decimalfp Documentation

Release 0.12.2

Michael Amrhein

Nov 19, 2021

CONTENTS

Contents:

CHAPTER

ONE

DECIMAL NUMBERS WITH FIXED-POINT ARITHMETIC

The package *decimalfp* provides a `Decimal` number type which can represent decimal numbers of (nearly) arbitrary magnitude and very large precision, i.e. with a very large number of fractional digits.

1.1 Usage

`decimalfp.Decimal` instances are created by giving a *value* (default: 0) and a *precision* (i.e the number of fractional digits, default: None).

```
>>> Decimal()
Decimal(0)
```

If *precision* is given, it must be of type *int* and >= 0.

```
>>> Decimal(5, 3)
Decimal(5, 3)
>>> Decimal(5555, -3)
ValueError: Precision must be >= 0.
```

If *value* is given, it must either be a string, an instance of *numbers.Integral*, *number.Rational* (for example *fractions.Fraction*), *decimal.Decimal*, a finite instance of *numbers.Real* (for example *float*) or be convertable to a *float* or an *int*.

The value is always adjusted to the given precision or the precision is calculated from the given value, if no precision is given.

```
>>> Decimal(b'12.345')
Decimal('12.345')
>>> Decimal(u'12.345')
Decimal('12.345')
>>> Decimal(u'12.345', 5)
Decimal('12.345', 5)
>>> Decimal(15, 4)
Decimal(15, 4)
>>> Decimal(10 ** 23)
Decimal(100000000000000000000000)
>>> Decimal(10 ** 48)
Decimal(100000000000000000000000000000000000000000000000)
>>> Decimal(2 ** 48)
Decimal(281474976710656)
>>> Decimal(2 ** 98)
```

(continues on next page)

(continued from previous page)

```
Decimal(3169126500570573503741758013 44)
>>> Decimal(Fraction(7, 56))
Decimal('0.125')
>>> Decimal(Fraction(8106479329266893, 4503599627370496), 7)
Decimal('1.8', 7)
>>> Decimal(1.8, 7)
Decimal('1.8', 7)
>>> Decimal(decimal.Decimal('-19.26', 5), 3)
Decimal('-19.26', 3)
```

When the given *precision* is lower than the precision of the given *value*, the result is rounded, according to the current default rounding mode (which itself defaults to ROUND_HALF_EVEN).

```
>>> Decimal(u'12.345', 2)
Decimal('12.34')
>>> Decimal(u'12.3456', 3)
Decimal('12.346')
>>> Decimal(0.2, 3)
Decimal('0.2', 3)
>>> Decimal(0.2, 17)
Decimal('0.20000000000000001')
>>> Decimal(0.2, 55)
Decimal('0.2000000000000000011102230246251565404236316680908203125', 55)
```

When no *precision* is given and the given *value* is a *float* or a *numbers.Rational* (but no `Decimal`), the `Decimal` constructor tries to convert *value* exactly. But this is done only up a fixed limit of fractional digits (imposed by the implementation, currently 65535). If *value* can not be represented as a `Decimal` within this limit, an exception is raised.

```
>>> Decimal(Fraction(1, 7))
ValueError: Can't convert Fraction(1, 7) exactly to Decimal.
```

`Decimal` does not deal with infinity, division by 0 always raises a *ZeroDivisionError*. Likewise, infinite instances of type *float* or *decimal.Decimal* can not be converted to `Decimal` instances. The same is true for the 'not a number' instances of these types.

1.2 Computations

When importing *decimalfp*, its `Decimal` type is registered in Pythons numerical stack as *number.Rational*. It supports all operations defined for that base class and its instances can be mixed in computations with instances of all numeric types mentioned above.

All numerical operations give an exact result, i.e. they are not automatically constraint to the precision of the operands or to a number of significant digits (like the floating-point *Decimal* type from the standard module *decimal*). When the result can not exactly be represented by a `Decimal` instance within the limit of fractional digits, an instance of *fractions.Fraction* is returned.

1.2.1 Addition and subtraction

Adding or subtracting *Decimal* instances results in a *Decimal* instance with a precision equal to the maximum of the precisions of the operands.

```
>>> Decimal('7.3') + Decimal('8.275')
Decimal('15.575')
>>> Decimal('-7.3', 4) + Decimal('8.275')
Decimal('0.975', 4)
```

In operations with other numerical types the precision of the result is at least equal to the precision of the involved *Decimal* instance, but may be greater, if neccessary. If the needed precision exceeds the limit of fractional digits, an instance of *fractions.Fraction* is returned.

```
>>> 0.25 + Decimal(3)
Decimal('3.25')
>>> 0.25 - Decimal(-3, 5)
Decimal('3.25', 5)
>>> 0.725 + Decimal('3')
Decimal('3.724999999999999977795539507496869191527366638183593 75')
>>> Decimal('3') + Fraction(1, 7)
Fraction(22, 7)
```

1.2.2 Multiplication and division

Multiplying *Decimal* instances results in a *Decimal* instance with precision equal to the sum of the precisions of the operands.

```
>>> Decimal('5.000') * Decimal('2.5')
Decimal('12.5', 4)
```

Dividing *Decimal* instances results in a *Decimal* instance with precision at least equal to max(0, numerator.precision - denominator.precision), but may be greater, if needed.

```
>>> Decimal('5.2000') / Decimal('2.5')
Decimal('2.08', 3)
>>> Decimal('5.2003') / Decimal('2.5')
Decimal('2.08012')
```

In operations with other numerical types the precision of the result is at least equal to the precision of the involved *Decimal* instance, but may be greater, if neccessary. If the needed precision exceeds the limit of fractional digits, an instance of *fractions.Fraction* is returned.

```
>>> 3 * Decimal('7.5')
Decimal('22.5')
>>> Decimal(5) * 0.25
Decimal('1.25')
>>> Decimal('3') * Fraction(1, 7)
Fraction(3, 7)
```

1.2.3 Rounding

Decimal supports rounding via the built-in function *round*.

Note: In Python 3.x the function *round* uses the rounding mode ROUND_HALF_EVEN and returns an *int* when called with one argument, otherwise the same type as the number to be rounded.

```
>>> round(Decimal('12.345'))
12
>>> round(Decimal('12.345'), 2)
Decimal('12.34')
>>> round(Decimal('1234.5'), -2)
Decimal(1200)
```

In addition, via the method `adjusted()` a *Decimal* with a different precision can be derived, supporting all rounding modes defined by the standard library module *decimal*.

The rounding modes defined in *decimal* are wrapped into the Enum *ROUNDING*.

```
>>> d = Decimal('12.345')
>>> d.adjusted(2)           # default rounding mode is ROUND_HALF_EVEN !
Decimal('12.34')
>>> d.adjusted(2, ROUNDING.ROUND_HALF_UP)
Decimal('12.35')
>>> d.adjusted(1, ROUNDING.ROUND_UP)
Decimal('12.4')
```

For the details of the different rounding modes see the documentation of the standard library module *decimal*.

round and *adjusted* only allow to round to a quantum that's a power to 10. The method `quantize()` can be used to round to any quantum and it does also support all rounding modes mentioned above.

```
>>> d = Decimal('12.345')
>>># equivalent to round(d, 2) or d.adjusted(2)
>>># (default rounding mode ROUNDING.ROUND_HALF_EVEN):
>>> d.quantize(Decimal('0.01'))
Decimal('12.34')
>>> d.quantize(Decimal('0.05'))
Decimal('12.35')
>>> d.quantize('0.6')
Decimal('12.6')
>>> d.quantize(4)
Decimal('12')
```

1.3 Class *Decimal*

DecValueT = Union[SupportsInt, SupportsFloat, SupportsAsIntegerRatio, str]

class Decimal

Decimal number with a given number of fractional digits.

Parameters

- **value** – numerical value (default: None)
- **precision** – number of fractional digits (default: None)

If *value* is given, it must either be a string, an instance of *int*, *number.Rational* (for example *fractions.Fraction*), *decimal.Decimal*, a finite instance of *numbers.Real* (for example *float*) or be convertable to a *float* or an *int*.

If a string is given as value, it must be a string in one of two formats:

- [+|-]<int>[.<frac>][<e|E>[+|-]<exp>] or
- [+|-].<frac>[<e|E>[+|-]<exp>].

If given value is *None*, Decimal(0) is returned.

Returns

Decimal **instance derived from *value* according** to *precision*

The value is always adjusted to the given precision or the precision is calculated from the given value, if no precision is given.

Raises

- **TypeError** – *precision* is given, but not of type *int*.
- **TypeError** – *value* is not an instance of the types listed above and not convertable to *float* or *int*.
- **ValueError** – *precision* is given, but not >= 0.
- **ValueError** – *precision* is given, but not <= *MAX_DEC_PRECISION*.
- **ValueError** – *value* can not be converted to a *Decimal* (with a number of fractional digits <= *MAX_DEC_PRECISION*).

Decimal instances are immutable.

__abs__() → *Decimal*
abs(self)

__add__(*other: decimalfp.SupportsAsIntegerRatio*) → numbers.Rational
self + other

__bytes__() → bytes
bytes(self)

__ceil__() → int
math.ceil(self)

__eq__(*other: Any*) → bool
self == other

__floor__() → int
math.floor(self)

__format__(*fmt_spec: str*) → str
Return *self* converted to a string according to *fmt_spec*.

Parameters **fmt_spec** – a standard format specifier for a number

Returns *str* – *self* converted to a string according to *fmt_spec*

__ge__(*other: Any*) → bool
self >= other

__gt__(*other: Any*) → bool
self > other

__hash__() → int
hash(self)

__init__()

__int__() → int
math.trunc(self)

__le__(*other: Any*) → bool
self <= other

__lt__(*other: Any*) → bool
self < other

__mul__(*other: decimalfp.SupportsAsIntegerRatio*) → numbers.Rational
self * other

__neg__() → *Decimal*
-self

static __new__(*cls*, *value: DecValueT = None*, *precision: Optional[Integer] = None*) → *Decimal*
Create and return new *Decimal* instance.

__pos__() → *Decimal*
+self

__pow__(*other: SupportsIntOrFloat*, *mod: Optional[Any] = None*) → numbers.Complex
self ** other

If other is an integer (or a Rational with denominator = 1), the result will be a Decimal or a Fraction. Otherwise, the result will be a float or a complex.

mod must always be None (otherwise a *TypeError* is raised).

__radd__(*other: decimalfp.SupportsAsIntegerRatio*) → numbers.Rational
other + self

__repr__() → str
repr(self)

__rmul__(*other: decimalfp.SupportsAsIntegerRatio*) → numbers.Rational
other * self

__round__(*ndigits: Optional[int] = None*) → Union[int, *Decimal*]
round(self [, n_digits])

Round *self* to a given precision in decimal digits (default 0). *n_digits* may be negative.

This method is called by the built-in *round* function. It returns an *int* when called with one argument, otherwise a `Decimal`.

__rsub__(*other: decimalfp.SupportsAsIntegerRatio*) → numbers.Rational
other - self

__rtruediv__(*other: decimalfp.SupportsAsIntegerRatio*) → numbers.Rational
other / self

__str__() → str
str(self)

__sub__(*other: decimalfp.SupportsAsIntegerRatio*) → numbers.Rational
self - other

__truediv__(*other: decimalfp.SupportsAsIntegerRatio*) → numbers.Rational
self / other

__trunc__() → int
math.trunc(self)

adjusted(*precision: Optional[int] = None*, *rounding: Optional[*ROUNDING*] = None*) → *Decimal*
Return adjusted copy of *self*.

Parameters

- **precision** – number of fractional digits (default: None)
- **rounding** – rounding mode (default: None)

Returns

Decimal **instance derived from *self*, adjusted** to the given *precision*, using the given *rounding* mode

If no *precision* is given, the result is adjusted to the minimum precision preserving x == x.adjusted().

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

If the given *precision* is less than the precision of *self*, the result is rounded and thus information may be lost.

as_fraction() → fractions.Fraction
Return an instance of *Fraction* equal to *self*.

Returns the *Fraction* with the smallest positive denominator, whose ratio is equal to *self*.

as_integer_ratio() → Tuple[int, int]
Return a pair of integers whose ratio is equal to *self*.

Returns the pair of numerator and denominator with the smallest positive denominator, whose ratio is equal to *self*.

as_tuple() → Tuple[int, int, int]
Return a tuple (sign, coeff, exp) equivalent to *self*.

self == sign * coeff * 10 ** exp.

sign in (-1, 0, 1), for self < 0, = 0, > 0. coeff = 0 only if self = 0.

classmethod from_decimal(*d: Union[*Decimal*, int, decimal.Decimal]*) → *Decimal*
Convert a finite decimal number to a *Decimal*.

Parameters **d** – decimal number to be converted to a *Decimal*

Returns *Decimal* instance derived from *d*

Raises

- **TypeError** – *d* is not an instance of the types listed above.

- **ValueError** – *d* can not be converted to a *Decimal*.

classmethod from_float(*f: Union[float, int]*) → *Decimal*
Convert a finite float (or int) to a *Decimal*.

Parameters **f** – number to be converted to a *Decimal*

Returns *Decimal* instance derived from *f*

Raises

- **TypeError** – *f* is neither a *float* nor an *int*.
- **ValueError** – *f* can not be converted to a *Decimal* with a precision <= *MAX_DEC_PRECISION*.

Beware that Decimal.from_float(0.3) != Decimal('0.3').

classmethod from_real(*r: numbers.Real*, *exact: bool = True*) → *Decimal*
Convert a finite Real number to a *Decimal*.

Parameters

- **r** – number to be converted to a *Decimal*
- **exact** – *True* if *r* shall exactly be represented by the resulting *Decimal*

Returns *Decimal* instance derived from *r*

Raises

- **TypeError** – *r* is not an instance of *numbers.Real*.
- **ValueError** – *exact* is *True* and *r* can not exactly be converted to a *Decimal* with a precision <= *MAX_DEC_PRECISION*.

If *exact* is *False* and *r* can not exactly be represented by a *Decimal* with a precision <= *MAX_DEC_PRECISION*, the result is rounded to a precision = *MAX_DEC_PRECISION*.

quantize(*quant: decimalfp.SupportsAsIntegerRatio*, *rounding: Optional[ROUNDING] = None*) → numbers.Rational
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 rounding mode is used.

Returns

Decimal* instance that is the integer multiple of *quant closest to *self* (according to *rounding* mode); if result can not be represented as *Decimal*, an instance of *Fraction* is returned

Raises

- **TypeError** – *quant* is not a number or does not support *as_integer_ratio*
- **ValueError** – *quant* is not convertable to a *Rational*

property denominator: int
Return the normalized denominator of 'self'.

I. e. the smallest positive denominator from the pairs of integers, whose ratio is equal to *self*.

property imag: int
Return imaginary part of *self*.

Returns 0 (Real numbers have no imaginary component).

property magnitude: int
Return magnitude of *self* in terms of power to 10.

I.e. the largest integer exp so that 10 ** exp <= self.

property numerator: int
Return the normalized numerator of *self*.

I. e. the numerator from the pair of integers with the smallest positive denominator, whose ratio is equal to *self*.

property precision: int
Return precision of *self*.

property real: *Decimal*
Return real part of *self*.

Returns *self* (Real numbers are their real component).

1.4 Rounding modes

Decimal supports rounding modes equivalent to those defined by the standard library module *decimal*: ROUND_DOWN, ROUND_UP, ROUND_HALF_DOWN, ROUND_HALF_UP, ROUND_HALF_EVEN, ROUND_CEILING, ROUND_FLOOR and ROUND_05UP.

The rounding modes are wrapped into the Enum *ROUNDING*.

class ROUNDING
Enumeration of rounding modes.

ROUND_05UP = 1
Round away from zero if last digit after rounding towards zero would have been 0 or 5; otherwise round towards zero.

ROUND_CEILING = 2
Round towards Infinity.

ROUND_DOWN = 3
Round towards zero.

ROUND_FLOOR = 4
Round towards -Infinity.

ROUND_HALF_DOWN = 5
Round to nearest with ties going towards zero.

ROUND_HALF_EVEN = 6
Round to nearest with ties going to nearest even integer.

ROUND_HALF_UP = 7
Round to nearest with ties going away from zero.

ROUND_UP = 8
Round away from zero.

Unless a rounding mode is explicitely given, the rounding mode set as current default is used. To get or set the default rounding mode, the package *decimalfp* provides the following two functions:

get_dflt_rounding_mode() → *ROUNDING*
Return default rounding mode.

set_dflt_rounding_mode(*rounding:* ROUNDING) → None
Set default rounding mode.

Parameters **rounding** (ROUNDING) – rounding mode to be set as default

Raises **TypeError** – given 'rounding' is not a valid rounding mode

CHAPTER

TWO

INDICES AND TABLES

- genindex
- modindex
- search

PYTHON MODULE INDEX

INDEX

Symbols

A

D

F

G

I

M

N

P

Q

R

S