unithandler

Release 1.6.0

Oct 07, 2021

Contents:

CHAPTER 1

unithandler package

1.1 unithandler.base module

class unithandler.base.**Constant**(*value: Union[float, int, str, UnitFloat, UnitInt], unit: Union[str, dict, unithandler.base.Unit] = '', uncertainty: float = None, scale_representation: bool = None*)

Bases: *unithandler.base.UnitFloat*

A class for storing a constant value with a unit. The value will be immutable except by accessing protected variables.

Parameters

- **value** (*float*) – the value to store
- **unit** (*str*) – unit for the value
- **uncertainty** (*float*) – Uncertainty in the float value (this will be interpreted as a +/- value with the same prefix as the incoming value).
- **scale_representation** (*bool*) – Whether to scale the representation of the value. Set this to False to disable value scaling.

real

Real numbers are their real component.

class unithandler.base.**Unit**(*unit: Union[str, dict, Unit] = '', sep: str = None, use_unicode: bool = None*)

Bases: object

A class for storing and managing units. This class is structured with the intent of being subclassed to provide a meaningful representation of a unit to an object. The *unit* attribute provides access to an appropriately formatted representation of the unit, and the units are stored in dictionary format along with the power associated with that unit, so that the unit may be easily modified without redefinition.

Parameters

- **or dict unit** (*str*) – Unit for the value. This may be a dictionary of units with *{unit: power, ... }* format for more complicated unit expressions. If a string is provided, an attempt at interpreting the unit will be made.
- **sep** (*str*) – separator for string representation
- **use_unicode** – whether to use unicode characters for unit representations

Examples

```
>>> uni = Unit('m')
>>> uni.unit
'm'
```

A variety of unit conventions are supported.

```
>>> uni = Unit('m/s')
>>> uni.unit
'm·s⁻¹'
>>> uni = Unit('m s-1')
>>> uni.unit
'm·s⁻¹'
```

The separator used in unit representation may be modified using the *sep* attribute.

```
>>> uni.sep = '*'
>>> uni.unit
'm*s⁻¹'
```

The unit of a *Unit* instance can be modified using multiplication or division operations. Augmented multiplication and division operations are also supported.

```
>>> uni * 's'
m
>>> uni * Unit('kg')
m·s·kg
>>> uni / 's'
m·s⁻²
```

Division and multiplication operations are supported for straightforward unit assignments. An instance of *UnitFloat* or *UnitInt* is returned depending on the type handed operated upon. This can be a convenient way of applying the same unit to several values.

```
>>> vel = 10 * uni  # uni * 10 performs the same effective operation
>>> vel
10 m·s⁻¹
>>> type(vel)
<class 'unithandler.base.UnitInt'>
>>> g = 9.8 * Unit('m/s2')
>>> g
9.8 m·s⁻²
>>> type(g)
<class 'unithandler.base.UnitFloat'>
```

In/Equality comparison is supported for unit checks. The comparison supports several types: another *Unit* instance, *dict*, or *str*. Comparison is also available in subclasses of *Unit* using the *unit_equality* method. Other comparisons such as less than or greater than are invalid and therefore unsupported.

```
>>> uni == {'m': 1, 's': -1}
True
>>> uni == 'm/s'
True
>>> uni == Unit('m/s')
True
```

Unit instances may also be queried whether they contain the specified unit(s) or the inverse of those units.

```
>>> uni.contains_unit('m')
True
>>> uni.contains_unit('m/s2')
False
>>> uni.contains_inverse_unit('s')
True
```

USE_UNICODE = False

contains_inverse_unit(*other: Union[str, dict, Unit]*) → bool
Checks whether the units in the instance contain the inverse of the specified unit.

Parameters **dict, Unit other** (*str,*) – other units

Returns whether the other unit is conatined in the instance

Return type bool

contains_unit(*other: Union[str, dict, Unit]*) → bool
Checks whether the units in the instance contain the specified unit.

Parameters **dict, Unit other** (*str,*) – other units

Returns whether the other unit is conatined in the instance

Return type bool

create_string_representation(*use_unicode: bool = None*) → str
Creates a string representation of the unit.

Parameters **use_unicode** – whether to use unicode characters in the representation (if not specified, the value of USE_UNICODE is used)

Returns string representation as specified

inverse() → dict
Returns the inverse of the units

static parse_unit_to_unit_dict(*unit: Union[str, dict, Unit]*) → dict
Parses a unit and returns a dictionary of unit: power values. Performs type assertions for interpretable types and raises a TypeError if the incoming values is not interpretable.

Parameters **unit** – incoming unit

Raises **TypeError** – if the incoming unit is not interpretable as a unit

Returns dictionary of unit: power values indicated by the unit

sep = '·'

unit
string representation of the unit

unit_equality(*other: Union[Unit, dict, str]*) → bool
Compares the units stored in the instance to the value provided.

Parameters **dict, Unit other** (*str,*) – other units

Returns whether the supplied units are identical to the stored units

Return type bool

exception unithandler.base.**UnitError**(*msg*)

Bases: Exception

An exception raised on unit discrepancies or mismatches

Parameters **msg** – message to supply

class unithandler.base.**UnitFloat**(*value: Union[float, int, str, UnitFloat, UnitInt], unit: Union[str, dict, unithandler.base.Unit] = '', si_prefix: str = None, stored_prefix: str = None, uncertainty: float = None, scale_representation: bool = None, sep: str = None*)

Bases: *unithandler.base.Unit*, numbers.Real

A float mimic that stores a value, it's unit, and allows specification of both the incoming and stored SI prefix. When a string or representation of this class is called, the optimal representation of the value is returned. This true stored value (e.g. when *float()* or *int()* is called on a *UnitFloat* instance is determined by the *prefix* attribute.

Parameters

- **value** – value to store
- **or dict unit** (*str*) – Unit for the value. This may be a dictionary of units with *{unit: power, ...}* format for more complicated unit expressions. If a string is provided, an attempt at interpreting the unit will be made.
- **si_prefix** – SI prefix for the incoming value
- **stored_prefix** – preferred SI prefix for stored value. The stored value will be scaled to reflect this prefix.
- **uncertainty** (*float*) – Uncertainty in the float value (this will be interpreted as a +/- value with the same prefix as the incoming value).
- **scale_representation** (*bool*) – Whether to scale the representation of the value. Set this to False to disable value scaling.
- **sep** – separator to use for unit string

Supported operations

- All normal, reversed, and augmented numeric Python operations (*, /, +, -, **, //, %, *=, /=, +=, -=, **=).
- Comparison to numeric values (or other *Unit* subclasses)
- Unary operations (*neg*, *pos*, *abs*, *round*, etc.)
- *pickle* package
- *copy* package
- *math* package

Behaviour of numeric operations

All numeric Python operators are supported in normal, reversed, and augmented form. Comparisons to values, as well as bitwise operations are also supported.

If the other value is *not* a *Unit* subclass:

- For addition and subtraction, it will be assumed to have the same unit as the *UnitFloat* value.

- For multiplication and division, the other value will be interpreted as a scalar. It will be assumed that the other value has the same prefix scalar as the *UnitFloat* value. This may lead to unintended scaling of the resulting value, and can be controlled by creating two *UnitFloat* instances prior to operation, or by performing the operation on two *float* values, then converting to *UnitFloat*.

If the other value is a *Unit* subclass:

- For addition and subtraction, the units of the other value will be compared, and if they are unequal, a *UnitError* will be raised.
- For multiplication and division, the units of the first value will be modified accordingly. If multiplied, the powers will be increased by the other unit's powers. If divided, the powers will be decreased by the other unit's power.

Examples

```
>>> val = UnitFloat(0.2, 'L')
>>> val  # the optimal representation of this will be automatically determined
200 mL
>>> float(val)  # but the true value will reflect the prefix of the UnitFloat␣
↪instance
0.2
```

If the value provided on instantiation is scaled to a particular SI prefix, it may be specified during instantiation with the *incoming_prefix* keyword argument.

```
>>> vol = 2.  # represents 2 mL
>>> uf_vol = UnitFloat(vol, 'L', si_prefix='m')  # equivalent to UnitFloat(2., 'L
↪', 'm')
>>> uf_vol
2 mL
>>> float(uf_vol)  # the value has been scaled to have no SI prefix by default
0.002
```

The desired prefix for the stored value of the *UnitFloat* instance may be set during instantiation using the *stored_prefix* keyword argument, or modified after instantiation by changing the *prefix* attribute. The prefix may be changed to any SI prefix, and will scale the stored float value, but not affect the representation. The stored value scaled to a specific prefix may be conveniently accessed using the *specific_prefix* method.

```
>>> uf_vol.prefix
''
# this is equivalent to calling UnitFloat(vol, 'L', 'm', stored_prefix='m') or␣
↪UnitFloat(vol, 'L', 'm', 'm')
>>> uf_vol.prefix = 'm'  # the prefix may be changed to any SI prefix
>>> uf_vol  # the representation will remain unchanged
2 mL
>>> float(uf_vol)  # the stored value will be scaled
2.0
>>> uf_vol.specific_prefix('u')
2000.0
```

For convenience, string values with units may be provided and automatically interpreted.

```
>>> from_string = UnitFloat('9.8 m/s2')
>>> from_string
9.8 m·s²
>>> float(from_string)
9.8
```

(continues on next page)

(continued from previous page)

```
>>> from_string.units
{'m': 1, 's': -2}
```

Additionally, SI prefixes may be included at the beginning of the unit and will be used to automatically scale the value.

```
>>> auto_scaled = UnitFloat(1.5, 'ug')
>>> auto_scaled
1.5 ug
>>> float(auto_scaled)  # the value was automatically interpreted and stored as␣
↪micro
1.5
>>> auto_scaled.prefix
'u'
>>> auto_scaled_2 = UnitFloat(250., 'ug', stored_prefix='m')  # the stored prefix␣
↪may still be overridden
>>> auto_scaled_2
250 ug
>>> float(auto_scaled_2)
0.25
```

Operations on *UnitFloat* instances will result in a new *UnitFloat* instance. See the *Behaviour of numeric operations* section above for the assumptions and prescribed behaviour in Python operations on *UnitFloat* instances.

```
>>> newval = val + 0.4
>>> newval
600 mL
>>> type(newval)
<class 'unithandler.base.UnitFloat'>
>>> val - 0.05
150 mL
>>> 0.4 + val
600 mL
>>> 1.0 - val
800 mL
>>> val * 2
400 mL
>>> val / 2
100 mL
>>> 2. * val
400 mL
>>> 1 / val
5 L^1
```

If the other value is a *Unit* or a *UnitInt*, the units will be compared the appropriate operation will be performed, returning a *UnitFloat* instance.

```
>>> otherval = UnitInt(1., 'L')
>>> val + otherval
1.2 L
>>> val - otherval
-800 mL
>>> val + UnitInt(1., 'm')  # addition/subtraction are not supported for␣
↪mismatched units
UnitError: The units of the two UnitFloat instances are mismatched: L != m
>>> val * otherval
```

(continues on next page)

(continued from previous page)

```
200 mL²
>>> val / otherval
200 m  # now a unitless number
```

If the other value is a *UnitFloat* value, both values will be scaled to have no prefix prior to operation, and a *UnitFloat* with the parent's stored_prefix will be returned.

```
>>> val.prefix = 'k'
>>> float(val)
0.0002
>>> val2 = UnitFloat(500000, 'L', 'n', 'n')
>>> val2
500 uL
>>> float(val2)
>>> val3 = val + val2  # matches the prefixes and returns a prefixless UnitFloat
700 uL
>>> float(val3)
0.2005
>>> val4 = UnitFloat(50, 'mol', 'm', 'm') / UnitFloat(100, 'L', 'm')
>>> val4
500 mmol·L¹
>>> val4.prefix  # the result inherits the stored prefix of the numerator
'm'
>>> val5 = UnitFloat(0.5, 'mol/L', 'm', 'n') * UnitFloat(50., 'L', 'm')
>>> val5
25 umol
>>> val5.prefix  # the result inherits the stored prefix of the value operated on
'n'
```

DEFAULT_SI_PREFIX = ''

DEFAULT_STORED_PREFIX = ''

OPENPYXL_FORMAT_CELL = True

as_constant() → unithandler.base.Constant
Returns the value as a Constant instance

as_integer_ratio()
returns the integer ratio of the float value

change_return_prefix(*newprefix=''*)
Changes the return prefix to the specified prefix.

Parameters **newprefix** (*str*) – new SI prefix

fromhex(*s: str*)
converts a hexidecimal string to a float value

hex()
Converts the value to hexidecimal

imag
imaginary components are not currently supported

lower
The lower bound of the value (incorporating uncertainty)

optimal_representation() → str
Determines the optimal representation of the value.

Returns value, prefixed unit

Return type str

prefix
the SI prefix to store the value in (scales the stored float value)

real
Real numbers are their real component.

round(*ndigits=None*, *prefix=None*)
Rounds the instance to the nearest ndigits. This has the effect of rounding and reassigning the real attribute of the instance. For convenience, a prefix different from that of the instance may be provided. The Python round builtin is used to round the values.

Parameters

- **ndigits** – number of digits to round to
- **prefix** – prefix to use (if different from prefix of instance)

e.g. rounding a no-prefix instance at the micro level would be accomplished by

```
>>> val = UnitFloat(1.25, 'uL', stored_prefix='')
>>> val
1.25 uL
>>> val.round(1, 'u')  # round the first decimal place at the micro level
>>> val
1.2 uL
```

scale_representation = True

specific_prefix(*newprefix*) → float
Returns the value converted to the specific prefix

Parameters **newprefix** – new SI prefix

Returns value with the specific prefix

Return type float

specified_representation() → str
Returns the value represented in the specified prefix.

Returns value with unit

Return type str

stored_unit

uncertainty
The uncertainty in the value

upper
The upper bound of the value (incorporating uncertainty)

write_to_openpyxl_cell(*cell: openpyxl.cell.cell.Cell*)
Writes the value of the UnitInt to an openpyxl Cell. The value of the instance is written as the cell value and a number format is applied to the cell to display the unit.

Parameters **cell** – openpyxl cell

class unithandler.base.**UnitInt**(*value: Union[int, float, str, UnitFloat, UnitInt], unit: Union[str, dict, unithandler.base.Unit] = '', sep: str = None*)

Bases: numbers.Real, *unithandler.base.Unit*

An *int* mimic with a unit attribute.

Parameters

- **value** – value to store
- **or dict unit** (*str*) – Unit for the value. This may be a dictionary of units with *{unit: power, ...}* format for more complicated unit expressions. If a string is provided, an attempt at interpreting the unit will be made.

Supported operations

- All normal, reversed, and augmented numeric Python operations (*, /, +, -, **, //, %, *=, /=, +=, -=, **=).
- Bitwise operations
- Comparison to numeric values (or other *Unit* subclasses)
- Unary operations (*neg*, *pos*, *abs*, *round*, etc.)
- *pickle* package
- *copy* package
- *math* package

Behaviour of numeric operations

All numeric Python operators are supported in normal, reversed, and augmented form. Comparisons to values, as well as bitwise operations are also supported.

If the other value is *not* a *Unit* subclass:

- For addition and subtraction, it will be assumed to have the same unit as the *UnitInt* value.
- For multiplication and division, the other value will be interpreted as a scalar.

If the other value is a *Unit* subclass:

- For addition and subtraction, the units of the other value will be compared, and if they are unequal, a *UnitError* will be raised.
- For multiplication and division, the units of the first value will be modified accordingly. If multiplied, the powers will be increased by the other unit's powers. If divided, the powers will be decreased by the other unit's power.

Examples

```
>>> distance = UnitInt(50, 'm')
>>> distance
50 m
```

```
>>> new = distance + 50
100 m
>>> type(new)  # the new value is a new UnitInt instance
<class 'unithandler.base.UnitInt'>
```

Reversed operations will automatically return a *UnitInt* instance.

```
>>> 50 + distance
100 m
>>> distance - 25
25 m
>>> 25 - distance
-25 m
>>> distance + UnitInt(50, 'm/s')  # mismatched units
UnitError: The units of the two UnitInt instances are mismatched: m != m·s¹
```

Operations that result in non-integer values will automatically convert to UnitFloat

```
>>> new = distance + 50.5
>>> type(new)
<class 'unithandler.base.UnitFloat'>
```

```
>>> distance * 2
100 m
>>> distance / 2
25 m
>>> 100 / distance
2 m¹
>>> 2 * distance
100 m
>>> distance * UnitInt(2, 'm')  # multiplication by another Unit subclass
100 m²
>>> distance / UnitInt(25, 's')  # division by another Unit subclass
2 m·s¹
```

OPENPYXL_FORMAT_CELL = True

as_constant() → unithandler.base.Constant
Returns the value as a Constant instance

real
Real numbers are their real component.

write_to_openpyxl_cell(*cell: openpyxl.cell.cell.Cell*)
Writes the value of the UnitInt to an openpyxl Cell. The value of the instance is written as the cell value and a number format is applied to the cell to display the unit.

Parameters **cell** – openpyxl cell

unithandler.base.**adjust_unit**(*original: dict*, *incoming: dict*) → dict
Adjusts the original unit dictionary with the incoming dictionary

Parameters

- **original** (*dict*) – original dictionary
- **incoming** (*dict*) – adjusting dictionary

Returns adjusted dictionary

Return type dict

unithandler.base.**chewunit**(*unit: str*) → Tuple[str, dict]
Iterates through the provided unit, extracting and interpreting the blocks, then returning the unit minus the block.

Parameters **unit** (*str*) – unit to be interpreted

Returns remaining unit minus the block, interpreted block

unithandler.base.**expand_units**(*units: dict*) → dict
Expands any derived SI units into their true SI unit equivalents.

Parameters **units** (*dict*) – units dictionary

Returns expanded units

Return type dict

unithandler.base.**interpret_unit**(*unit: str*) → dict
Interprets a unit, converting it into unit: power associations

Parameters **unit** – unit to interpret

Returns unit dictionary

Return type dict

unithandler.base.**interpret_unit_block**(*unitblock: str*) → dict
Interprets a unit and breaks it into *unit: power* associations.

Parameters **unitblock** – unit to interpret

Returns dictionary of unit: power associations

Return type dict

unithandler.base.**prefix_from_power**(*power: int*) → str
Returns the SI prefix associated with the provided power.

Parameters **power** (*int*) – power

Returns SI prefix

Return type str

unithandler.base.**reduce_units**(*units: dict*) → dict
Checks for unit equivalency in SI derived units and returns the converted units dictionary.

Parameters **units** (*dict*) – dictionary of units

Returns converted dictionary

Return type dict

unithandler.base.**scale_value_by_prefix**(*value: Union[float, UnitFloat], target_prefix: str, current_prefix: str = ''*) → float
Converts a value to a different SI prefix.

Parameters

- **value** (*float*) – value to convert
- **target_prefix** (*str*) – target prefix to convert to
- **current_prefix** (*str*) – current prefix of the value

Returns scaled value

Return type float

unithandler.base.**to_superscript**(*val: int*) → str
Returns the integer value represented as a superscript string.

Parameters **val** (*int*) – value to represent

Returns superscript string

Return type str

1.2 unithandler.constants module

some predefined standard constants

1.3 unithandler.conversion module

module with pre-instatiated unit-conversion instances

1.4 unithandler.siunits module

module with pre-instantiated SI units

CHAPTER 2

Indices and tables

- genindex
- modindex
- search

Python Module Index

u

Index

T

U

W