Jeplerudecimal Library Documentation

Release 1.0

jepler

Jun 08, 2022

CONTENTS

Reduced version of the decimal library for CircuitPython

CHAPTER

ONE

DEPENDENCIES

This library depends on:

- Adafruit CircuitPython

The library also runs on desktop Python3, and should give numerically identical results across all platorms.

CHAPTER

TWO

INSTALLING FROM PYPI

To install for current user:

```
python3 -mpip install --user jepler-circuitpython-udecimal
```

To install system-wide (this may be required in some cases):

```
sudo python3 -mpip install jepler-circuitpython-udecimal
```

To install in a virtual environment in your current project:

```
mkdir project-name && cd project-name
python3 -m venv .env
source .env/bin/activate
pip3 install jepler-circuitpython-udecimal
```

CHAPTER

THREE

USAGE EXAMPLE

```
>>> from jepler_udecimal import Decimal
>>> Decimal(2)/3
Decimal('0.66666666666666666666666666667')
>>> Decimal('.1') + Decimal('.2') == Decimal('.3')
True
```

CHAPTER

FOUR

CONTRIBUTING

Contributions are welcome! Please read our Code of Conduct before contributing to help this project stay welcoming.

CHAPTER

FIVE

DOCUMENTATION

For information on building library documentation, please check out this guide.

CHAPTER

SIX

TABLE OF CONTENTS

6.1 API Reference

This page contains auto-generated API reference documentation[1].

6.1.1 jepler_udecimal

Reduced version of the decimal library for CircuitPython. It runs on CircuitPython as well as standard Python, though you should probably use the built in decimal module on standard Python.

It still requires a fairly beefy mcu to run. Importing jepler_udecimal on an nRF52840 uses about 52kB of heap.

- Author(s): jepler

6.1.1.1 Implementation Notes

This is a reduced version of the "_pydecimal" library from Python 3.7, together with a translation of trig routines from https://git.yzena.com/gavin/bc/src/branch/master/gen/lib.bc

Software and Dependencies:

- Adafruit CircuitPython firmware for the supported boards: https://github.com/adafruit/circuitpython/releases

This is an implementation of decimal floating point arithmetic based on the General Decimal Arithmetic Specification and IEEE standard 854-1987.

Decimal floating point has finite precision with arbitrarily large bounds.

The purpose of this module is to support arithmetic using familiar "schoolhouse" rules and to avoid some of the tricky representation issues associated with binary floating point. The package is especially useful for financial applications or for contexts where users have expectations that are at odds with binary floating point (for instance, in binary floating point, `1.00 % 0.1` gives `0.09999999999999995` instead of `0.0`; `Decimal('1.00') % Decimal('0.1')` returns the expected `Decimal('0.00')`).

Here are some examples of using the udecimal module:

```
>>> from jepler_udecimal import *
>>> setcontext(ExtendedContext)
>>> Decimal(0)
Decimal('0')
>>> Decimal('1')
```

(continues on next page)

[1] Created with sphinx-autoapi

(continued from previous page)

```
Decimal('1')
>>> Decimal('-.0123')
Decimal('-0.0123')
>>> Decimal(123456)
Decimal('123456')
>>> Decimal('123.45e12345678')
Decimal('1.2345E+12345680')
>>> Decimal('1.33') + Decimal('1.27')
Decimal('2.60')
>>> Decimal('12.34') + Decimal('3.87') - Decimal('18.41')
Decimal('-2.20')
>>> dig = Decimal(1)
>>> print(dig / Decimal(3))
0.333333333
>>> getcontext().prec = 18
>>> print(dig / Decimal(3))
0.333333333333333333
>>> print(dig.sqrt())
1
>>> print(Decimal(3).sqrt())
1.73205080756887729
>>> print(Decimal(3) ** 123)
4.85192780976896427E+58
>>> inf = Decimal(1) / Decimal(0)
>>> print(inf)
Infinity
>>> neginf = Decimal(-1) / Decimal(0)
>>> print(neginf)
-Infinity
>>> print(neginf + inf)
NaN
>>> print(neginf * inf)
-Infinity
>>> print(dig / 0)
Infinity
>>> getcontext().traps[DivisionByZero] = 1
>>> print(dig / 0)
Traceback (most recent call last):
  ...
  ...
  ...
jepler_udecimal.DivisionByZero: x / 0
>>> c = Context()
>>> c.traps[InvalidOperation] = 0
>>> print(c.flags[InvalidOperation])
0
>>> c.divide(Decimal(0), Decimal(0))
Decimal('NaN')
>>> c.traps[InvalidOperation] = 1
>>> print(c.flags[InvalidOperation])
1
>>> c.flags[InvalidOperation] = 0
```

(continues on next page)

(continued from previous page)

```
>>> print(c.flags[InvalidOperation])
0
>>> print(c.divide(Decimal(0), Decimal(0)))
Traceback (most recent call last):
  ...
  ...
  ...
jepler_udecimal.InvalidOperation: 0 / 0
>>> print(c.flags[InvalidOperation])
1
>>> c.flags[InvalidOperation] = 0
>>> c.traps[InvalidOperation] = 0
>>> print(c.divide(Decimal(0), Decimal(0)))
NaN
>>> print(c.flags[InvalidOperation])
1
>>>
```

6.1.1.2 Submodules

jepler_udecimal.test

Module Contents

Functions

load_tests(loader, tests, ignore)

jepler_udecimal.test.**load_tests**(*loader*, *tests*, *ignore*)

jepler_udecimal.utrig

Trig functions using jepler_udecimal

Importing this module adds the relevant methods to the *Decimal* object.

Generally speaking, these routines increase the precision by some amount, perform argument range reduction followed by evaluation of a taylor polynomial, then reduce the precision of the result to equal the origial context's precision.

There is no guarantee that the results are correctly rounded in all cases, however, in all but the rarest cases the digits except the last one can be trusted.

Here are some examples of using utrig:

```
>>> import jepler_udecimal.utrig
>>> from jepler_udecimal import Decimal
>>> Decimal('.7').atan()
Decimal('0.610725964389208616543758876 5')
>>> Decimal('.1').acos()
```

(continues on next page)

(continued from previous page)

```
Decimal('1.4706289056333368228857985 12')
>>> Decimal('-.1').asin()
Decimal('-0.1001674211615597963455231795')
>>> Decimal('.4').tan()
Decimal('0.4227932187381617619816354272')
>>> Decimal('.5').cos()
Decimal('0.8775825618903727161162815826')
>>> Decimal('.6').sin()
Decimal('0.5646424733950353572009454457')
>>> Decimal('1').asin()
Decimal('1.5707963267948966192313216 92')
>>> Decimal('-1').acos()
Decimal('3.1415926535897932384626433 83')
```

Module Contents

Functions

atan(x, context=None)	Compute the arctangent of the specified value, in radians
sin(x, context=None)	Compute the sine of the specified value, in radians
cos(x, context=None)	Compute the cosine of the specified value, in radians
tan(x, context=None)	Compute the tangent of the specified value, in radians
asin(x, context=None)	Compute the arcsine of the specified value, in radians
acos(x, context=None)	Compute the arccosine of the specified value, in radians

jepler_udecimal.utrig.**atan**(*x*, *context=None*)

Compute the arctangent of the specified value, in radians

jepler_udecimal.utrig.**sin**(*x*, *context=None*)

Compute the sine of the specified value, in radians

jepler_udecimal.utrig.**cos**(*x*, *context=None*)

Compute the cosine of the specified value, in radians

jepler_udecimal.utrig.**tan**(*x*, *context=None*)

Compute the tangent of the specified value, in radians

jepler_udecimal.utrig.**asin**(*x*, *context=None*)

Compute the arcsine of the specified value, in radians

jepler_udecimal.utrig.**acos**(*x*, *context=None*)

Compute the arccosine of the specified value, in radians

6.1.1.3 Package Contents

Classes

Decimal	Floating point class for decimal arithmetic.
Context	Contains the context for a Decimal instance.

Functions

getcontext()	Returns this thread's context.
setcontext(context)	Set this thread's context to context.
localcontext(ctx=None)	Return a context manager for a copy of the supplied context

jepler_udecimal.**__version__ = 0.0.0-auto.0**

jepler_udecimal.**__repo__ = https://github.com/jepler/jepler_CircuitPython_udecimal.git**

jepler_udecimal.**DecimalTuple**

jepler_udecimal.**ROUND_DOWN = ROUND_DOWN**

jepler_udecimal.**ROUND_HALF_UP = ROUND_HALF_UP**

jepler_udecimal.**ROUND_HALF_EVEN = ROUND_HALF_EVEN**

jepler_udecimal.**ROUND_CEILING = ROUND_CEILING**

jepler_udecimal.**ROUND_FLOOR = ROUND_FLOOR**

jepler_udecimal.**ROUND_UP = ROUND_UP**

jepler_udecimal.**ROUND_HALF_DOWN = ROUND_HALF_DOWN**

jepler_udecimal.**ROUND_05UP = ROUND_05UP**

jepler_udecimal.**NotImplemented**

exception jepler_udecimal.**DecimalException**

Bases: ArithmeticError

Base exception class.

Used exceptions derive from this. If an exception derives from another exception besides this (such as Underflow (Inexact, Rounded, Subnormal) that indicates that it is only called if the others are present. This isn't actually used for anything, though.

handle – Called when context._raise_error is called and the
trap_enabler is not set. First argument is self, second is the context. More arguments can be given, those being after the explanation in _raise_error (For example, context._raise_error(NewError, '(-x)!', self._sign) would call NewError().handle(context, self._sign).)

To define a new exception, it should be sufficient to have it derive from DecimalException.

Initialize self. See help(type(self)) for accurate signature.

handle(*self*, *context*, **args*)

exception jepler_udecimal.**Clamped**

Bases: *DecimalException*

Exponent of a 0 changed to fit bounds.

This occurs and signals clamped if the exponent of a result has been altered in order to fit the constraints of a specific concrete representation. This may occur when the exponent of a zero result would be outside the bounds of a representation, or when a large normal number would have an encoded exponent that cannot be represented. In this latter case, the exponent is reduced to fit and the corresponding number of zero digits are appended to the coefficient ("fold-down").

Initialize self. See help(type(self)) for accurate signature.

exception jepler_udecimal.**InvalidOperation**

Bases: *DecimalException*

An invalid operation was performed.

Various bad things cause this:

Something creates a signaling NaN -INF + INF 0 * (+-)INF (+-)INF / (+-)INF x % 0 (+-)INF % x x._rescale(non-integer) sqrt(-x) , x > 0 0 ** 0 x ** (non-integer) x ** (+-)INF An operand is invalid

The result of the operation after these is a quiet positive NaN, except when the cause is a signaling NaN, in which case the result is also a quiet NaN, but with the original sign, and an optional diagnostic information.

Initialize self. See help(type(self)) for accurate signature.

handle(*self*, *context*, **args*)

exception jepler_udecimal.**ConversionSyntax**

Bases: *InvalidOperation*

Trying to convert badly formed string.

This occurs and signals invalid-operation if a string is being converted to a number and it does not conform to the numeric string syntax. The result is [0,qNaN].

Initialize self. See help(type(self)) for accurate signature.

handle(*self*, *context*, **args*)

exception jepler_udecimal.**DivisionByZero**

Bases: *DecimalException*

Division by 0.

This occurs and signals division-by-zero if division of a finite number by zero was attempted (during a divide-integer or divide operation, or a power operation with negative right-hand operand), and the dividend was not zero.

The result of the operation is [sign,inf], where sign is the exclusive or of the signs of the operands for divide, or is 1 for an odd power of -0, for power.

Initialize self. See help(type(self)) for accurate signature.

handle(*self*, *context*, *sign*, **args*)

exception jepler_udecimal.**DivisionImpossible**

Bases: *InvalidOperation*

Cannot perform the division adequately.

This occurs and signals invalid-operation if the integer result of a divide-integer or remainder operation had too many digits (would be longer than precision). The result is [0,qNaN].

Initialize self. See help(type(self)) for accurate signature.

handle(*self*, *context*, **args*)

exception jepler_udecimal.**DivisionUndefined**

Bases: *InvalidOperation*

Undefined result of division.

This occurs and signals invalid-operation if division by zero was attempted (during a divide-integer, divide, or remainder operation), and the dividend is also zero. The result is [0,qNaN].

Initialize self. See help(type(self)) for accurate signature.

handle(*self*, *context*, **args*)

exception jepler_udecimal.**Inexact**

Bases: *DecimalException*

Had to round, losing information.

This occurs and signals inexact whenever the result of an operation is not exact (that is, it needed to be rounded and any discarded digits were non-zero), or if an overflow or underflow condition occurs. The result in all cases is unchanged.

The inexact signal may be tested (or trapped) to determine if a given operation (or sequence of operations) was inexact.

Initialize self. See help(type(self)) for accurate signature.

exception jepler_udecimal.**InvalidContext**

Bases: *InvalidOperation*

Invalid context. Unknown rounding, for example.

This occurs and signals invalid-operation if an invalid context was detected during an operation. This can occur if contexts are not checked on creation and either the precision exceeds the capability of the underlying concrete representation or an unknown or unsupported rounding was specified. These aspects of the context need only be checked when the values are required to be used. The result is [0,qNaN].

Initialize self. See help(type(self)) for accurate signature.

handle(*self*, *context*, **args*)

exception jepler_udecimal.**Rounded**

Bases: *DecimalException*

Number got rounded (not necessarily changed during rounding).

This occurs and signals rounded whenever the result of an operation is rounded (that is, some zero or non-zero digits were discarded from the coefficient), or if an overflow or underflow condition occurs. The result in all cases is unchanged.

The rounded signal may be tested (or trapped) to determine if a given operation (or sequence of operations) caused a loss of precision.

Initialize self. See help(type(self)) for accurate signature.

exception jepler_udecimal.**Subnormal**

Bases: *DecimalException*

Exponent < Emin before rounding.

This occurs and signals subnormal whenever the result of a conversion or operation is subnormal (that is, its adjusted exponent is less than Emin, before any rounding). The result in all cases is unchanged.

The subnormal signal may be tested (or trapped) to determine if a given or operation (or sequence of operations) yielded a subnormal result.

Initialize self. See help(type(self)) for accurate signature.

exception jepler_udecimal.**Overflow**

Bases: *DecimalException*

Numerical overflow.

This occurs and signals overflow if the adjusted exponent of a result (from a conversion or from an operation that is not an attempt to divide by zero), after rounding, would be greater than the largest value that can be handled by the implementation (the value Emax).

The result depends on the rounding mode:

For round-half-up and round-half-even (and for round-half-down and round-up, if implemented), the result of the operation is [sign,inf], where sign is the sign of the intermediate result. For round-down, the result is the largest finite number that can be represented in the current precision, with the sign of the intermediate result. For round-ceiling, the result is the same as for round-down if the sign of the intermediate result is 1, or is [0,inf] otherwise. For round-floor, the result is the same as for round-down if the sign of the intermediate result is 0, or is [1,inf] otherwise. In all cases, Inexact and Rounded will also be raised.

Initialize self. See help(type(self)) for accurate signature.

handle(*self*, *context*, *sign*, **args*)

exception jepler_udecimal.**Underflow**

Bases: *DecimalException*

Numerical underflow with result rounded to 0.

This occurs and signals underflow if a result is inexact and the adjusted exponent of the result would be smaller (more negative) than the smallest value that can be handled by the implementation (the value Emin). That is, the result is both inexact and subnormal.

The result after an underflow will be a subnormal number rounded, if necessary, so that its exponent is not less than Etiny. This may result in 0 with the sign of the intermediate result and an exponent of Etiny.

In all cases, Inexact, Rounded, and Subnormal will also be raised.

Initialize self. See help(type(self)) for accurate signature.

exception jepler_udecimal.**FloatOperation**

Bases: *DecimalException*

Enable stricter semantics for mixing floats and Decimals.

If the signal is not trapped (default), mixing floats and Decimals is permitted in the Decimal() constructor, context.create_decimal() and all comparison operators. Both conversion and comparisons are exact. Any occurrence of a mixed operation is silently recorded by setting FloatOperation in the context flags. Explicit conversions with Decimal.from_float() or context.create_decimal_from_float() do not set the flag.

Otherwise (the signal is trapped), only equality comparisons and explicit conversions are silent. All other mixed operations raise FloatOperation.

Initialize self. See help(type(self)) for accurate signature.

jepler_udecimal.**getcontext**()

Returns this thread's context.

If this thread does not yet have a context, returns a new context and sets this thread's context. New contexts are copies of DefaultContext.

jepler_udecimal.**setcontext**(*context*)

Set this thread's context to context.

jepler_udecimal.**localcontext**(*ctx=None*)

Return a context manager for a copy of the supplied context

Uses a copy of the current context if no context is specified The returned context manager creates a local decimal context in a with statement:

```
def sin(x):
     with localcontext() as ctx:
         ctx.prec += 2
         # Rest of sin calculation algorithm
         # uses a precision 2 greater than normal
     return +s  # Convert result to normal precision

 def sin(x):
     with localcontext(ExtendedContext):
         # Rest of sin calculation algorithm
         # uses the Extended Context from the
         # General Decimal Arithmetic Specification
     return +s  # Convert result to normal context
```

```
>>> setcontext(DefaultContext)
>>> print(getcontext().prec)
28
>>> with localcontext():
...     ctx = getcontext()
...     ctx.prec += 2
...     print(ctx.prec)
...
30
>>> with localcontext(ExtendedContext):
...     print(getcontext().prec)
...
9
>>> print(getcontext().prec)
28
```

class jepler_udecimal.**Decimal**

Bases: object

Floating point class for decimal arithmetic.

Create a decimal point instance.

```
>>> Decimal('3.14')              # string input
Decimal('3.14')
>>> Decimal((0, (3, 1, 4), -2))  # tuple (sign, digit_tuple, exponent)
Decimal('3.14')
>>> Decimal(314)                 # int
Decimal('314')
>>> Decimal(Decimal(314))        # another decimal instance
Decimal('314')
>>> Decimal('  3.14  \n')        # leading and trailing whitespace okay
Decimal('3.14')
```

__slots__ = ['_exp', '_int', '_sign', '_is_special']

__radd__

__rmul__

__trunc__

to_integral

classmethod from_float(*cls*, *f*)

Converts a float to a decimal number, exactly.

Note that Decimal.from_float(0.1) is not the same as Decimal('0.1'). Since 0.1 is not exactly representable in binary floating point, the value is stored as the nearest representable value which is 0x1.999999999999ap-4. The exact equivalent of the value in decimal is 0.1000000000000000055511151231257827021181583404541015625 on desktop Python. On CircuitPython, the value has fewer significant figures.

```
>>> Decimal.from_float(0.1)
Decimal('0.1000000000000000055511151231257827021181583404541015625')
>>> Decimal.from_float(float('nan'))
Decimal('NaN')
>>> Decimal.from_float(float('inf'))
Decimal('Infinity')
>>> Decimal.from_float(-float('inf'))
Decimal('-Infinity')
>>> Decimal.from_float(-0.0)
Decimal('-0')
```

__bool__(*self*)

Return True if self is nonzero; otherwise return False.

NaNs and infinities are considered nonzero.

__eq__(*self*, *other*, *context=None*)

Return self==value.

__lt__(*self*, *other*, *context=None*)

Return self<value.

__le__(*self*, *other*, *context=None*)

Return self<=value.

__gt__(*self*, *other*, *context=None*)

Return self>value.

__ge__(*self*, *other*, *context=None*)

Return self>=value.

compare(*self*, *other*, *context=None*)

Compare self to other. Return a decimal value:

a or b is a NaN	==>	Decimal('NaN')
a < b	==>	Decimal('-1')
a == b	==>	Decimal('0')
a > b	==>	Decimal('1')

__hash__(*self*)

x.__hash__() <==> hash(x)

as_tuple(*self*)

Represents the number as a triple tuple.

To show the internals exactly as they are.

__repr__(*self*)

Represents the number as an instance of Decimal.

__str__(*self*, *eng=False*, *context=None*)

Return string representation of the number in scientific notation.

Captures all of the information in the underlying representation.

to_eng_string(*self*, *context=None*)

Convert to a string, using engineering notation if an exponent is needed.

Engineering notation has an exponent which is a multiple of 3. This can leave up to 3 digits to the left of the decimal place and may require the addition of either one or two trailing zeros.

__neg__(*self*, *context=None*)

Returns a copy with the sign switched.

Rounds, if it has reason.

__pos__(*self*, *context=None*)

Returns a copy, unless it is a sNaN.

Rounds the number (if more than precision digits)

__abs__(*self*, *round=True*, *context=None*)

Returns the absolute value of self.

If the keyword argument 'round' is false, do not round. The expression self.__abs__(round=False) is equivalent to self.copy_abs().

__add__(*self*, *other*, *context=None*)

Returns self + other.

-INF + INF (or the reverse) cause InvalidOperation errors.

__sub__(*self*, *other*, *context=None*)

Return self - other

__rsub__(*self*, *other*, *context=None*)

Return other - self

__mul__(*self*, *other*, *context=None*)

Return self * other.

(+-) INF * 0 (or its reverse) raise InvalidOperation.

__truediv__(*self*, *other*, *context=None*)

Return self / other.

__rtruediv__(*self*, *other*, *context=None*)

Swaps self/other and returns __truediv__.

__divmod__(*self*, *other*, *context=None*)

Return (self // other, self % other)

__rdivmod__(*self*, *other*, *context=None*)

Swaps self/other and returns __divmod__.

__mod__(*self*, *other*, *context=None*)

self % other

__rmod__(*self*, *other*, *context=None*)

Swaps self/other and returns __mod__.

remainder_near(*self*, *other*, *context=None*)

Remainder nearest to 0- abs(remainder-near) <= other/2

__floordiv__(*self*, *other*, *context=None*)

self // other

__rfloordiv__(*self*, *other*, *context=None*)

Swaps self/other and returns __floordiv__.

__float__(*self*)

Float representation.

__int__(*self*)

Converts self to an int, truncating if necessary.

__round__(*self*, *n=None*)

Round self to the nearest integer, or to a given precision.

If only one argument is supplied, round a finite Decimal instance self to the nearest integer. If self is infinite or a NaN then a Python exception is raised. If self is finite and lies exactly halfway between two integers then it is rounded to the integer with even last digit.

```
>>> round(Decimal('123.456'))
123
>>> round(Decimal('-456.789'))
-457
>>> round(Decimal('-3.0'))
-3
>>> round(Decimal('2.5'))
2
>>> round(Decimal('3.5'))
4
```

(continues on next page)

(continued from previous page)

```
>>> round(Decimal('Inf'))
Traceback (most recent call last):
  ...
OverflowError: cannot round an infinity
>>> round(Decimal('NaN'))
Traceback (most recent call last):
  ...
ValueError: cannot round a NaN
```

If a second argument n is supplied, self is rounded to n decimal places using the rounding mode for the current context.

For an integer n, round(self, -n) is exactly equivalent to self.quantize(Decimal('1En')).

```
>>> round(Decimal('123.456'), 0)
Decimal('123')
>>> round(Decimal('123.456'), 2)
Decimal('123.46')
>>> round(Decimal('123.456'), -2)
Decimal('1E+2')
>>> getcontext().traps[InvalidOperation] = 0
>>> round(Decimal('-Infinity'), 37)
Decimal('NaN')
>>> round(Decimal('sNaN123'), 0)
Decimal('NaN123')
```

__floor__(*self*)

Return the floor of self, as an integer.

For a finite Decimal instance self, return the greatest integer n such that n <= self. If self is infinite or a NaN then a Python exception is raised.

__ceil__(*self*)

Return the ceiling of self, as an integer.

For a finite Decimal instance self, return the least integer n such that n >= self. If self is infinite or a NaN then a Python exception is raised.

__pow__(*self*, *other*, *, *context=None*)

Return self ** other.

__rpow__(*self*, *other*, *context=None*)

Swaps self/other and returns __pow__.

normalize(*self*, *context=None*)

Normalize- strip trailing 0s, change anything equal to 0 to 0e0

quantize(*self*, *exp*, *rounding=None*, *context=None*)

Quantize self so its exponent is the same as that of exp.

Similar to self._rescale(exp._exp) but with error checking.

same_quantum(*self*, *other*, *context=None*)

Return True if self and other have the same exponent; otherwise return False.

If either operand is a special value, the following rules are used:

- return True if both operands are infinities
- return True if both operands are NaNs
- otherwise, return False.

to_integral_exact(*self*, *rounding=None*, *context=None*)

Rounds to a nearby integer.

If no rounding mode is specified, take the rounding mode from the context. This method raises the Rounded and Inexact flags when appropriate.

See also: to_integral_value, which does exactly the same as this method except that it doesn't raise Inexact or Rounded.

to_integral_value(*self*, *rounding=None*, *context=None*)

Rounds to the nearest integer, without raising inexact, rounded.

sqrt(*self*, *context=None*)

Return the square root of self.

max(*self*, *other*, *context=None*)

Returns the larger value.

Like max(self, other) except if one is not a number, returns NaN (and signals if one is sNaN). Also rounds.

min(*self*, *other*, *context=None*)

Returns the smaller value.

Like min(self, other) except if one is not a number, returns NaN (and signals if one is sNaN). Also rounds.

adjusted(*self*)

Return the adjusted exponent of self

canonical(*self*)

Returns the same Decimal object.

As we do not have different encodings for the same number, the received object already is in its canonical form.

compare_signal(*self*, *other*, *context=None*)

Compares self to the other operand numerically.

It's pretty much like compare(), but all NaNs signal, with signaling NaNs taking precedence over quiet NaNs.

compare_total(*self*, *other*, *context=None*)

Compares self to other using the abstract representations.

This is not like the standard compare, which use their numerical value. Note that a total ordering is defined for all possible abstract representations.

compare_total_mag(*self*, *other*, *context=None*)

Compares self to other using abstract repr., ignoring sign.

Like compare_total, but with operand's sign ignored and assumed to be 0.

copy_abs(*self*)

Returns a copy with the sign set to 0.

copy_negate(*self*)

Returns a copy with the sign inverted.

copy_sign(*self*, *other*, *context=None*)

Returns self with the sign of other.

exp(*self*, *context=None*)

Returns e ** self.

is_canonical(*self*)

Return True if self is canonical; otherwise return False.

Currently, the encoding of a Decimal instance is always canonical, so this method returns True for any Decimal.

is_finite(*self*)

Return True if self is finite; otherwise return False.

A Decimal instance is considered finite if it is neither infinite nor a NaN.

is_infinite(*self*)

Return True if self is infinite; otherwise return False.

is_nan(*self*)

Return True if self is a qNaN or sNaN; otherwise return False.

is_normal(*self*, *context=None*)

Return True if self is a normal number; otherwise return False.

is_qnan(*self*)

Return True if self is a quiet NaN; otherwise return False.

is_signed(*self*)

Return True if self is negative; otherwise return False.

is_snan(*self*)

Return True if self is a signaling NaN; otherwise return False.

is_subnormal(*self*, *context=None*)

Return True if self is subnormal; otherwise return False.

is_zero(*self*)

Return True if self is a zero; otherwise return False.

ln(*self*, *context=None*)

Returns the natural (base e) logarithm of self.

log10(*self*, *context=None*)

Returns the base 10 logarithm of self.

logb(*self*, *context=None*)

Returns the exponent of the magnitude of self's MSD.

The result is the integer which is the exponent of the magnitude of the most significant digit of self (as though it were truncated to a single digit while maintaining the value of that digit and without limiting the resulting exponent).

max_mag(*self*, *other*, *context=None*)

Compares the values numerically with their sign ignored.

min_mag(*self*, *other*, *context=None*)

Compares the values numerically with their sign ignored.

number_class(*self*, *context=None*)

Returns an indication of the class of self.

The class is one of the following strings:

- sNaN
- NaN
- -Infinity
- -Normal
- -Subnormal
- -Zero
- +Zero
- +Subnormal
- +Normal
- +Infinity

radix(*self*)

Just returns 10, as this is Decimal, :)

scaleb(*self*, *other*, *context=None*)

Returns self operand after adding the second value to its exp.

class jepler_udecimal.**Context**(*prec=None*, *rounding=None*, *Emin=None*, *Emax=None*, *capitals=None*, *clamp=None*, *flags=None*, *traps=None*, *_ignored_flags=None*)

Bases: object

Contains the context for a Decimal instance.

Parameters

- **prec** (*int*) – precision (for use in rounding, division, square roots..)
- **rounding** (*str*) – rounding type (how you round)
- **traps** (*dict*) – if traps[exception] = 1, then the exception is raised when it is caused. Otherwise, a value is substituted in.
- **flags** (*dict*) – when an exception is caused, flags[exception] is set. (Whether or not the trap_enabler is set) Should be reset by user of Decimal instance.
- **Emin** (*int*) – minimum exponent
- **Emax** (*int*) – maximum exponent
- **capitals** (*bool*) – if true, 1*10^1 is printed as 1E+1, else printed as 1e1
- **clamp** (*bool*) – If true, change exponents if too high

__copy__

__hash__

to_integral

__setattr__(*self*, *name*, *value*)

Implement setattr(self, name, value).

__delattr__(*self*, *name*)

Implement delattr(self, name).

__reduce__(*self*)

Helper for pickle.

__repr__(*self*)

Show the current context.

clear_flags(*self*)

Reset all flags to zero

clear_traps(*self*)

Reset all traps to zero

copy(*self*)

Returns a deep copy from self.

Etiny(*self*)

Returns Etiny (= Emin - prec + 1)

Etop(*self*)

Returns maximum exponent (= Emax - prec + 1)

create_decimal(*self*, *num='0'*)

Creates a new Decimal instance but using self as context.

This method implements the to-number operation of the IBM Decimal specification.

create_decimal_from_float(*self*, *f*)

Creates a new Decimal instance from a float but rounding using self as the context.

```
>>> context = Context(prec=5, rounding=ROUND_DOWN)
>>> context.create_decimal_from_float(3.1415926535897932)
Decimal('3.1415')
>>> context = Context(prec=5, traps=[Inexact])
>>> context.create_decimal_from_float(3.1415926535897932)
Traceback (most recent call last):
    ...
jepler_udecimal.Inexact: None
```

abs(*self*, *a*)

Returns the absolute value of the operand.

If the operand is negative, the result is the same as using the minus operation on the operand. Otherwise, the result is the same as using the plus operation on the operand.

```
>>> ExtendedContext.abs(Decimal('2.1'))
Decimal('2.1')
>>> ExtendedContext.abs(Decimal('-100'))
Decimal('100')
>>> ExtendedContext.abs(Decimal('101.5'))
Decimal('101.5')
>>> ExtendedContext.abs(Decimal('-101.5'))
Decimal('101.5')
>>> ExtendedContext.abs(-1)
Decimal('1')
```

add(*self*, *a*, *b*)

Return the sum of the two operands.

```
>>> ExtendedContext.add(Decimal('12'), Decimal('7.00'))
Decimal('19.00')
>>> ExtendedContext.add(Decimal('1E+2'), Decimal('1.01E+4'))
Decimal('1.02E+4')
>>> ExtendedContext.add(1, Decimal(2))
Decimal('3')
>>> ExtendedContext.add(Decimal(8), 5)
Decimal('13')
>>> ExtendedContext.add(5, 5)
Decimal('10')
```

canonical(*self*, *a*)

Returns the same Decimal object.

As we do not have different encodings for the same number, the received object already is in its canonical form.

```
>>> ExtendedContext.canonical(Decimal('2.50'))
Decimal('2.50')
```

compare(*self*, *a*, *b*)

Compares values numerically.

If the signs of the operands differ, a value representing each operand ('-1' if the operand is less than zero, '0' if the operand is zero or negative zero, or '1' if the operand is greater than zero) is used in place of that operand for the comparison instead of the actual operand.

The comparison is then effected by subtracting the second operand from the first and then returning a value according to the result of the subtraction: '-1' if the result is less than zero, '0' if the result is zero or negative zero, or '1' if the result is greater than zero.

```
>>> ExtendedContext.compare(Decimal('2.1'), Decimal('3'))
Decimal('-1')
>>> ExtendedContext.compare(Decimal('2.1'), Decimal('2.1'))
Decimal('0')
>>> ExtendedContext.compare(Decimal('2.1'), Decimal('2.10'))
Decimal('0')
>>> ExtendedContext.compare(Decimal('3'), Decimal('2.1'))
Decimal('1')
>>> ExtendedContext.compare(Decimal('2.1'), Decimal('-3'))
Decimal('1')
>>> ExtendedContext.compare(Decimal('-3'), Decimal('2.1'))
Decimal('-1')
>>> ExtendedContext.compare(1, 2)
Decimal('-1')
>>> ExtendedContext.compare(Decimal(1), 2)
Decimal('-1')
>>> ExtendedContext.compare(1, Decimal(2))
Decimal('-1')
```

compare_signal(*self*, *a*, *b*)

Compares the values of the two operands numerically.

It's pretty much like compare(), but all NaNs signal, with signaling NaNs taking precedence over quiet NaNs.

```
>>> c = ExtendedContext
>>> c.compare_signal(Decimal('2.1'), Decimal('3'))
Decimal('-1')
>>> c.compare_signal(Decimal('2.1'), Decimal('2.1'))
Decimal('0')
>>> c.flags[InvalidOperation] = 0
>>> print(c.flags[InvalidOperation])
0
>>> c.compare_signal(Decimal('NaN'), Decimal('2.1'))
Decimal('NaN')
>>> print(c.flags[InvalidOperation])
1
>>> c.flags[InvalidOperation] = 0
>>> print(c.flags[InvalidOperation])
0
>>> c.compare_signal(Decimal('sNaN'), Decimal('2.1'))
Decimal('NaN')
>>> print(c.flags[InvalidOperation])
1
>>> c.compare_signal(-1, 2)
Decimal('-1')
>>> c.compare_signal(Decimal(-1), 2)
Decimal('-1')
>>> c.compare_signal(-1, Decimal(2))
Decimal('-1')
```

compare_total(*self*, *a*, *b*)

Compares two operands using their abstract representation.

This is not like the standard compare, which use their numerical value. Note that a total ordering is defined for all possible abstract representations.

```
>>> ExtendedContext.compare_total(Decimal('12.73'), Decimal('127.9'))
Decimal('-1')
>>> ExtendedContext.compare_total(Decimal('-127'),  Decimal('12'))
Decimal('-1')
>>> ExtendedContext.compare_total(Decimal('12.30'), Decimal('12.3'))
Decimal('-1')
>>> ExtendedContext.compare_total(Decimal('12.30'), Decimal('12.30'))
Decimal('0')
>>> ExtendedContext.compare_total(Decimal('12.3'),  Decimal('12.300'))
Decimal('1')
>>> ExtendedContext.compare_total(Decimal('12.3'),  Decimal('NaN'))
Decimal('-1')
>>> ExtendedContext.compare_total(1, 2)
Decimal('-1')
>>> ExtendedContext.compare_total(Decimal(1), 2)
Decimal('-1')
>>> ExtendedContext.compare_total(1, Decimal(2))
Decimal('-1')
```

compare_total_mag(*self*, *a*, *b*)

Compares two operands using their abstract representation ignoring sign.

Like compare_total, but with operand's sign ignored and assumed to be 0.

copy_abs(*self*, *a*)

Returns a copy of the operand with the sign set to 0.

```
>>> ExtendedContext.copy_abs(Decimal('2.1'))
Decimal('2.1')
>>> ExtendedContext.copy_abs(Decimal('-100'))
Decimal('100')
>>> ExtendedContext.copy_abs(-1)
Decimal('1')
```

copy_decimal(*self*, *a*)

Returns a copy of the decimal object.

```
>>> ExtendedContext.copy_decimal(Decimal('2.1'))
Decimal('2.1')
>>> ExtendedContext.copy_decimal(Decimal('-1.00'))
Decimal('-1.00')
>>> ExtendedContext.copy_decimal(1)
Decimal('1')
```

copy_negate(*self*, *a*)

Returns a copy of the operand with the sign inverted.

```
>>> ExtendedContext.copy_negate(Decimal('101.5'))
Decimal('-101.5')
>>> ExtendedContext.copy_negate(Decimal('-101.5'))
Decimal('101.5')
>>> ExtendedContext.copy_negate(1)
Decimal('-1')
```

copy_sign(*self*, *a*, *b*)

Copies the second operand's sign to the first one.

In detail, it returns a copy of the first operand with the sign equal to the sign of the second operand.

```
>>> ExtendedContext.copy_sign(Decimal( '1.50'), Decimal('7.33'))
Decimal('1.50')
>>> ExtendedContext.copy_sign(Decimal('-1.50'), Decimal('7.33'))
Decimal('1.50')
>>> ExtendedContext.copy_sign(Decimal( '1.50'), Decimal('-7.33'))
Decimal('-1.50')
>>> ExtendedContext.copy_sign(Decimal('-1.50'), Decimal('-7.33'))
Decimal('-1.50')
>>> ExtendedContext.copy_sign(1, -2)
Decimal('-1')
>>> ExtendedContext.copy_sign(Decimal(1), -2)
Decimal('-1')
>>> ExtendedContext.copy_sign(1, Decimal(-2))
Decimal('-1')
```

divide(*self*, *a*, *b*)

Decimal division in a specified context.

```
>>> ExtendedContext.divide(Decimal('1'), Decimal('3'))
Decimal('0.333333333')
>>> ExtendedContext.divide(Decimal('2'), Decimal('3'))
Decimal('0.666666667')
>>> ExtendedContext.divide(Decimal('5'), Decimal('2'))
Decimal('2.5')
>>> ExtendedContext.divide(Decimal('1'), Decimal('10'))
Decimal('0.1')
>>> ExtendedContext.divide(Decimal('12'), Decimal('12'))
Decimal('1')
>>> ExtendedContext.divide(Decimal('8.00'), Decimal('2'))
Decimal('4.00')
>>> ExtendedContext.divide(Decimal('2.400'), Decimal('2.0'))
Decimal('1.20')
>>> ExtendedContext.divide(Decimal('1000'), Decimal('100'))
Decimal('10')
>>> ExtendedContext.divide(Decimal('1000'), Decimal('1'))
Decimal('1000')
>>> ExtendedContext.divide(Decimal('2.40E+6'), Decimal('2'))
Decimal('1.20E+6')
>>> ExtendedContext.divide(5, 5)
Decimal('1')
>>> ExtendedContext.divide(Decimal(5), 5)
Decimal('1')
>>> ExtendedContext.divide(5, Decimal(5))
Decimal('1')
```

divide_int(*self*, *a*, *b*)

Divides two numbers and returns the integer part of the result.

```
>>> ExtendedContext.divide_int(Decimal('2'), Decimal('3'))
Decimal('0')
>>> ExtendedContext.divide_int(Decimal('10'), Decimal('3'))
Decimal('3')
>>> ExtendedContext.divide_int(Decimal('1'), Decimal('0.3'))
Decimal('3')
>>> ExtendedContext.divide_int(10, 3)
Decimal('3')
>>> ExtendedContext.divide_int(Decimal(10), 3)
Decimal('3')
>>> ExtendedContext.divide_int(10, Decimal(3))
Decimal('3')
```

divmod(*self*, *a*, *b*)

Return (a // b, a % b).

```
>>> ExtendedContext.divmod(Decimal(8), Decimal(3))
(Decimal('2'), Decimal('2'))
>>> ExtendedContext.divmod(Decimal(8), Decimal(4))
(Decimal('2'), Decimal('0'))
```

(continues on next page)

(continued from previous page)

```
>>> ExtendedContext.divmod(8, 4)
(Decimal('2'), Decimal('0'))
>>> ExtendedContext.divmod(Decimal(8), 4)
(Decimal('2'), Decimal('0'))
>>> ExtendedContext.divmod(8, Decimal(4))
(Decimal('2'), Decimal('0'))
```

exp(*self*, *a*)

Returns e ** a.

```
>>> c = ExtendedContext.copy()
>>> c.Emin = -999
>>> c.Emax = 999
>>> c.exp(Decimal('-Infinity'))
Decimal('0')
>>> c.exp(Decimal('-1'))
Decimal('0.367879441')
>>> c.exp(Decimal('0'))
Decimal('1')
>>> c.exp(Decimal('1'))
Decimal('2.71828183')
>>> c.exp(Decimal('0.693147181'))
Decimal('2.00000000')
>>> c.exp(Decimal('+Infinity'))
Decimal('Infinity')
>>> c.exp(10)
Decimal('22026.4658')
```

is_canonical(*self*, *a*)

Return True if the operand is canonical; otherwise return False.

Currently, the encoding of a Decimal instance is always canonical, so this method returns True for any Decimal.

```
>>> ExtendedContext.is_canonical(Decimal('2.50'))
True
```

is_finite(*self*, *a*)

Return True if the operand is finite; otherwise return False.

A Decimal instance is considered finite if it is neither infinite nor a NaN.

```
>>> ExtendedContext.is_finite(Decimal('2.50'))
True
>>> ExtendedContext.is_finite(Decimal('-0.3'))
True
>>> ExtendedContext.is_finite(Decimal('0'))
True
>>> ExtendedContext.is_finite(Decimal('Inf'))
False
>>> ExtendedContext.is_finite(Decimal('NaN'))
False
```

(continues on next page)

(continued from previous page)

```
>>> ExtendedContext.is_finite(1)
True
```

is_infinite(*self*, *a*)

Return True if the operand is infinite; otherwise return False.

```
>>> ExtendedContext.is_infinite(Decimal('2.50'))
False
>>> ExtendedContext.is_infinite(Decimal('-Inf'))
True
>>> ExtendedContext.is_infinite(Decimal('NaN'))
False
>>> ExtendedContext.is_infinite(1)
False
```

is_nan(*self*, *a*)

Return True if the operand is a qNaN or sNaN; otherwise return False.

```
>>> ExtendedContext.is_nan(Decimal('2.50'))
False
>>> ExtendedContext.is_nan(Decimal('NaN'))
True
>>> ExtendedContext.is_nan(Decimal('-sNaN'))
True
>>> ExtendedContext.is_nan(1)
False
```

is_normal(*self*, *a*)

Return True if the operand is a normal number; otherwise return False.

```
>>> c = ExtendedContext.copy()
>>> c.Emin = -999
>>> c.Emax = 999
>>> c.is_normal(Decimal('2.50'))
True
>>> c.is_normal(Decimal('0.1E-999'))
False
>>> c.is_normal(Decimal('0.00'))
False
>>> c.is_normal(Decimal('-Inf'))
False
>>> c.is_normal(Decimal('NaN'))
False
>>> c.is_normal(1)
True
```

is_qnan(*self*, *a*)

Return True if the operand is a quiet NaN; otherwise return False.

```
>>> ExtendedContext.is_qnan(Decimal('2.50'))
False
>>> ExtendedContext.is_qnan(Decimal('NaN'))
```

(continues on next page)

(continued from previous page)

```
True
>>> ExtendedContext.is_qnan(Decimal('sNaN'))
False
>>> ExtendedContext.is_qnan(1)
False
```

is_signed(*self*, *a*)

Return True if the operand is negative; otherwise return False.

```
>>> ExtendedContext.is_signed(Decimal('2.50'))
False
>>> ExtendedContext.is_signed(Decimal('-12'))
True
>>> ExtendedContext.is_signed(Decimal('-0'))
True
>>> ExtendedContext.is_signed(8)
False
>>> ExtendedContext.is_signed(-8)
True
```

is_snan(*self*, *a*)

Return True if the operand is a signaling NaN; otherwise return False.

```
>>> ExtendedContext.is_snan(Decimal('2.50'))
False
>>> ExtendedContext.is_snan(Decimal('NaN'))
False
>>> ExtendedContext.is_snan(Decimal('sNaN'))
True
>>> ExtendedContext.is_snan(1)
False
```

is_subnormal(*self*, *a*)

Return True if the operand is subnormal; otherwise return False.

```
>>> c = ExtendedContext.copy()
>>> c.Emin = -999
>>> c.Emax = 999
>>> c.is_subnormal(Decimal('2.50'))
False
>>> c.is_subnormal(Decimal('0.1E-999'))
True
>>> c.is_subnormal(Decimal('0.00'))
False
>>> c.is_subnormal(Decimal('-Inf'))
False
>>> c.is_subnormal(Decimal('NaN'))
False
>>> c.is_subnormal(1)
False
```

is_zero(*self*, *a*)

Return True if the operand is a zero; otherwise return False.

```
>>> ExtendedContext.is_zero(Decimal('0'))
True
>>> ExtendedContext.is_zero(Decimal('2.50'))
False
>>> ExtendedContext.is_zero(Decimal('-0E+2'))
True
>>> ExtendedContext.is_zero(1)
False
>>> ExtendedContext.is_zero(0)
True
```

ln(*self*, *a*)

Returns the natural (base e) logarithm of the operand.

```
>>> c = ExtendedContext.copy()
>>> c.Emin = -999
>>> c.Emax = 999
>>> c.ln(Decimal('0'))
Decimal('-Infinity')
>>> c.ln(Decimal('1.000'))
Decimal('0')
>>> c.ln(Decimal('2.71828183'))
Decimal('1.00000000')
>>> c.ln(Decimal('10'))
Decimal('2.30258509')
>>> c.ln(Decimal('+Infinity'))
Decimal('Infinity')
>>> c.ln(1)
Decimal('0')
```

log10(*self*, *a*)

Returns the base 10 logarithm of the operand.

```
>>> c = ExtendedContext.copy()
>>> c.Emin = -999
>>> c.Emax = 999
>>> c.log10(Decimal('0'))
Decimal('-Infinity')
>>> c.log10(Decimal('0.001'))
Decimal('-3')
>>> c.log10(Decimal('1.000'))
Decimal('0')
>>> c.log10(Decimal('2'))
Decimal('0.301029996')
>>> c.log10(Decimal('10'))
Decimal('1')
>>> c.log10(Decimal('70'))
Decimal('1.84509804')
>>> c.log10(Decimal('+Infinity'))
Decimal('Infinity')
>>> c.log10(0)
Decimal('-Infinity')
>>> c.log10(1)
```

(continues on next page)

(continued from previous page)

```
Decimal('0')
```

logb(*self*, *a*)

Returns the exponent of the magnitude of the operand's MSD.

The result is the integer which is the exponent of the magnitude of the most significant digit of the operand (as though the operand were truncated to a single digit while maintaining the value of that digit and without limiting the resulting exponent).

```
>>> ExtendedContext.logb(Decimal('250'))
Decimal('2')
>>> ExtendedContext.logb(Decimal('2.50'))
Decimal('0')
>>> ExtendedContext.logb(Decimal('0.03'))
Decimal('-2')
>>> ExtendedContext.logb(Decimal('0'))
Decimal('-Infinity')
>>> ExtendedContext.logb(1)
Decimal('0')
>>> ExtendedContext.logb(10)
Decimal('1')
>>> ExtendedContext.logb(100)
Decimal('2')
```

max(*self*, *a*, *b*)

max compares two values numerically and returns the maximum.

If either operand is a NaN then the general rules apply. Otherwise, the operands are compared as though by the compare operation. If they are numerically equal then the left-hand operand is chosen as the result. Otherwise the maximum (closer to positive infinity) of the two operands is chosen as the result.

```
>>> ExtendedContext.max(Decimal('3'), Decimal('2'))
Decimal('3')
>>> ExtendedContext.max(Decimal('-10'), Decimal('3'))
Decimal('3')
>>> ExtendedContext.max(Decimal('1.0'), Decimal('1'))
Decimal('1')
>>> ExtendedContext.max(Decimal('7'), Decimal('NaN'))
Decimal('7')
>>> ExtendedContext.max(1, 2)
Decimal('2')
>>> ExtendedContext.max(Decimal(1), 2)
Decimal('2')
>>> ExtendedContext.max(1, Decimal(2))
Decimal('2')
```

max_mag(*self*, *a*, *b*)

Compares the values numerically with their sign ignored.

```
>>> ExtendedContext.max_mag(Decimal('7'), Decimal('NaN'))
Decimal('7')
>>> ExtendedContext.max_mag(Decimal('7'), Decimal('-10'))
Decimal('-10')
```

(continues on next page)

(continued from previous page)

```
>>> ExtendedContext.max_mag(1, -2)
Decimal('-2')
>>> ExtendedContext.max_mag(Decimal(1), -2)
Decimal('-2')
>>> ExtendedContext.max_mag(1, Decimal(-2))
Decimal('-2')
```

min(*self*, *a*, *b*)

min compares two values numerically and returns the minimum.

If either operand is a NaN then the general rules apply. Otherwise, the operands are compared as though by the compare operation. If they are numerically equal then the left-hand operand is chosen as the result. Otherwise the minimum (closer to negative infinity) of the two operands is chosen as the result.

```
>>> ExtendedContext.min(Decimal('3'), Decimal('2'))
Decimal('2')
>>> ExtendedContext.min(Decimal('-10'), Decimal('3'))
Decimal('-10')
>>> ExtendedContext.min(Decimal('1.0'), Decimal('1'))
Decimal('1.0')
>>> ExtendedContext.min(Decimal('7'), Decimal('NaN'))
Decimal('7')
>>> ExtendedContext.min(1, 2)
Decimal('1')
>>> ExtendedContext.min(Decimal(1), 2)
Decimal('1')
>>> ExtendedContext.min(1, Decimal(29))
Decimal('1')
```

min_mag(*self*, *a*, *b*)

Compares the values numerically with their sign ignored.

```
>>> ExtendedContext.min_mag(Decimal('3'), Decimal('-2'))
Decimal('-2')
>>> ExtendedContext.min_mag(Decimal('-3'), Decimal('NaN'))
Decimal('-3')
>>> ExtendedContext.min_mag(1, -2)
Decimal('1')
>>> ExtendedContext.min_mag(Decimal(1), -2)
Decimal('1')
>>> ExtendedContext.min_mag(1, Decimal(-2))
Decimal('1')
```

minus(*self*, *a*)

Minus corresponds to unary prefix minus in Python.

The operation is evaluated using the same rules as subtract; the operation minus(a) is calculated as subtract('0', a) where the '0' has the same exponent as the operand.

```
>>> ExtendedContext.minus(Decimal('1.3'))
Decimal('-1.3')
>>> ExtendedContext.minus(Decimal('-1.3'))
Decimal('1.3')
```

(continues on next page)

(continued from previous page)

```
>>> ExtendedContext.minus(1)
Decimal('-1')
```

multiply(*self*, *a*, *b*)

multiply multiplies two operands.

If either operand is a special value then the general rules apply. Otherwise, the operands are multiplied together ('long multiplication'), resulting in a number which may be as long as the sum of the lengths of the two operands.

```
>>> ExtendedContext.multiply(Decimal('1.20'), Decimal('3'))
Decimal('3.60')
>>> ExtendedContext.multiply(Decimal('7'), Decimal('3'))
Decimal('21')
>>> ExtendedContext.multiply(Decimal('0.9'), Decimal('0.8'))
Decimal('0.72')
>>> ExtendedContext.multiply(Decimal('0.9'), Decimal('-0'))
Decimal('-0.0')
>>> ExtendedContext.multiply(Decimal('654321'), Decimal('654321'))
Decimal('4.28135971E+11')
>>> ExtendedContext.multiply(7, 7)
Decimal('49')
>>> ExtendedContext.multiply(Decimal(7), 7)
Decimal('49')
>>> ExtendedContext.multiply(7, Decimal(7))
Decimal('49')
```

normalize(*self*, *a*)

normalize reduces an operand to its simplest form.

Essentially a plus operation with all trailing zeros removed from the result.

```
>>> ExtendedContext.normalize(Decimal('2.1'))
Decimal('2.1')
>>> ExtendedContext.normalize(Decimal('-2.0'))
Decimal('-2')
>>> ExtendedContext.normalize(Decimal('1.200'))
Decimal('1.2')
>>> ExtendedContext.normalize(Decimal('-120'))
Decimal('-1.2E+2')
>>> ExtendedContext.normalize(Decimal('120.00'))
Decimal('1.2E+2')
>>> ExtendedContext.normalize(Decimal('0.00'))
Decimal('0')
>>> ExtendedContext.normalize(6)
Decimal('6')
```

number_class(*self*, *a*)

Returns an indication of the class of the operand.

The class is one of the following strings:

- -sNaN
- -NaN

- -Infinity
- -Normal
- -Subnormal
- -Zero
- +Zero
- +Subnormal
- +Normal
- +Infinity

```
>>> c = ExtendedContext.copy()
>>> c.Emin = -999
>>> c.Emax = 999
>>> c.number_class(Decimal('Infinity'))
'+Infinity'
>>> c.number_class(Decimal('1E-10'))
'+Normal'
>>> c.number_class(Decimal('2.50'))
'+Normal'
>>> c.number_class(Decimal('0.1E-999'))
'+Subnormal'
>>> c.number_class(Decimal('0'))
'+Zero'
>>> c.number_class(Decimal('-0'))
'-Zero'
>>> c.number_class(Decimal('-0.1E-999'))
'-Subnormal'
>>> c.number_class(Decimal('-1E-10'))
'-Normal'
>>> c.number_class(Decimal('-2.50'))
'-Normal'
>>> c.number_class(Decimal('-Infinity'))
'-Infinity'
>>> c.number_class(Decimal('NaN'))
'NaN'
>>> c.number_class(Decimal('-NaN'))
'NaN'
>>> c.number_class(Decimal('sNaN'))
'sNaN'
>>> c.number_class(123)
'+Normal'
```

plus(*self*, *a*)

Plus corresponds to unary prefix plus in Python.

The operation is evaluated using the same rules as add; the operation plus(a) is calculated as add('0', a) where the '0' has the same exponent as the operand.

```
>>> ExtendedContext.plus(Decimal('1.3'))
Decimal('1.3')
>>> ExtendedContext.plus(Decimal('-1.3'))
```

(continues on next page)

(continued from previous page)

```
Decimal('-1.3')
>>> ExtendedContext.plus(-1)
Decimal('-1')
```

power(*self*, *a*, *b*)

Raises a to the power of b

If a is negative then b must be integral. The result will be inexact unless b is integral and the result is finite and can be expressed exactly in 'precision' digits.

```
>>> c = ExtendedContext.copy()
>>> c.Emin = -999
>>> c.Emax = 999
>>> c.power(Decimal('2'), Decimal('3'))
Decimal('8')
>>> c.power(Decimal('-2'), Decimal('3'))
Decimal('-8')
>>> c.power(Decimal('2'), Decimal('-3'))
Decimal('0.125')
>>> c.power(Decimal('1.7'), Decimal('8'))
Decimal('69.7575744')
>>> c.power(Decimal('10'), Decimal('0.301029996'))
Decimal('2.00000000')
>>> c.power(Decimal('Infinity'), Decimal('-1'))
Decimal('0')
>>> c.power(Decimal('Infinity'), Decimal('0'))
Decimal('1')
>>> c.power(Decimal('Infinity'), Decimal('1'))
Decimal('Infinity')
>>> c.power(Decimal('-Infinity'), Decimal('-1'))
Decimal('-0')
>>> c.power(Decimal('-Infinity'), Decimal('0'))
Decimal('1')
>>> c.power(Decimal('-Infinity'), Decimal('1'))
Decimal('-Infinity')
>>> c.power(Decimal('-Infinity'), Decimal('2'))
Decimal('Infinity')
>>> c.power(Decimal('0'), Decimal('0'))
Decimal('NaN')
```

```
>>> ExtendedContext.power(7, 7)
Decimal('823543')
>>> ExtendedContext.power(Decimal(7), 7)
Decimal('823543')
```

quantize(*self*, *a*, *b*)

Returns a value equal to 'a' (rounded), having the exponent of 'b'.

The coefficient of the result is derived from that of the left-hand operand. It may be rounded using the current rounding setting (if the exponent is being increased), multiplied by a positive power of ten (if the exponent is being decreased), or is unchanged (if the exponent is already equal to that of the right-hand operand).

Unlike other operations, if the length of the coefficient after the quantize operation would be greater than

precision then an Invalid operation condition is raised. This guarantees that, unless there is an error condition, the exponent of the result of a quantize is always equal to that of the right-hand operand.

Also unlike other operations, quantize will never raise Underflow, even if the result is subnormal and inexact.

```
>>> ExtendedContext.quantize(Decimal('2.17'), Decimal('0.001'))
Decimal('2.170')
>>> ExtendedContext.quantize(Decimal('2.17'), Decimal('0.01'))
Decimal('2.17')
>>> ExtendedContext.quantize(Decimal('2.17'), Decimal('0.1'))
Decimal('2.2')
>>> ExtendedContext.quantize(Decimal('2.17'), Decimal('1e+0'))
Decimal('2')
>>> ExtendedContext.quantize(Decimal('2.17'), Decimal('1e+1'))
Decimal('0E+1')
>>> ExtendedContext.quantize(Decimal('-Inf'), Decimal('Infinity'))
Decimal('-Infinity')
>>> ExtendedContext.quantize(Decimal('2'), Decimal('Infinity'))
Decimal('NaN')
>>> ExtendedContext.quantize(Decimal('-0.1'), Decimal('1'))
Decimal('-0')
>>> ExtendedContext.quantize(Decimal('-0'), Decimal('1e+5'))
Decimal('-0E+5')
>>> ExtendedContext.quantize(Decimal('+35236450.6'), Decimal('1e-2'))
Decimal('NaN')
>>> ExtendedContext.quantize(Decimal('-35236450.6'), Decimal('1e-2'))
Decimal('NaN')
>>> ExtendedContext.quantize(Decimal('217'), Decimal('1e-1'))
Decimal('217.0')
>>> ExtendedContext.quantize(Decimal('217'), Decimal('1e-0'))
Decimal('217')
>>> ExtendedContext.quantize(Decimal('217'), Decimal('1e+1'))
Decimal('2.2E+2')
>>> ExtendedContext.quantize(Decimal('217'), Decimal('1e+2'))
Decimal('2E+2')
>>> ExtendedContext.quantize(1, 2)
Decimal('1')
>>> ExtendedContext.quantize(Decimal(1), 2)
Decimal('1')
>>> ExtendedContext.quantize(1, Decimal(2))
Decimal('1')
```

radix(*self*)

Just returns 10, as this is Decimal, :)

```
>>> ExtendedContext.radix()
Decimal('10')
```

remainder(*self*, *a*, *b*)

Returns the remainder from integer division.

The result is the residue of the dividend after the operation of calculating integer division as described for divide-integer, rounded to precision digits if necessary. The sign of the result, if non-zero, is the same as that of the original dividend.

This operation will fail under the same conditions as integer division (that is, if integer division on the same two operands would fail, the remainder cannot be calculated).

```
>>> ExtendedContext.remainder(Decimal('2.1'), Decimal('3'))
Decimal('2.1')
>>> ExtendedContext.remainder(Decimal('10'), Decimal('3'))
Decimal('1')
>>> ExtendedContext.remainder(Decimal('-10'), Decimal('3'))
Decimal('-1')
>>> ExtendedContext.remainder(Decimal('10.2'), Decimal('1'))
Decimal('0.2')
>>> ExtendedContext.remainder(Decimal('10'), Decimal('0.3'))
Decimal('0.1')
>>> ExtendedContext.remainder(Decimal('3.6'), Decimal('1.3'))
Decimal('1.0')
>>> ExtendedContext.remainder(22, 6)
Decimal('4')
>>> ExtendedContext.remainder(Decimal(22), 6)
Decimal('4')
>>> ExtendedContext.remainder(22, Decimal(6))
Decimal('4')
```

remainder_near(*self*, *a*, *b*)

Returns to be "a - b * n", where n is the integer nearest the exact value of "x / b" (if two integers are equally near then the even one is chosen). If the result is equal to 0 then its sign will be the sign of a.

This operation will fail under the same conditions as integer division (that is, if integer division on the same two operands would fail, the remainder cannot be calculated).

```
>>> ExtendedContext.remainder_near(Decimal('2.1'), Decimal('3'))
Decimal('-0.9')
>>> ExtendedContext.remainder_near(Decimal('10'), Decimal('6'))
Decimal('-2')
>>> ExtendedContext.remainder_near(Decimal('10'), Decimal('3'))
Decimal('1')
>>> ExtendedContext.remainder_near(Decimal('-10'), Decimal('3'))
Decimal('-1')
>>> ExtendedContext.remainder_near(Decimal('10.2'), Decimal('1'))
Decimal('0.2')
>>> ExtendedContext.remainder_near(Decimal('10'), Decimal('0.3'))
Decimal('0.1')
>>> ExtendedContext.remainder_near(Decimal('3.6'), Decimal('1.3'))
Decimal('-0.3')
>>> ExtendedContext.remainder_near(3, 11)
Decimal('3')
>>> ExtendedContext.remainder_near(Decimal(3), 11)
Decimal('3')
>>> ExtendedContext.remainder_near(3, Decimal(11))
Decimal('3')
```

same_quantum(*self*, *a*, *b*)

Returns True if the two operands have the same exponent.

The result is never affected by either the sign or the coefficient of either operand.

```
>>> ExtendedContext.same_quantum(Decimal('2.17'), Decimal('0.001'))
False
>>> ExtendedContext.same_quantum(Decimal('2.17'), Decimal('0.01'))
True
>>> ExtendedContext.same_quantum(Decimal('2.17'), Decimal('1'))
False
>>> ExtendedContext.same_quantum(Decimal('Inf'), Decimal('-Inf'))
True
>>> ExtendedContext.same_quantum(10000, -1)
True
>>> ExtendedContext.same_quantum(Decimal(10000), -1)
True
>>> ExtendedContext.same_quantum(10000, Decimal(-1))
True
```

scaleb(*self*, *a*, *b*)

Returns the first operand after adding the second value its exp.

```
>>> ExtendedContext.scaleb(Decimal('7.50'), Decimal('-2'))
Decimal('0.0750')
>>> ExtendedContext.scaleb(Decimal('7.50'), Decimal('0'))
Decimal('7.50')
>>> ExtendedContext.scaleb(Decimal('7.50'), Decimal('3'))
Decimal('7.50E+3')
>>> ExtendedContext.scaleb(1, 4)
Decimal('1E+4')
>>> ExtendedContext.scaleb(Decimal(1), 4)
Decimal('1E+4')
>>> ExtendedContext.scaleb(1, Decimal(4))
Decimal('1E+4')
```

sqrt(*self*, *a*)

Square root of a non-negative number to context precision.

If the result must be inexact, it is rounded using the round-half-even algorithm.

```
>>> ExtendedContext.sqrt(Decimal('0'))
Decimal('0')
>>> ExtendedContext.sqrt(Decimal('-0'))
Decimal('-0')
>>> ExtendedContext.sqrt(Decimal('0.39'))
Decimal('0.624499800')
>>> ExtendedContext.sqrt(Decimal('100'))
Decimal('10')
>>> ExtendedContext.sqrt(Decimal('1'))
Decimal('1')
>>> ExtendedContext.sqrt(Decimal('1.0'))
Decimal('1.0')
>>> ExtendedContext.sqrt(Decimal('1.00'))
Decimal('1.0')
>>> ExtendedContext.sqrt(Decimal('7'))
Decimal('2.64575131')
>>> ExtendedContext.sqrt(Decimal('10'))
```

(continues on next page)

(continued from previous page)

```
Decimal('3.16227766')
>>> ExtendedContext.sqrt(2)
Decimal('1.41421356')
>>> ExtendedContext.prec
9
```

subtract(*self*, *a*, *b*)

Return the difference between the two operands.

```
>>> ExtendedContext.subtract(Decimal('1.3'), Decimal('1.07'))
Decimal('0.23')
>>> ExtendedContext.subtract(Decimal('1.3'), Decimal('1.30'))
Decimal('0.00')
>>> ExtendedContext.subtract(Decimal('1.3'), Decimal('2.07'))
Decimal('-0.77')
>>> ExtendedContext.subtract(8, 5)
Decimal('3')
>>> ExtendedContext.subtract(Decimal(8), 5)
Decimal('3')
>>> ExtendedContext.subtract(8, Decimal(5))
Decimal('3')
```

to_eng_string(*self*, *a*)

Convert to a string, using engineering notation if an exponent is needed.

Engineering notation has an exponent which is a multiple of 3. This can leave up to 3 digits to the left of the decimal place and may require the addition of either one or two trailing zeros.

The operation is not affected by the context.

```
>>> ExtendedContext.to_eng_string(Decimal('123E+1'))
'1.23E+3'
>>> ExtendedContext.to_eng_string(Decimal('123E+3'))
'123E+3'
>>> ExtendedContext.to_eng_string(Decimal('123E-10'))
'12.3E-9'
>>> ExtendedContext.to_eng_string(Decimal('-123E-12'))
'-123E-12'
>>> ExtendedContext.to_eng_string(Decimal('7E-7'))
'700E-9'
>>> ExtendedContext.to_eng_string(Decimal('7E+1'))
'70'
>>> ExtendedContext.to_eng_string(Decimal('0E+1'))
'0.00E+3'
```

to_sci_string(*self*, *a*)

Converts a number to a string, using scientific notation.

The operation is not affected by the context.

to_integral_exact(*self*, *a*)

Rounds to an integer.

When the operand has a negative exponent, the result is the same as using the quantize() operation using the given operand as the left-hand-operand, 1E+0 as the right-hand-operand, and the precision of the operand

as the precision setting; Inexact and Rounded flags are allowed in this operation. The rounding mode is taken from the context.

```
>>> ExtendedContext.to_integral_exact(Decimal('2.1'))
Decimal('2')
>>> ExtendedContext.to_integral_exact(Decimal('100'))
Decimal('100')
>>> ExtendedContext.to_integral_exact(Decimal('100.0'))
Decimal('100')
>>> ExtendedContext.to_integral_exact(Decimal('101.5'))
Decimal('102')
>>> ExtendedContext.to_integral_exact(Decimal('-101.5'))
Decimal('-102')
>>> ExtendedContext.to_integral_exact(Decimal('10E+5'))
Decimal('1.0E+6')
>>> ExtendedContext.to_integral_exact(Decimal('7.89E+77'))
Decimal('7.89E+77')
>>> ExtendedContext.to_integral_exact(Decimal('-Inf'))
Decimal('-Infinity')
```

to_integral_value(*self*, *a*)

Rounds to an integer.

When the operand has a negative exponent, the result is the same as using the quantize() operation using the given operand as the left-hand-operand, 1E+0 as the right-hand-operand, and the precision of the operand as the precision setting, except that no flags will be set. The rounding mode is taken from the context.

```
>>> ExtendedContext.to_integral_value(Decimal('2.1'))
Decimal('2')
>>> ExtendedContext.to_integral_value(Decimal('100'))
Decimal('100')
>>> ExtendedContext.to_integral_value(Decimal('100.0'))
Decimal('100')
>>> ExtendedContext.to_integral_value(Decimal('101.5'))
Decimal('102')
>>> ExtendedContext.to_integral_value(Decimal('-101.5'))
Decimal('-102')
>>> ExtendedContext.to_integral_value(Decimal('10E+5'))
Decimal('1.0E+6')
>>> ExtendedContext.to_integral_value(Decimal('7.89E+77'))
Decimal('7.89E+77')
>>> ExtendedContext.to_integral_value(Decimal('-Inf'))
Decimal('-Infinity')
```

jepler_udecimal.**DefaultContext**

jepler_udecimal.**BasicContext**

jepler_udecimal.**ExtendedContext**

CHAPTER

SEVEN

INDICES AND TABLES

- genindex
- modindex
- search

PYTHON MODULE INDEX

j

INDEX

Symbols

A

B

C