}' is used:

```
>>> v = Volume('19.36')
>>> format(v)
'19.36 m³'
>>> format(v, '{a:*>10.2f} {u:<3}')
'*****19.36 m³ '
```

2.2 Types

QuantityClsDefT

Defintion of derived Quantity sub-classes.

alias of Term[*QuantityMeta*]

UnitDefT

Definition of derived units.

alias of Term[*Unit*]

AmountUnitTupleT

Tuple of an amount and an optional unit

alias of Tuple[Rational, Optional[*Unit*]]

BinOpResT

Result of binary operations on quantities / units

alias of Union[*Quantity*, Rational, Tuple[Rational, Optional[*Unit*]]]

ConverterT

Type of converters

alias of Callable[[...], Optional[Rational]]

ConvMapT

The central part of internal API.

This represents a generic version of type 'origin' with type arguments 'params'. There are two kind of these aliases: user defined and special. The special ones are wrappers around builtin collections and ABCs in collections.abc. These must have 'name' always set. If 'inst' is False, then the alias can't be instantiated, this is used by e.g. typing.List and typing.Dict.

alias of Mapping[Tuple[*Unit*, *Unit*], Tuple[Rational, Rational]]

ConvSpecIterableT

The central part of internal API.

This represents a generic version of type 'origin' with type arguments 'params'. There are two kind of these aliases: user defined and special. The special ones are wrappers around builtin collections and ABCs in collections.abc. These must have 'name' always set. If 'inst' is False, then the alias can't be instantiated, this is used by e.g. typing.List and typing.Dict.

alias of `Iterable[Tuple[Unit, Unit, Rational, Rational]]`

2.3 Classes

class Unit

Unit of measure.

Note: New instances of *Unit* can not be created directly by calling *Unit*. Instead, use *<Quantity subclass>.new_unit*.

__copy__() → *Unit*

Return self (*Unit* instances are immutable).

__eq__(*other: Any*) → bool

self == other

__format__(*fmt_spec: str = ''*) → str

Convert to string (according to *fmt_spec*).

fmt_spec must be a valid format spec for strings.

__ge__(*other: Any*) → bool

self >= other

__gt__(*other: Any*) → bool

self > other

__hash__() → int

hash(self)

__le__(*other: Any*) → bool

self <= other

__lt__(*other: Any*) → bool

self < other

__mul__(*other: int*) → *Quantity*

__mul__(*other: float*) → *Quantity*

__mul__(*other: Real*) → *Quantity*

__mul__(*other: SIPrefix*) → *Quantity*

__mul__(*other:* Unit) → AmountUnitTupleT

__mul__(*other:* Quantity) → BinOpResT

self * other

__pow__(*exp: Any*) → *Quantity* | Rational

self ** exp

__repr__() → str
repr(self)

__rmul__(*other: int*) → *Quantity*
__rmul__(*other: float*) → *Quantity*
__rmul__(*other: Real*) → *Quantity*
__rmul__(*other: SIPrefix*) → *Quantity*
__rmul__(*other:* Unit) → AmountUnitTupleT
__rmul__(*other:* Quantity) → BinOpResT
other * self

__rtruediv__(*other: Any*) → *Quantity*
other / self

__str__() → str
str(self)

__truediv__(*other: int*) → *Quantity*
__truediv__(*other: float*) → *Quantity*
__truediv__(*other: Real*) → *Quantity*
__truediv__(*other:* Unit) → AmountUnitTupleT
__truediv__(*other:* Quantity) → BinOpResT
self / other

is_base_unit() → bool
Return True if the unit is not derived from another unit.

is_derived_unit() → bool
Return True if the unit is derived from another unit.

is_ref_unit() → bool
Return True if the unit is a reference unit.

property definition: UnitDefT
Return the units definition.

property name: str
Return the units name.

If the unit was not given a name, its symbol is returned.

property normalized_definition: UnitDefT
Return the units normalized definition.

property qty_cls: *QuantityMeta*
Return the *Quantity* subclass related to the unit.

property quantum: Rational | None
Return the minimum amount of a quantity with the unit as unit.

Returns None if the quantity class related to the unit does not define a quantum.

property symbol: str
Return the units symbol.

The symbol is a unique string representation of the unit.

class QuantityMeta

Meta class allowing to construct Quantity subclasses.

Parameters

- **name** – name of the new quantity type
- **define_as** (*Optional[QuantityClsDefT]*) – definition of the new derived quantity type
- **ref_unit_symbol** (*Optional[str]*) – symbol of the reference unit to be created
- **ref_unit_name** (*Optional[str]*) – name of the reference unit
- **quantum** (*Optional[Rational]*) – minimum absolute amount for an instance of the new quantity type

__init__(*name: str*, *bases: Tuple[type, ...] = ()*, *clsdict=MappingProxyType({})*, ***kwds: Any*)

__mul__(*other:* QuantityMeta | *Term[ClassWithDefinitionMeta]*) → Term[ClassWithDefinitionMeta]

Return class definition: *cls * other*.

static __new__(*mcs*, *name: str*, *bases: Tuple[type, ...] = ()*, *clsdict=MappingProxyType({})*, ***kwds: Any*) → *QuantityMeta*

Create new Quantity (sub-)class.

__rmul__(*other: Term[ClassWithDefinitionMeta]*) → Term[ClassWithDefinitionMeta]

Return class definition: *other * cls*.

__truediv__(*other:* QuantityMeta | *Term[ClassWithDefinitionMeta]*) → Term[ClassWithDefinitionMeta]

Return class definition: *cls / other*.

derive_unit_from(**args:* Unit, *symbol: str | None = None*, *name: str | None = None*) → *Unit*

Derive a new unit for *cls* from units of its base quantities.

Parameters

- **args** – iterable of units of the base quantities of the quantity type
- **symbol** – symbol of the new unit, generated based on *args* if not given
- **name** – name of the new unit, defaults to *symbol* if not given

Raises

- **TypeError** – 'derive_unit_from' called on a base quantity
- **TypeError** – not all members of *args* are instances of *Unit*
- **ValueError** – number of given base units doesn't match number of base quantities of *cls*
- **ValueError** – given base units don't match base quantities
- **TypeError** – *symbol* is not a string or None
- **ValueError** – *symbol* is empty

get_unit_by_symbol(*symbol: str*) → *Unit*

Return the unit with symbol *symbol*.

Parameters
symbol – symbol to look-up

Returns
unit with given *symbol*

Raises

ValueError – a unit with given *symbol* is not registered with *cls*

is_base_cls() → bool

Return True if *cls* is not derived from other class(es).

is_derived_cls() → bool

Return True if *cls* is derived from other class(es).

new_unit(*symbol: str*, *name: str | None = None*, *define_as:* Quantity | *UnitDefT | None = None*) → *Unit*

Create, register and return a new unit for *cls*.

Parameters

- **symbol** – symbol of the new unit
- **name** – name of the new unit, defaults to *symbol* if not given
- **define_as** – equivalent of the new unit in terms of another unit (usually given by multiplying a scalar or a SI scale and a unit) or a term defining the new unit in terms of other units

Raises

- **TypeError** – *symbol* is not a string or None
- **ValueError** – *symbol* is empty
- **ValueError** – a unit with the given symbol is already registered
- **TypeError** – *define_as* does not match the quantity type
- **ValueError** – term given as *define_as* does not define a unit

register_converter(*conv: ConverterT*) → None

Add converter *conv* to the list of converters registered in cls.

Does nothing if converter is already registered.

registered_converters() → Iterator[ConverterT]

Return an iterator over the converters registered in 'cls'.

The converts are returned in reversed order of registration.

remove_converter(*conv: ConverterT*) → None

Remove converter *conv* from the converters registered in cls.

Raises ValueError if the converter is not present.

units() → Tuple[*Unit*, ...]

Return all registered units of *cls* as tuple.

property definition: Term[ClassWithDefinitionMeta]

Definition of *cls*.

property normalized_definition: Term[ClassWithDefinitionMeta]

Normalized definition of *cls*.

property quantum: Rational | None

Return the minimum absolute amount for an instance of *cls*.

The quantum is the minimum amount (in terms of the reference unit) an instance of *cls* can take (None if no quantum is defined).

property ref_unit: *Unit* | **None**

Return the reference unit of *cls*, or None if no one is defined.

class Quantity

Base class for all types of quantities.

__abs__() → *Quantity*

abs(self) -> self.Quantity(abs(self.amount), self.unit)

__add__(*other:* Quantity) → *Quantity*

self + other

__eq__(*other: Any*) → bool

self == other

__format__(*fmt_spec: str = ''*) → str

Convert to string (according to format specifier).

The specifier must be a standard format specifier in the form described in PEP 3101. It should use two keys: 'a' for self.amount and 'u' for self.unit, where 'a' can be followed by a valid format spec for numbers and 'u' by a valid format spec for strings.

__ge__(*other: Any*) → bool

self >= other

__gt__(*other: Any*) → bool

self > other

__hash__() → int

hash(self)

__le__(*other: Any*) → bool

self <= other

__lt__(*other: Any*) → bool

self < other

__mul__(*other: int*) → *Quantity*
__mul__(*other: float*) → *Quantity*
__mul__(*other: Real*) → *Quantity*
__mul__(*other:* Quantity) → BinOpResT
__mul__(*other:* Unit) → BinOpResT

self * other

__neg__() → *Quantity*

-self -> self.Quantity(-self.amount, self.unit)

__pos__() → *Quantity*

+self

__pow__(*exp: int*) → *Quantity*

self ** exp

__radd__(*other:* Quantity) → *Quantity*

self + other

__repr__() → str
repr(self)

__rmul__(*other: Any*) → BinOpResT
self * other

__round__(*n_digits: int = 0*) → *Quantity*
Return copy of *self* with its amount rounded to *n_digits*.

Parameters
n_digits – number of fractional digits to be rounded to

Returns
round(self.amount, n_digits) * self.unit

__rsub__(*other:* Quantity) → *Quantity*
other - self

__rtruediv__(*other: Any*) → *Quantity*
other / self

__str__() → str
str(self)

__sub__(*other:* Quantity) → *Quantity*
self - other

__truediv__(*other: int*) → *Quantity*
__truediv__(*other: float*) → *Quantity*
__truediv__(*other: Real*) → *Quantity*
__truediv__(*other:* Quantity) → BinOpResT
__truediv__(*other:* Unit) → BinOpResT
self / other

allocate(*ratios: Collection[Rational | Quantity], disperse_rounding_error: bool = True*) → Tuple[List[*Quantity*], *Quantity*]
Apportion *self* according to *ratios*.

Parameters

- **ratios** – sequence of values defining the relative amount of the requested portions
- **disperse_rounding_error** – determines whether a rounding error (if there is one due to quantization) shall be dispersed

Returns

(portions of *self* according to *ratios*,
remainder = *self* - sum(portions))

Raises

- **TypeError** – *ratios* contains elements that can not be added
- ***IncompatibleUnitsError*** – *ratios* contains quantities that can not be added

convert(*to_unit:* Unit) → *Quantity*
Return quantity q where q == *self* and q.unit is *to_unit*.

Parameters
to_unit – unit to be converted to

Returns
quantity equivalent to *self*, having unit *to_unit*

Raises
IncompatibleUnitsError – *self* can't be converted to *to_unit*.

equiv_amount(*unit:* Unit) → Rational | None

Return amount e so that e * *unit* == *self*.

quantize(*quant:* Quantity, *rounding: ROUNDING | None = None*) → *Quantity*

Return integer multiple of *quant* closest to *self*.

Parameters

- **quant** – quantum to get a multiple from
- **rounding** – rounding mode (default: None)

If no *rounding* mode is given, the current default mode from module *decimalfp* is used.

Returns

integer multiple of *quant* closest to *self* (according to *rounding* mode)

Raises

- ***IncompatibleUnitsError*** – *quant* can not be converted to self.unit
- **TypeError** – *quant* is not an instance of type(self)
- **TypeError** – type(self) has no reference unit

property amount: Rational

Return the numerical part of the quantity.

property unit: *Unit*

Return the quantity's unit.

class Converter

Convert a quantity's amount to the equivalent amount for another unit.

A quantity converter can be any callable with a signature like conv(qty, to_unit) -> number f so that type(qty)(f, to_unit) == qty.

__call__(*qty:* Quantity, *to_unit:* Unit) → Rational | None

Convert *qty*'s amount to the equivalent amount for *to_unit*.

Parameters

- **qty** – quantity to be converted
- **to_unit** – unit for equivalent amount

Returns
factor f so that f * *to_unit* == *qty*, or None if no such factor is available

Raises

- ***IncompatibleUnitsError*** – *qty* and *to_unit* are incompatible
- ***UnitConversionError*** – conversion factor not available

__init__()

__new__(***kwargs*)

class TableConverter

Bases: *Converter*

Converter using a conversion table.

Parameters
conv_table (*Mapping or list*) – the mapping used to initialize the conversion table

Each item of the conversion table defines a conversion from one unit to another unit and consists of four elements:

- from_unit (Unit): unit of the quantity to be converted
- to_unit (Unit): target unit of the conversion
- factor (Rational): factor to be applied to the quantity's amount
- offset (Rational): an amount added after applying the factor

When a *Mapping* is given as *convTable*, each key / value pair must map a tuple (from_unit, to_unit) to a tuple (factor, offset).

When a *list* is given as *convTable*, each item must be a tuple (from_unit, to_unit, factor, offset).

factor and *offset* must be set so that for an amount in terms of *from_unit* the eqivalent amount in terms of *to_unit* is:

result = amount * factor + offset

An instance of *TableConverter* can be called with a *Quantity* sub-class' instance *qty* and a *Unit* sub-class' instance *to_unit* as parameters. It looks-up the pair (*qty.unit*, *to_unit*) for a factor and an offset and returns the resulting amount according to the formula given above.

If there is no item for the pair (*qty.unit*, *to_unit*), it tries to find a reverse mapping by looking-up the pair (*to_unit*, *qty.unit*), and, if it finds one, it returns a result by applying a reversed formula:

result = (amount - offset) / factor

That means, for each pair of units it is sufficient to define a conversion in one direction.

An instance of *TableConverter* can be directly registered as a converter by calling the method *Quantity.register_converter*.

__init__(*conv_table: ConvMapT | ConvSpecIterableT*)

__new__(***kwargs*)

2.4 Functions

sum(*items: Iterable[Any]*, *start: Any | None = None*) → Any

Return the sum of *start* (if not None) plus all items in *items*.

Parameters

- **items** – iterable of numbers or number-like objects (NOT strings)
- **start** – starting value to be added (default: None)

Returns
sum of all elements in *items* plus the value of *start* (if not None). When *items* is empty, returns *start*, if not None, otherwise 0.

In contrast to the built-in function 'sum' this function allows to sum sequences of number-like objects (like quantities) without having to provide a start value.

2.5 Exceptions

exception QuantityError

Raised when a quantity can not be instanciated.

__init__(**args*, ***kwargs*)

__new__(***kwargs*)

exception IncompatibleUnitsError

Raised when operands do not have compatible units.

__init__(*msg: str*, *operand1: Any*, *operand2: Any*)

__new__(***kwargs*)

exception UndefinedResultError

Raised when operation results in an undefined quantity.

__init__(*op: Callable[[Any, Any], Any]*, *operand1: Any*, *operand2: Any*)

__new__(***kwargs*)

exception UnitConversionError

Raised when a conversion between two compatible units fails.

__init__(*msg: str*, *from_unit: Any*, *to_unit: Any*)

__new__(***kwargs*)

CHAPTER

THREE

PREDEFINED QUANTITIES

Definitions of commonly used quantities and units.

3.1 Mass

Reference unit: Kilogram (‘kg’)

Predefined units:

Symbol	Name	Definition	Equivalent in ‘kg’
mg	Milligram	0.001·g	0.000001
ct	Carat	0.2·g	0.0002
g	Gram	0.001·kg	0.001
oz	Ounce	0.0625·lb	0.028349523125
lb	Pound	0.45359237·kg	0.45359237
st	Stone	14·lb	6.35029318
t	Tonne	1000·kg	1000

3.2 Length

Reference unit: Metre (‘m’)

Predefined units:

Symbol	Name	Definition	Equivalent in ‘m’
nm	Nanometre	0.000000001·m	0.000000001
µm	Micrometre	0.000001·m	0.000001
mm	Millimetre	0.001·m	0.001
cm	Centimetre	0.01·m	0.01
in	Inch	2.54·cm	0.0254
dm	Decimetre	0.1·m	0.1
ft	Foot	12·in	0.3048
yd	Yard	3·ft	0.9144
ch	Chain	22·yd	20.1168
fur	Furlog	10·ch	201.168
km	Kilometre	1000·m	1000
mi	Mile	8·fur	1609.344

3.3 Duration

Reference unit: Second ('s')

Predefined units:

Symbol	Name	Definition	Equivalent in 's'
ns	Nanosecond	0.000000001·s	0.000000001
µs	Microsecond	0.000001·s	0.000001
ms	Millisecond	0.001·s	0.001
min	Minute	60·s	60
h	Hour	60·min	3600
d	Day	24·h	86400

3.4 Area

Definition: Length^2

Reference unit: Square Metre ('m^2')

Predefined units:

Symbol	Name	Definition	Equivalent in 'm^2'
mm^2	Square Millimetre	mm^2	0.000001
cm^2	Square Centimetre	cm^2	0.0001
in^2	Square Inch	in^2	0.00064516
dm^2	Square Decimetre	dm^2	0.01
ft^2	Square Foot	ft^2	0.09290304
yd^2	Square Yard	yd^2	0.83612736
a	Are	100·m^2	100
ac	Acre	4840·yd^2	4046.8564224
ha	Hectare	100·a	10000
km^2	Square Kilometre	km^2	1000000
mi^2	Square Mile	mi^2	2589988.110336

3.5 Volume

Definition: Length^3

Reference unit: Cubic Metre ('m^3')

Predefined units:

Symbol	Name	Definition	Equivalent in ‘m^3’
mm^3	Cubic Millimetre	mm^3	0.000000001
cm^3	Cubic Centimetre	cm^3	0.000001
ml	Millilitre	0.001·l	0.000001
cl	Centilitre	0.01·l	0.00001
in^3	Cubic Inch	in^3	0.000016387064
dl	Decilitre	0.1·l	0.0001
dm^3	Cubic Decimetre	dm^3	0.001
l	Litre	0.001·m^3	0.001
ft^3	Cubic Foot	ft^3	0.028316846592
yd^3	Cubic Yard	yd^3	0.764554857984
km^3	Cubic Kilometre	km^3	1000000000

3.6 Velocity

Definition: Length/Duration

Reference unit: Metre per Second (‘m/s’)

Predefined units:

Symbol	Name	Definition	Equivalent in ‘m/s’
km/h	Kilometre per hour	km/h	5/18
ft/s	Foot per Second	ft/s	0.3048
mph	Mile per Hour	mi/h	0.44704

3.7 Acceleration

Definition: Length/$Duration^2$

Reference unit: Metre per Second squared (‘m/s^2’)

Predefined units:

Symbol	Name	Definition	Equivalent in ‘m/s^2’
mps^2	Mile per Second squared	mi/s^2	1609.344

3.8 Force

Definition: Mass·Acceleration

Reference unit: Newton (‘N’ = ‘kg·m/s^2’)

Predefined units:

Symbol	Name	Definition	Equivalent in 'N'
J/m	Joule per Metre	J/m	1

3.9 Energy

Definition: Force·Length

Reference unit: Joule ('J' = 'N·m' = 'kg·m^2/s^2')

Predefined units:

Symbol	Name	Definition	Equivalent in 'J'
Nm	Newton Meter	N·m	1
Ws	Watt Second	W·s	1
kWh	Kilowatt Hour	kW·h	3600000

3.10 Power

Definition: Energy/Duration

Reference unit: Watt ('W' = 'J/s' = 'kg·m^2/s^3')

Predefined units:

Symbol	Name	Definition	Equivalent in 'W'
mW	Milliwatt	0.001·W	0.001
kW	Kilowatt	1000·W	1000
MW	Megawatt	1000000·W	1000000
GW	Gigawatt	1000000000·W	1000000000
TW	Terawatt	1000000000000·W	1000000000000

3.11 Frequency

Definition: 1/Duration

Reference unit: Hertz ('Hz' = '1/s')

Predefined units:

Symbol	Name	Definition	Equivalent in 'Hz'
kHz	Kilohertz	1000·Hz	1000
MHz	Megahertz	1000000·Hz	1000000
GHz	Gigahertz	1000000000·Hz	1000000000

3.12 DataVolume

Reference unit: Byte ('B')

Predefined units:

Symbol	Name	Definition	Equivalent in 'B'
b	Bit	0.125·B	0.125
kb	Kilobit	1000·b	125
Kib	Kibibit	1024·b	128
kB	Kilobyte	1000·B	1000
KiB	Kibibyte	1024·B	1024
Mb	Megabit	1000000·b	125000
Mib	Mebibit	1048576·b	131072
MB	Megabyte	1000000·B	1000000
MiB	Mebibyte	1048576·B	1048576
Gb	Gigabit	1000000000·b	125000000
Gib	Gibibit	1073741824·b	134217728
GB	Gigabyte	1000000000·B	1000000000
GiB	Gibibyte	1073741824·B	1073741824
Tb	Terabit	1000000000000·b	125000000000
Tib	Tebibit	1099511627776·b	137438953472
TB	Terabyte	1000000000000·B	1000000000000
TiB	Tebibyte	1099511627776·B	1099511627776

3.13 DataThroughput

Definition: DataVolume/Duration

Reference unit: Byte per Second ('B/s')

Predefined units:

Symbol	Name	Definition	Equivalent in 'B/s'
b/s	Bit per Second	b/s	0.125
kb/s	Kilobit per Second	1000·b/s	125
Kib/s	Kibibit per Second	1024·b/s	128
kB/s	Kilobyte per Second	1000·B/s	1000
KiB/s	Kibibyte per Second	1024·B/s	1024
Mb/s	Megabit per Second	1000000·b/s	125000
Mib/s	Mebibit per Second	1048576·b/s	131072
MB/s	Megabyte per Second	1000000·B/s	1000000
MiB/s	Mebibyte per Second	1048576·B/s	1048576
Gb/s	Gigabit per Second	1000000000·b/s	125000000
Gib/s	Gibibit per Second	1073741824·b/s	134217728
GB/s	Gigabyte per Second	1000000000·B/s	1000000000
GiB/s	Gibibyte per Second	1073741824·B/s	1073741824
Tb/s	Terabit per Second	1000000000000·b/s	125000000000
Tib/s	Tebibit per Second	1099511627776·b/s	137438953472
TB/s	Terabyte per Second	1000000000000·B/s	1000000000000
TiB/s	Tebibyte per Second	1099511627776·B/s	1099511627776

3.14 Temperature

Predefined units:

Symbol	Name	Equivalents
°C	Degree Celsius	0 °C = 32 °F = 273,25 K
°F	Degree Fahrenheit	0 °F -17.778 °C 255.372 K
K	Kelvin	0 K = -273,25 °C = -459.67 °F

Temperature units are converted using the following formulas:

from to	Celsius	Fahrenheit	Kelvin
Celsius	•	[°F] = [°C] * 9/5 + 32	[K] = [°C] + 273.15
Fahrenheit	[°C] = ([°F] - 32) * 5/9	•	[K] = ([°F] + 459.67) * 5/9
Kelvin	[°C] = [K] - 273.15	[°F] = [K] * 9/5 - 459.67	•

CHAPTER

FOUR

MONEY

Currency-safe computations with money amounts.

4.1 Usage

4.1.1 Registering a currency

A currency must explicitly be registered as a unit for further use. The easiest way to do this is to call `Money.register_currency()`:

```
>>> from quantity.money import Money
>>> EUR = Money.register_currency('EUR')
>>> HKD = Money.register_currency('HKD')
>>> TND = Money.register_currency('TND')
>>> USD = Money.register_currency('USD')
>>> EUR, HKD, TND, USD
(Currency('EUR'), Currency('HKD'), Currency('TND'), Currency('USD'))
```

The method is backed by a database of currencies defined in ISO 4217. It takes the 3-character ISO 4217 code as parameter.

Currency derives from *Unit*. Each instance has a symbol (which is usually the 3-character ISO 4217 code) and a name. In addition, it holds the smallest fraction defined for amounts in this currency:

```
>>> TND.symbol
'TND'
>>> TND.name
'Tunisian Dinar'
>>> TND.smallest_fraction
Decimal('0.001')
```

4.1.2 Instantiating a money amount

As *Money* derives from *Quantity*, an instance can simply be created by giving an amount and a unit:

```
>>> Money(30, EUR)
Money(Decimal(30, 2), Currency('EUR'))
```

All amounts of money are rounded according to the smallest fraction defined for the currency:

```
>>> Money(3.128, EUR)
Money(Decimal('3.13'), Currency('EUR'))
>>> Money(41.1783, TND)
Money(Decimal('41.178'), Currency('TND'))
```

As with other quantities, money amounts can also be derived from a string or build using the operator *:

```
>>> Money('3.18 USD')
Money(Decimal('3.18'), Currency('USD'))
>>> 3.18 * USD
Money(Decimal('3.18'), Currency('USD'))
```

4.1.3 Computing with money amounts

Money derives from *Quantity*, so all operations on quantities can also be applied to instances of *Money*. But because there is no fixed relation between currencies, there is no implicit conversion between money amounts of different currencies:

```
>>> Money(30, EUR) + Money(3.18, EUR)
Money(Decimal('33.18'), Currency('EUR'))
>>> Money(30, EUR) + Money(3.18, USD)
Traceback (most recent call last):
UnitConversionError: Can't convert 'USD' to 'EUR'
```

Resulting values are always quantized to the smallest fraction defined with the currency:

```
>>> Money('3.20 USD') / 3
Money(Decimal('1.07'), Currency('USD'))
>>> Money('3.20 TND') / 3
Money(Decimal('1.067'), Currency('TND'))
```

4.1.4 Converting between different currencies

Exchange rates

A conversion factor between two currencies can be defined by using the *ExchangeRate*. It is given a unit currency (aka base currency), a unit multiple, a term currency (aka price currency) and a term amount, i.e. the amount in term currency equivalent to unit multiple in unit currency:

```
>>> fxEUR2HKD = ExchangeRate(EUR, 1, HKD, Decimal('8.395804'))
>>> fxEUR2HKD
ExchangeRate(Currency('EUR'), Decimal(1), Currency('HKD'), Decimal('8.395804'))
```

unit_multiple and *term_amount* will always be adjusted so that the resulting unit multiple is a power to 10 and the resulting term amounts magnitude is >= -1. The latter will always be rounded to 6 decimal digits:

```
>>> fxTND2EUR = ExchangeRate(TND, 5, EUR, Decimal('0.0082073'))
>>> fxTND2EUR
ExchangeRate(Currency('TND'), Decimal(100), Currency('EUR'), Decimal('0.164146'))
```

The resulting rate for an amount of 1 unit currency in term currency can be obtained via the property *ExchangeRate.rate*:

```
>>> fxTND2EUR.rate
Decimal('0.00164146')
```

The property *ExchangeRate.quotation* gives a tuple of unit currency, term currency and rate:

```
>>> fxTND2EUR.quotation
(Currency('TND'), Currency('EUR'), Decimal('0.00164146'))
```

The properties `ExchangeRate.inverseRate` and `ExchangeRate.inverseQuotation` give the rate and the quotation in the opposite direction (but do not round the rate!):

```
>>> fxTND2EUR.inverse_rate
Fraction(50000000, 82073)
>>> fxTND2EUR.inverse_quotation
(Currency('EUR'), Currency('TND'), Fraction(50000000, 82073))
```

The inverse ExchangeRate can be created by calling the method *ExchangeRate.inverted()*:

```
>>> fxEUR2TND = fxTND2EUR.inverted()
>>> fxEUR2TND
ExchangeRate(Currency('EUR'), Decimal(1), Currency('TND'), Decimal('609.213749'))
```

An exchange rate can be derived from two other exchange rates, provided that they have one currency in common ("triangulation"). If the unit currency of one exchange rate is equal to the term currency of the other, the two exchange rates can be multiplied with each other. If either the unit currencies or the term currencies are equal, the two exchange rates can be divided:

```
>>> fxEUR2HKD * fxTND2EUR
ExchangeRate(Currency('TND'), Decimal(10), Currency('HKD'), Decimal('0.137814'))
>>> fxEUR2HKD / fxEUR2TND
ExchangeRate(Currency('TND'), Decimal(10), Currency('HKD'), Decimal('0.137814'))
>>> fxEUR2TND / fxEUR2HKD
ExchangeRate(Currency('HKD'), Decimal(1), Currency('TND'), Decimal('72.561693'))
>>> fxHKD2EUR = fxEUR2HKD.inverted()
>>> fxTND2EUR / fxHKD2EUR
ExchangeRate(Currency('TND'), Decimal(10), Currency('HKD'), Decimal('0.137814'))
```

Converting money amounts using exchange rates

Multiplying an amount in some currency with an exchange rate with the same currency as unit currency results in the equivalent amount in term currency:

```
>>> mEUR = 5.27 * EUR
>>> mEUR * fxEUR2HKD
Money(Decimal('44.25'), Currency('HKD'))
>>> mEUR * fxEUR2TND
Money(Decimal('3210.556'), Currency('TND'))
```

Likewise, dividing an amount in some currency with an exchange rate with the same currency as term currency results in the equivalent amount in unit currency:

```
>>> fxHKD2EUR = fxEUR2HKD.inverted()
>>> mEUR / fxHKD2EUR
Money(Decimal('44.25'), Currency('HKD'))
```

Using money converters

Money converters can be used to hold different exchange rates, each of them linked to a period of validity. The type of period must be the same for all exchange rates held by a money converter.

A money converter is created by calling *MoneyConverter*, giving the base currency used by this converter:

```
>>> conv = MoneyConverter(EUR)
```

The method *MoneyConverter.update()* is then used to feed exchange rates into the converter.

For example, a money converter with monthly rates can be created like this:

```
>>> import datetime
>>> today = datetime.date.today()
>>> year, month, day = today.timetuple()[:3]
>>> prev_month = month - 1
>>> rates = [(USD, Decimal('1.1073'), 1),
...          (HKD, Decimal('8.7812'), 1)]
>>> conv.update((year, prev_month), rates)
>>> rates = [(USD, Decimal('1.0943'), 1),
...          (HKD, Decimal('8.4813'), 1)]
>>> conv.update((year, month), rates)
```

Exchange rates can be retrieved by calling *MoneyConverter.get_rate()*. If no reference date is given, the current date is used (unless a callable returning a different date is given when the converter is created, see below). The method returns not only rates directly given to the converter, but also inverted rates and rates calculated by triangulation:

```
>>> conv.get_rate(EUR, USD)
ExchangeRate(Currency('EUR'), Decimal(1), Currency('USD'), Decimal('1.0943', 6))
>>> conv.get_rate(HKD, EUR, date(year, prev_month, 3))
ExchangeRate(Currency('HKD'), Decimal(1), Currency('EUR'), Decimal('0.11388', 6))
>>> conv.get_rate(USD, EUR)
ExchangeRate(Currency('USD'), Decimal(1), Currency('EUR'), Decimal('0.913826'))
>>> conv.get_rate(HKD, USD)
ExchangeRate(Currency('HKD'), Decimal(1), Currency('USD'), Decimal('0.129025'))
```

A money converter can be registered with the class *Money* in order to support implicit conversion of money amounts from one currency into another (using the default reference date, see below):

```
>>> Money.register_converter(conv)
>>> twoEUR = 2 * EUR
>>> twoEUR.convert(USD)
Money(Decimal('2.19'), Currency('USD'))
```

A money converter can also be registered and unregistered by using it as context manager in a *with* statement.

In order to use a default reference date other than the current date, a callable can be given to *MoneyConverter*. It must be callable without arguments and return a date. It is then used by *get_rate()* to get the default reference date:

```
>>> yesterday = lambda: datetime.date.today() - datetime.timedelta(1)
>>> conv = MoneyConverter(EUR)       # uses today as default
>>> conv.update(yesterday(), [(USD, Decimal('1.0943'), 1)])
>>> conv.update(datetime.date.today(), [(USD, Decimal('1.0917'), 1)])
>>> conv.get_rate(EUR, USD)
ExchangeRate(Currency('EUR'), Decimal(1), Currency('USD'), Decimal('1.0917', 6))
>>> conv = MoneyConverter(EUR, get_dflt_effective_date=yesterday)
>>> conv.update(yesterday(), [(USD, Decimal('1.0943'), 1)])
>>> conv.update(datetime.date.today(), [(USD, Decimal('1.0917'), 1)])
>>> conv.get_rate(EUR, USD)
ExchangeRate(Currency('EUR'), Decimal(1), Currency('USD'), Decimal('1.0943', 6))
```

As other quantity converters, a *MoneyConverter* instance can be called to convert a money amount into the equivalent amount in another currency. But note that the amount is not adjusted to the smallest fraction of that currency:

```
>>> conv(twoEUR, USD)
Decimal('2.1886', 8)
>>> conv(twoEUR, USD, datetime.date.today())
Decimal('2.1834', 8)
```

4.1.5 Combining Money with other quantities

As *Money* derives from *Quantity*, it can be combined with other quantities in order to define a new quantity. This is, for example, useful for defining prices per quantum:

```
>>> from quantity import Quantity
>>> class Mass(Quantity,
...            ref_unit_name='Kilogram',
...            ref_unit_symbol='kg'):
...     pass
>>> KILOGRAM = Mass.ref_unit
>>> class PricePerMass(Quantity, define_as=Money / Mass):
...     pass
```

Because *Money* has no reference unit, there is no reference unit created for the derived quantity . . . :

```
>>> PricePerMass.units()
()
```

. . . instead, units must be explicitly defined:

```
>>> EURpKG = PricePerMass.derive_unit_from(EUR, KILOGRAM)
>>> PricePerMass.units()
(Unit('EUR/kg'),)
```

Instances of the derived quantity can be created and used just like those of other quantities:

```
>>> from decimalfp import Decimal
>>> p = Decimal("17.45") * EURpKG
>>> p * Decimal("1.05")
PricePerMass(Decimal('18.3225'), Unit('EUR/kg'))
>>> GRAM = Mass.new_unit('g', 'Gram', Decimal("0.001") * KILOGRAM)
>>> m = 530 * GRAM
>>> m * p
Money(Decimal('9.25'), Currency('EUR'))
```

Note that instances of the derived class are not automatically quantized to the quantum defined for the currency:

```
>>> EURpKG.quantum is None
True
```

Instances of such a "money per quantum" class can also be converted using exchange rates, as long as the resulting unit is defined:

```
>>> p * fxEUR2HKD
Traceback (most recent call last):
QuantityError: Resulting unit not defined: HKD/kg.
>>> HKDpKG = PricePerMass.derive_unit_from(HKD, KILOGRAM)
>>> p * fxEUR2HKD
PricePerMass(Decimal('146.5067798', 8), Unit('HKD/kg'))
```

4.2 Types

MoneyConverterT

Type of money converters

alias of `Callable[[Money, Currency, Optional[date]], Rational]`

ValidityT

Types used to specify time periods for the validity of exchange rates

alias of `Optional[Union[date, int, str, SupportsInt, Tuple[int, int], Tuple[Union[str, SupportsInt], Union[str, SupportsInt]]]]`

RateSpecT

Type of tuple to specify an exchange rate

alias of `Tuple[Union[Currency, str], Union[Rational, float, str], Rational]`

4.3 Classes

class Currency

Represents a currency, i.e. a money unit.

Note: New instances of *Currency* can not be created directly by calling *Currency*. Instead, use *Money.register_currency* or *Money.new_unit*.

__init__()

static __new__(*cls*, *symbol: str*) → *Unit*

Return the Unit registered with symbol *symbol*.

Parameters
symbol – symbol of the requested unit

Raises
ValueEror – no unit with given symbol registered

property iso_code: str

ISO 4217 3-character code.

property name: str

Return the units name.

If the unit was not given a name, its symbol is returned.

property smallest_fraction: Decimal

The smallest fraction available for this currency.

class Money

Bases: *Quantity*

Represents a money amount, i.e. the combination of a numerical value and a money unit, aka. currency.

Instances of *Money* can be created in two ways, by providing a numerical amount and a *Currency* or by providing a string representation of a money amount.

1. Form

Parameters

- **amount** – money amount (gets rounded to a Decimal according to smallest fraction of currency)
- **currency** – money unit

amount must convertable to a *decimalfp.Decimal*, it can also be given as a string.

Raises

- **TypeError** – *amount* can not be converted to a Decimal number
- **ValueError** – no currency given

2. Form

Parameters

- **mStr** – unicode string representation of a money amount (incl. currency symbol)
- **currency** – the money's unit (optional)

mStr must contain a numerical value and a currency symbol, separated atleast by one blank. Any surrounding white space is ignored. If *currency* is given in addition, the resulting money's currency is set to this currency and its amount is converted accordingly, if possible.

Returns
Money instance

Raises

- **TypeError** – amount given in *mStr* can not be converted to a Decimal number
- **ValueError** – no currency given
- **TypeError** – a byte string is given that can not be decoded using the standard encoding
- **ValueError** – given string does not represent a *Money* amount
- ***IncompatibleUnitsError*** – the currency derived from the symbol given in *mStr* can not be converted to given *currency*

MoneyMeta.**register_currency**(*iso_code: str*) → *Currency*

Register the currency with code *iso_code* from ISO 4217 database.

Parameters
iso_code – ISO 4217 3-character code for the currency to be registered

Returns
registered currency

Raises
ValueError – currency with code *iso_code* not in database

MoneyMeta.**new_unit**(*symbol: str*, *name: str | None = None*, *minor_unit: int | None = None*, *smallest_fraction: Real | str | None = None*) → *Currency*

Create, register and return a new *Currency* instance.

Parameters

- **symbol** – symbol of the currency (should be a ISO 4217 3-character code, if possible)
- **name** – name of the currency
- **minor_unit** – amount of minor unit (as exponent to 10), optional, defaults to precision of smallest fraction, if that is given, otherwise to 2
- **smallest_fraction** – smallest fraction available for the currency, optional, defaults to Decimal(10) ** -minor_unit. Can also be given as a string, as long as it is convertable to a Decimal.

Raises

- **TypeError** – given *symbol* is not a string
- **ValueError** – no *symbol* was given
- **TypeError** – given *minor_unit* is not an Integral number
- **ValueError** – given *minor_unit* < 0
- **ValueError** – given *smallest_fraction* can not be converted to a Decimal
- **ValueError** – given *smallest_fraction* not > 0
- **ValueError** – 1 is not an integer multiple of given *smallest_fraction*
- **ValueError** – given *smallest_fraction* does not fit given *minor_unit*

__init__()

__new__(*amount: Real | str*, *unit:* Unit | *None = None*) → <class 'quantity.Quantity'>

Create new *Quantity* instance.

property currency: *Currency*

The money's currency, i.e. its unit.

class ExchangeRate

Basic representation of a conversion factor between two currencies.

Parameters

- **unit_currency** – currency to be converted from, aka base currency
- **unit_multiple** – amount of base currency (must be equal to an integer)
- **term_currency** – currency to be converted to, aka price currency
- **term_amount** – equivalent amount of term currency

unit_currency and *term_currency* can also be given as 3-character ISO 4217 codes of already registered currencies.

unit_multiple must be > 1. It can also be given as a string, as long as it is convertable to an Integral.

term_amount can also be given as a string, as long as it is convertable to a number.

Example:

1 USD = 0.9683 EUR => ExchangeRate('USD', 1, 'EUR', '0.9683')

unit_multiple and *term_amount* will always be adjusted so that the resulting unit multiple is a power to 10 and the resulting term amounts magnitude is >= -1. The latter will always be rounded to 6 decimal digits.

Raises

- **ValueError** – non-registered / unknown symbol given for a currency
- **TypeError** – value of type other than *Currency* or string given for a currency
- **ValueError** – currencies given are identical
- **ValueError** – unit multiple is not equal to an integer or is not >= 1
- **ValueError** – term amount is not >= 0.000001
- **ValueError** – unit multiple or term amount can not be converted to a Decimal

__eq__(*other: Any*) → bool

self == other

Parameters
other – object to compare with

Returns
True if other is an instance of ExchangeRate and self.quotation == other.quotation, False otherwise

__hash__() → int

hash(self)

__init__(*unit_currency:* Currency | *str*, *unit_multiple: Rational*, *term_currency:* Currency | *str*, *term_amount: Rational* | *float* | *str*) → None

__mul__(*other: <class 'quantity.money.Money'>*) → <class 'quantity.money.Money'>

__mul__(*other:* ExchangeRate) → *ExchangeRate*

__mul__(*other: <class 'quantity.Quantity'>*) → <class 'quantity.Quantity'>

self * other

1. Form

Parameters
other – money amount to multiply with

Returns
Money equivalent of *other* in term currency

Raises
ValueError – currency of *other* is not equal to unit currency

2. Form

Parameters
other – exchange rate to multiply with

Returns
"triangulated" exchange rate

Raises
ValueError – unit currency of one multiplicant does not equal the term currency of the other multiplicant

3. Form

Parameters
other – quantity to multiply with

The type of *other* must be a sub-class of `Quantity` derived from *Money* divided by some other sub-class of `Quantity`.

Returns
equivalent of *other* in term currency

Raises
ValueError – resulting unit is not defined

__rtruediv__(*other: <class 'quantity.money.Money'>*) → <class 'quantity.money.Money'>

__rtruediv__(*other: <class 'quantity.Quantity'>*) → <class 'quantity.Quantity'>

other / self

1. Form

Parameters
other – money amount to divide

Returns
equivalent of *other* in unit currency

Raises
ValueError – currency of *other* is not equal to term currency

2. Form

Parameters
other – quantity to divide

The type of *other* must be a sub-class of `Quantity` derived from *Money* divided by some other sub-class of `Quantity`.

Returns
equivalent of *other* in unit currency

Raises
QuantityError – resulting unit is not defined

__truediv__(*other:* ExchangeRate) → *ExchangeRate*

self / other

Parameters
other – exchange rate to divide with

Returns
"triangulated" exchange rate

Raises
ValueError – unit currencies of operands not equal and term currencies of operands not equal

inverted() → *ExchangeRate*

Return inverted exchange rate.

property inverse_quotation: Tuple[*Currency*, *Currency*, Rational]

Tuple of term currency, unit currency and inverse rate.

property inverse_rate: Rational

Inverted rate, i.e. relative value of unit currency to term currency.

property quotation: Tuple[*Currency*, *Currency*, Rational]

Tuple of unit currency, term currency and rate.

property rate: Rational

Relative value of term currency to unit currency.

property term_currency: *Currency*

Currency to be converted to, aka price currency.

property unit_currency: *Currency*

Currency to be converted from, aka base currency.

class MoneyConverter

Converter for money amounts.

Money converters can be used to hold different exchange rates. They can be registered with the class *Money* in order to support implicit conversion of money amounts from one currency into another.

Parameters

- **base_currency** – currency used as reference currency
- **get_dflt_effective_date** – a callable without parameters that must return a date which is then used as default effective date in *MoneyConverter.get_rate()* (default: *datetime.date.today*)

__call__(*money_amnt: <class 'quantity.money.Money'>*, *to_currency: ~quantity.money.Currency*, *effective_date: ~datetime.date | None = None*) → Rational

Convert a money amount in one currency to the equivalent amount for another currency.

Parameters

- **money_amnt** – money amount to be converted
- **to_currency** – currency for which the equivalent amount is to be returned
- **effective_date** – date for which the exchange rate to be used must be effective (default: None)

If *effective_date* is not given, the return value of the callable given as *get_dflt_effective_date* to *MoneyConverter* is used as reference (default: today).

Returns
amount equiv so that equiv * to_currency == money_amnt

Raises
UnitConversionError – exchange rate not available

__enter__() → *MoneyConverter*

Register self as converter in *Money*.

__exit__(**args: Tuple[Any, ...]*) → None

Unregister self as converter in *Money*.

__init__(*base_currency:* Currency, *get_dflt_effective_date: Callable[[], date] | None = None*) → None

get_rate(*unit_currency:* Currency, *term_currency:* Currency, *effective_date: date | None = None*) → *ExchangeRate* | None

Return exchange rate from *unit_currency* to *term_currency* that is effective for *effective_date*.

Parameters

- **unit_currency** – currency to be converted from
- **term_currency** – currency to be converted to
- **effective_date** – date at which the rate must be effective (default: None)

If *effective_date* is not given, the return value of the callable given as *get_dflt_effective_date* to *MoneyConverter* is used as reference (default: today).

Returns

exchange rate from *unit_currency* to *term_currency* that is
effective for *effective_date*, *None* if there is no such rate

update(*validity: ValidityT*, *rate_specs: Iterable[RateSpecT]*) → None

Update the exchange rate dictionary used by the converter.

Parameters

- **validity** – specifies the validity period of the given exchange rates
- **rate_specs** – list of entries to update the converter

validity can be given in different ways:

If *None* is given, the validity of the given rates is not restricted, i. e. they are used for all times (“constant rates”).

If an *int* (or a string convertable to an *int*) is given, it is interpreted as a year and the given rates are treated as valid for that year ("yearly rates").

If a tuple of two *int`s (or two strings convertable to an `int*) or a string in the form 'YYYY-MM' is given, it is interpreted as a combination of a year and a month, and the given rates are treated as valid for that month ("monthly rates").

If a *date* or a string holding a date in ISO format ('YYYY-MM-DD') is given, the rates are treated as valid just for that date ("daily rates").

The type of validity must be the same in recurring updates.

Each entry in *rate_specs* must be comprised of the following elements:

- ***term_currency*** **(Union[Currency, str]): currency of equivalent** amount, aka price currency
- ***term_amount*** **(Union[Rational, float, str]): equivalent amount of** term currency
- *unit_multiple* (Rational): amount of base currency

validity and *term_currency* are used together as the key for the internal exchange rate dictionary.

Raises

- **ValueError** – invalid date given for *validity*
- **ValueError** – invalid year / month given for *validity*
- **ValueError** – invalid year given for *validity*
- **ValueError** – unknown value given for *validity*
- **ValueError** – different types of validity period given in subsequent calls

property base_currency: *Currency*

The currency used as reference currency.

4.4 Functions

get_currency_info(*iso_code: str*) → Tuple[str, int, str, int, List[str]]

Return infos from ISO 4217 currency database.

Parameters

iso_code – ISO 4217 3-character code for the currency to be looked-up

Returns

3-character code, numerical code, name, minor unit and list of countries which use the currency as functional currency

Raises

ValueError – currency with code *iso_code* not in database

Note: The database available here does only include entries from ISO 4217 which are used as functional currency, not those used for bond markets, noble metals and testing purposes.

---

genindex

PYTHON MODULE INDEX

q

INDEX

Symbols

A

B

C