10.4. `decimal` — Decimal fixed point and floating point arithmetic¶

New in version 2.4.

The `decimal` module provides support for decimal floating point arithmetic. It offers several advantages over the `float` datatype:

> Decimal "is based on a floating-point model which was designed with people in mind, and necessarily has a paramount guiding principle – computers must provide an arithmetic that works in the same way as the arithmetic that people learn at school." – excerpt from the decimal arithmetic specification.
>
> Decimal numbers can be represented exactly. In contrast, numbers like `1.1` do not have an exact representation in binary floating point. End users typically would not expect `1.1` to display as `1.1000000000000001` as it does with binary floating point.
>
> The exactness carries over into arithmetic. In decimal floating point, `0.1 + 0.1 + 0.1 - 0.3` is exactly equal to zero. In binary floating point, the result is `5.5511151231257827e-017`. While near to zero, the differences prevent reliable equality testing and differences can accumulate. For this reason, decimal is preferred in accounting applications which have strict equality invariants.
>
> The decimal module incorporates a notion of significant places so that `1.30 + 1.20` is `2.50`. The trailing zero is kept to indicate significance. This is the customary presentation for monetary applications. For multiplication, the "schoolbook" approach uses all the figures in the multiplicands. For instance, `1.3 * 1.2` gives `1.56` while `1.30 * 1.20` gives `1.5600`.
>
> Unlike hardware based binary floating point, the decimal module has a user alterable precision (defaulting to 28 places) which can be as large as needed for a given problem:

```
>>> getcontext().prec = 6
>>> Decimal(1) / Decimal(7)
Decimal('0.142857')
>>> getcontext().prec = 28
>>> Decimal(1) / Decimal(7)
Decimal('0.1428571428571428571428571429')
```

> Both binary and decimal floating point are implemented in terms of published standards. While the built-in float type exposes only a modest portion of its capabilities, the decimal module exposes all required parts of the standard. When needed, the programmer has full control over rounding and signal handling. This includes an option to enforce exact arithmetic by using exceptions to block any inexact operations.
>
> The decimal module was designed to support "without prejudice, both exact unrounded decimal arithmetic (sometimes called fixed-point arithmetic) and rounded floating-point arithmetic." – excerpt from the decimal arithmetic specification.

The module design is centered around three concepts: the decimal number, the context for arithmetic, and signals.

A decimal number is immutable. It has a sign, coefficient digits, and an exponent. To preserve significance, the coefficient digits do not truncate trailing zeros. Decimals also include special values such as `Infinity`, `-Infinity`, and `NaN`. The standard also differentiates `-0` from `+0`.

The context for arithmetic is an environment specifying precision, rounding rules, limits on exponents, flags indicating the results of operations, and trap enablers which determine whether signals are treated as exceptions. Rounding options include `ROUND_CEILING`, `ROUND_DOWN`, `ROUND_FLOOR`, `ROUND_HALF_DOWN`, `ROUND_HALF_EVEN`, `ROUND_HALF_UP`, `ROUND_UP`, and `ROUND_05UP`.

Signals are groups of exceptional conditions arising during the course of computation. Depending on the needs of the application, signals may be ignored, considered as informational, or treated as exceptions. The signals in the decimal module are: `Clamped`, `InvalidOperation`, `DivisionByZero`, `Inexact`, `Rounded`, `Subnormal`, `Overflow`, and `Underflow`.

For each signal there is a flag and a trap enabler. When a signal is encountered, its flag is set to one, then, if the trap enabler is set to one, an exception is raised. Flags are sticky, so the user needs to reset them before monitoring a calculation.

See also

- IBM's General Decimal Arithmetic Specification, The General Decimal Arithmetic Specification.
- IEEE standard 854-1987, Unofficial IEEE 854 Text.

10.4.1. Quick-start Tutorial¶

The usual start to using decimals is importing the module, viewing the current context with `getcontext()` and, if necessary, setting new values for precision, rounding, or enabled traps:

```
>>> from decimal import *
>>> getcontext()
Context(prec=28, rounding=ROUND_HALF_EVEN, Emin=-999999999, Emax=999999999,
       capitals=1, flags=[], traps=[Overflow, DivisionByZero,
       InvalidOperation])

>>> getcontext().prec = 7       # Set a new precision
```

Decimal instances can be constructed from integers, strings, or tuples. To create a Decimal from a `float`, first convert it to a string. This serves as an explicit reminder of the details of the conversion (including representation error). Decimal numbers include special values such as `NaN` which stands for “Not a number”, positive and negative `Infinity`, and `-0`.

```
>>> getcontext().prec = 28
>>> Decimal(10)
Decimal('10')
>>> Decimal('3.14')
Decimal('3.14')
>>> Decimal((0, (3, 1, 4), -2))
Decimal('3.14')
>>> Decimal(str(2.0 ** 0.5))
Decimal('1.41421356237')
>>> Decimal(2) ** Decimal('0.5')
Decimal('1.41421356237309504880168872')
>>> Decimal('NaN')
Decimal('NaN')
>>> Decimal('-Infinity')
Decimal('-Infinity')
```

The significance of a new Decimal is determined solely by the number of digits input. Context precision and rounding only come into play during arithmetic operations.

```
>>> getcontext().prec = 6
>>> Decimal('3.0')
Decimal('3.0')
>>> Decimal('3.1415926535')
Decimal('3.1415926535')
>>> Decimal('3.1415926535') + Decimal('2.7182818285')
Decimal('5.85987')
>>> getcontext().rounding = ROUND_UP
>>> Decimal('3.1415926535') + Decimal('2.7182818285')
Decimal('5.85988')
```

Decimals interact well with much of the rest of Python. Here is a small decimal floating point flying circus:

```
>>> data = map(Decimal, '1.34 1.87 3.45 2.35 1.00 0.03 9.25'.split())
>>> max(data)
Decimal('9.25')
>>> min(data)
Decimal('0.03')
>>> sorted(data)
[Decimal('0.03'), Decimal('1.00'), Decimal('1.34'), Decimal('1.87'),
Decimal('2.35'), Decimal('3.45'), Decimal('9.25')]
>>> sum(data)
Decimal('19.29')
>>> a,b,c = data[:3]
>>> str(a)
'1.34'
>>> float(a)
1.3400000000000001
>>> round(a, 1)     # round() first converts to binary floating point
1.3
>>> int(a)
1
>>> a * 5
Decimal('6.70')
>>> a * b
Decimal('2.5058')
```

```
>>> c % a
Decimal('0.77')
```

And some mathematical functions are also available to Decimal:

```
>>> getcontext().prec = 28
>>> Decimal(2).sqrt()
Decimal('1.4142135623730950488016887242')
>>> Decimal(1).exp()
Decimal('2.7182818284590452353602874713')
>>> Decimal('10').ln()
Decimal('2.3025850929940456840179914547')
>>> Decimal('10').log10()
Decimal('1')
```

The `quantize()` method rounds a number to a fixed exponent. This method is useful for monetary applications that often round results to a fixed number of places:

```
>>> Decimal('7.325').quantize(Decimal('.01'), rounding=ROUND_DOWN)
Decimal('7.32')
>>> Decimal('7.325').quantize(Decimal('1.'), rounding=ROUND_UP)
Decimal('8')
```

As shown above, the `getcontext()` function accesses the current context and allows the settings to be changed. This approach meets the needs of most applications.

For more advanced work, it may be useful to create alternate contexts using the Context() constructor. To make an alternate active, use the `setcontext()` function.

In accordance with the standard, the `Decimal` module provides two ready to use standard contexts, `BasicContext` and `ExtendedContext`. The former is especially useful for debugging because many of the traps are enabled:

```
>>> myothercontext = Context(prec=60, rounding=ROUND_HALF_DOWN)
>>> setcontext(myothercontext)
>>> Decimal(1) / Decimal(7)
Decimal('0.142857142857142857142857142857142857142857142857142857142857')

>>> ExtendedContext
Context(prec=9, rounding=ROUND_HALF_EVEN, Emin=-999999999, Emax=999999999,
        capitals=1, flags=[], traps=[])
>>> setcontext(ExtendedContext)
>>> Decimal(1) / Decimal(7)
Decimal('0.142857143')
>>> Decimal(42) / Decimal(0)
Decimal('Infinity')

>>> setcontext(BasicContext)
>>> Decimal(42) / Decimal(0)
Traceback (most recent call last):
 File "<pyshell#143>", line 1, in -toplevel-
   Decimal(42) / Decimal(0)
DivisionByZero: x / 0
```

Contexts also have signal flags for monitoring exceptional conditions encountered during computations. The flags remain set until explicitly cleared, so it is best to clear the flags before each set of monitored computations by using the `clear_flags()` method.

```
>>> setcontext(ExtendedContext)
>>> getcontext().clear_flags()
>>> Decimal(355) / Decimal(113)
Decimal('3.14159292')
>>> getcontext()
Context(prec=9, rounding=ROUND_HALF_EVEN, Emin=-999999999, Emax=999999999,
        capitals=1, flags=[Rounded, Inexact], traps=[])
```

The *flags* entry shows that the rational approximation to `Pi` was rounded (digits beyond the context precision were thrown away) and that the result is inexact (some of the discarded digits were non-zero).

Individual traps are set using the dictionary in the `traps` field of a context:

```
>>> setcontext(ExtendedContext)
>>> Decimal(1) / Decimal(0)
```

```
Decimal('Infinity')
>>> getcontext().traps[DivisionByZero] = 1
>>> Decimal(1) / Decimal(0)
Traceback (most recent call last):
 File "<pyshell#112>", line 1, in -toplevel-
   Decimal(1) / Decimal(0)
DivisionByZero: x / 0
```

Most programs adjust the current context only once, at the beginning of the program. And, in many applications, data is converted to `Decimal` with a single cast inside a loop. With context set and decimals created, the bulk of the program manipulates the data no differently than with other Python numeric types.

10.4.2. Decimal objects¶

class `decimal.Decimal`([*value*[, *context*]])¶

Construct a new `Decimal` object based from *value*.

value can be an integer, string, tuple, or another `Decimal` object. If no *value* is given, returns `Decimal('0')`. If *value* is a string, it should conform to the decimal numeric string syntax after leading and trailing whitespace characters are removed:

```
sign           ::=  '+' | '-'
digit          ::=  '0' | '1' | '2' | '3' | '4' | '5' | '6' | '7' | '8' | '9'
indicator      ::=  'e' | 'E'
digits         ::=  digit [digit]...
decimal-part   ::=  digits '.' [digits] | ['.'] digits
exponent-part  ::=  indicator [sign] digits
infinity       ::=  'Infinity' | 'Inf'
nan            ::=  'NaN' [digits] | 'sNaN' [digits]
numeric-value  ::=  decimal-part [exponent-part] | infinity
numeric-string ::=  [sign] numeric-value | [sign] nan
```

If *value* is a unicode string then other Unicode decimal digits are also permitted where `digit` appears above. These include decimal digits from various other alphabets (for example, Arabic-Indic and Devan■gar■ digits) along with the fullwidth digits `u'\uff10'` through `u'\uff19'`.

If *value* is a `tuple`, it should have three components, a sign (`0` for positive or `1` for negative), a `tuple` of digits, and an integer exponent. For example, `Decimal((0, (1, 4, 1, 4), -3))` returns `Decimal('1.414')`.

The *context* precision does not affect how many digits are stored. That is determined exclusively by the number of digits in *value*. For example, `Decimal('3.00000')` records all five zeros even if the context precision is only three.

The purpose of the *context* argument is determining what to do if *value* is a malformed string. If the context traps `InvalidOperation`, an exception is raised; otherwise, the constructor returns a new Decimal with the value of `NaN`.

Once constructed, `Decimal` objects are immutable.

Changed in version 2.6: leading and trailing whitespace characters are permitted when creating a Decimal instance from a string.

Decimal floating point objects share many properties with the other built-in numeric types such as `float` and `int`. All of the usual math operations and special methods apply. Likewise, decimal objects can be copied, pickled, printed, used as dictionary keys, used as set elements, compared, sorted, and coerced to another type (such as `float` or `long`).

In addition to the standard numeric properties, decimal floating point objects also have a number of specialized methods:

`adjusted`()¶
Return the adjusted exponent after shifting out the coefficient's rightmost digits until only the lead digit remains: `Decimal('321e+5').adjusted()` returns seven. Used for determining the position of the most significant digit with respect to the decimal point.

`as_tuple`()¶

Return a *named tuple* representation of the number: `DecimalTuple(sign, digits, exponent)`.

Changed in version 2.6: Use a named tuple.

`canonical`()¶

Return the canonical encoding of the argument. Currently, the encoding of a `Decimal` instance is always canonical, so this operation returns its argument unchanged.

New in version 2.6.

`compare`(*other*[, *context*])¶

Compare the values of two Decimal instances. This operation behaves in the same way as the usual comparison method `__cmp__()`, except that `compare()` returns a Decimal instance rather than an integer, and if either operand is a NaN then the result is a NaN:

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

`compare_signal`(*other*[, *context*])¶

This operation is identical to the `compare()` method, except that all NaNs signal. That is, if neither operand is a signaling NaN then any quiet NaN operand is treated as though it were a signaling NaN.

New in version 2.6.

`compare_total`(*other*)¶

Compare two operands using their abstract representation rather than their numerical value. Similar to the `compare()` method, but the result gives a total ordering on `Decimal` instances. Two `Decimal` instances with the same numeric value but different representations compare unequal in this ordering:

```
>>> Decimal('12.0').compare_total(Decimal('12'))
Decimal('-1')
```

Quiet and signaling NaNs are also included in the total ordering. The result of this function is `Decimal('0')` if both operands have the same representation, `Decimal('-1')` if the first operand is lower in the total order than the second, and `Decimal('1')` if the first operand is higher in the total order than the second operand. See the specification for details of the total order.

New in version 2.6.

`compare_total_mag`(*other*)¶

Compare two operands using their abstract representation rather than their value as in `compare_total()`, but ignoring the sign of each operand. `x.compare_total_mag(y)` is equivalent to `x.copy_abs().compare_total(y.copy_abs())`.

New in version 2.6.

`conjugate`()¶

Just returns self, this method is only to comply with the Decimal Specification.

New in version 2.6.

`copy_abs`()¶

Return the absolute value of the argument. This operation is unaffected by the context and is quiet: no flags are changed and no rounding is performed.

New in version 2.6.

`copy_negate`()¶

Return the negation of the argument. This operation is unaffected by the context and is quiet: no flags are changed and no rounding is performed.

New in version 2.6.

`copy_sign`(*other*)¶

Return a copy of the first operand with the sign set to be the same as the sign of the second operand. For example:

```
>>> Decimal('2.3').copy_sign(Decimal('-1.5'))
Decimal('-2.3')
```

This operation is unaffected by the context and is quiet: no flags are changed and no rounding is performed.

New in version 2.6.

`exp`([*context*])¶

Return the value of the (natural) exponential function `e**x` at the given number. The result is correctly rounded using the `ROUND_HALF_EVEN` rounding mode.

```
>>> Decimal(1).exp()
Decimal('2.718281828459045235360287471')
>>> Decimal(321).exp()
Decimal('2.561702493119680037517373933E+139')
```

New in version 2.6.

`fma`(*other*, *third*[, *context*])¶

Fused multiply-add. Return self*other+third with no rounding of the intermediate product self*other.

```
>>> Decimal(2).fma(3, 5)
Decimal('11')
```

New in version 2.6.

`is_canonical`()¶

Return `True` if the argument is canonical and `False` otherwise. Currently, a `Decimal` instance is always canonical, so this operation always returns `True`.

New in version 2.6.

`is_finite`()¶

Return `True` if the argument is a finite number, and `False` if the argument is an infinity or a NaN.

New in version 2.6.

`is_infinite`()¶

Return `True` if the argument is either positive or negative infinity and `False` otherwise.

New in version 2.6.

`is_nan`()¶

Return `True` if the argument is a (quiet or signaling) NaN and `False` otherwise.

New in version 2.6.

`is_normal`()¶

Return `True` if the argument is a *normal* finite non-zero number with an adjusted exponent greater than or equal to *Emin*. Return `False` if the argument is zero, subnormal, infinite or a NaN. Note, the term *normal* is used here in a different sense with the `normalize()` method which is used to create canonical values.

New in version 2.6.

`is_qnan`()¶

Return `True` if the argument is a quiet NaN, and `False` otherwise.

New in version 2.6.

`is_signed`()¶

Return `True` if the argument has a negative sign and `False` otherwise. Note that zeros and NaNs can both carry signs.

New in version 2.6.

`is_snan`()¶

Return `True` if the argument is a signaling NaN and `False` otherwise.

New in version 2.6.

`is_subnormal`()¶

Return `True` if the argument is subnormal, and `False` otherwise. A number is subnormal is if it is nonzero, finite, and has an adjusted exponent less than *Emin*.

New in version 2.6.

`is_zero`()¶

Return `True` if the argument is a (positive or negative) zero and `False` otherwise.

New in version 2.6.

`ln`([*context*])¶

Return the natural (base e) logarithm of the operand. The result is correctly rounded using the `ROUND_HALF_EVEN` rounding mode.

New in version 2.6.

`log10`(*[context]*)¶

Return the base ten logarithm of the operand. The result is correctly rounded using the `ROUND_HALF_EVEN` rounding mode.

New in version 2.6.

`logb`(*[context]*)¶

For a nonzero number, return the adjusted exponent of its operand as a `Decimal` instance. If the operand is a zero then `Decimal('-Infinity')` is returned and the `DivisionByZero` flag is raised. If the operand is an infinity then `Decimal('Infinity')` is returned.

New in version 2.6.

`logical_and`(*other*[, *context*])¶

`logical_and()` is a logical operation which takes two *logical operands* (see *Logical operands*). The result is the digit-wise `and` of the two operands.

New in version 2.6.

`logical_invert`(*[context]*)¶

`logical_invert()` is a logical operation. The result is the digit-wise inversion of the operand.

New in version 2.6.

`logical_or`(*other*[, *context*])¶

`logical_or()` is a logical operation which takes two *logical operands* (see *Logical operands*). The result is the digit-wise `or` of the two operands.

New in version 2.6.

`logical_xor`(*other*[, *context*])¶

`logical_xor()` is a logical operation which takes two *logical operands* (see *Logical operands*). The result is the digit-wise exclusive or of the two operands.

New in version 2.6.

`max`(*other*[, *context*])¶
Like `max(self, other)` except that the context rounding rule is applied before returning and that `NaN` values are either signaled or ignored (depending on the context and whether they are signaling or quiet).

`max_mag`(*other*[, *context*])¶

Similar to the `max()` method, but the comparison is done using the absolute values of the operands.

New in version 2.6.

`min`(*other*[, *context*])¶
Like `min(self, other)` except that the context rounding rule is applied before returning and that `NaN` values are either signaled or ignored (depending on the context and whether they are signaling or quiet).

`min_mag`(*other*[, *context*])¶

Similar to the `min()` method, but the comparison is done using the absolute values of the operands.

New in version 2.6.

`next_minus`(*[context]*)¶

Return the largest number representable in the given context (or in the current thread's context if no context is given) that is smaller than the given operand.

New in version 2.6.

`next_plus`(*[context]*)¶

Return the smallest number representable in the given context (or in the current thread's context if no context is given) that is larger than the given operand.

New in version 2.6.

`next_toward`(*other*[, *context*])¶

If the two operands are unequal, return the number closest to the first operand in the direction of the second operand. If both operands are numerically equal, return a copy of the first operand with the sign set to be the same as the sign of the second operand.

New in version 2.6.

`normalize`([*context*])¶
Normalize the number by stripping the rightmost trailing zeros and converting any result equal to `Decimal('0')` to `Decimal('0e0')`. Used for producing canonical values for members of an equivalence class. For example, `Decimal('32.100')` and `Decimal('0.321000e+2')` both normalize to the equivalent value `Decimal('32.1')`.

`number_class`([*context*])¶

Return a string describing the *class* of the operand. The returned value is one of the following ten strings.

- `"-Infinity"`, indicating that the operand is negative infinity.
- `"-Normal"`, indicating that the operand is a negative normal number.
- `"-Subnormal"`, indicating that the operand is negative and subnormal.
- `"-Zero"`, indicating that the operand is a negative zero.
- `"+Zero"`, indicating that the operand is a positive zero.
- `"+Subnormal"`, indicating that the operand is positive and subnormal.
- `"+Normal"`, indicating that the operand is a positive normal number.
- `"+Infinity"`, indicating that the operand is positive infinity.
- `"NaN"`, indicating that the operand is a quiet NaN (Not a Number).
- `"sNaN"`, indicating that the operand is a signaling NaN.

New in version 2.6.

`quantize`(*exp*[, *rounding*[, *context*[, *watchexp*]]])¶

Return a value equal to the first operand after rounding and having the exponent of the second operand.

```
>>> Decimal('1.41421356').quantize(Decimal('1.000'))
Decimal('1.414')
```

Unlike other operations, if the length of the coefficient after the quantize operation would be greater than precision, then an `InvalidOperation` is signaled. This guarantees that, unless there is an error condition, the quantized exponent is always equal to that of the right-hand operand.

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

If the exponent of the second operand is larger than that of the first then rounding may be necessary. In this case, the rounding mode is determined by the `rounding` argument if given, else by the given `context` argument; if neither argument is given the rounding mode of the current thread's context is used.

If *watchexp* is set (default), then an error is returned whenever the resulting exponent is greater than `Emax` or less than `Etiny`.

`radix`()¶

Return `Decimal(10)`, the radix (base) in which the `Decimal` class does all its arithmetic. Included for compatibility with the specification.

New in version 2.6.

`remainder_near`(*other*[, *context*])¶

Compute the modulo as either a positive or negative value depending on which is closest to zero. For instance, `Decimal(10).remainder_near(6)` returns `Decimal('-2')` which is closer to zero than `Decimal('4')`.

If both are equally close, the one chosen will have the same sign as *self*.

`rotate`(*other*[, *context*])¶

Return the result of rotating the digits of the first operand by an amount specified by the second operand. The second operand must be an integer in the range -precision through precision. The absolute value of the second operand gives the number of places to rotate. If the second operand is positive then rotation is to the left; otherwise rotation is to the right. The coefficient of the first operand is padded on the left with zeros to length precision if necessary. The sign and exponent of the first operand are unchanged.

New in version 2.6.

`same_quantum`(*other*[, *context*])¶
Test whether self and other have the same exponent or whether both are `NaN`.

`scaleb`(*other*[, *context*])¶

Return the first operand with exponent adjusted by the second. Equivalently, return the first operand multiplied by `10**other`. The second operand must be an integer.

New in version 2.6.

`shift`(*other*[, *context*])¶

Return the result of shifting the digits of the first operand by an amount specified by the second operand. The second operand must be an integer in the range -precision through precision. The absolute value of the second operand gives the number of places to shift. If the second operand is positive then the shift is to the left; otherwise the shift is to the right. Digits shifted into the coefficient are zeros. The sign and exponent of the first operand are unchanged.

New in version 2.6.

`sqrt`([*context*])¶
Return the square root of the argument to full precision.

`to_eng_string`([*context*])¶

Convert to an engineering-type string.

Engineering notation has an exponent which is a multiple of 3, so there are up to 3 digits left of the decimal place. For example, converts `Decimal('123E+1')` to `Decimal('1.23E+3')`

`to_integral`([*rounding*[, *context*]])¶
Identical to the `to_integral_value()` method. The `to_integral` name has been kept for compatibility with older versions.

`to_integral_exact`([*rounding*[, *context*]])¶

Round to the nearest integer, signaling `Inexact` or `Rounded` as appropriate if rounding occurs. The rounding mode is determined by the `rounding` parameter if given, else by the given `context`. If neither parameter is given then the rounding mode of the current context is used.

New in version 2.6.

`to_integral_value`([*rounding*[, *context*]])¶

Round to the nearest integer without signaling `Inexact` or `Rounded`. If given, applies *rounding*; otherwise, uses the rounding method in either the supplied *context* or the current context.

Changed in version 2.6: renamed from `to_integral` to `to_integral_value`. The old name remains valid for compatibility.

10.4.2.1. Logical operands¶

The `logical_and()`, `logical_invert()`, `logical_or()`, and `logical_xor()` methods expect their arguments to be *logical operands*. A *logical operand* is a `Decimal` instance whose exponent and sign are both zero, and whose digits are all either `0` or `1`.

10.4.3. Context objects¶

Contexts are environments for arithmetic operations. They govern precision, set rules for rounding, determine which signals are treated as exceptions, and limit the range for exponents.

Each thread has its own current context which is accessed or changed using the `getcontext()` and `setcontext()` functions:

`decimal.getcontext`()¶
Return the current context for the active thread.

`decimal.setcontext`(*c*)¶
Set the current context for the active thread to *c*.

Beginning with Python 2.5, you can also use the `with` statement and the `localcontext()` function to temporarily change the active context.

`decimal.localcontext`([*c*])¶

Return a context manager that will set the current context for the active thread to a copy of *c* on entry to the with-statement and restore the previous context when exiting the with-statement. If no context is specified, a copy of the current context is used.

New in version 2.5.

For example, the following code sets the current decimal precision to 42 places, performs a calculation, and then automatically restores the previous context:

```
from decimal import localcontext

with localcontext() as ctx:
   ctx.prec = 42   # Perform a high precision calculation
   s = calculate_something()
s = +s  # Round the final result back to the default precision
```

New contexts can also be created using the `Context` constructor described below. In addition, the module provides three pre-made contexts:

class `decimal.BasicContext`¶

This is a standard context defined by the General Decimal Arithmetic Specification. Precision is set to nine. Rounding is set to `ROUND_HALF_UP`. All flags are cleared. All traps are enabled (treated as exceptions) except `Inexact`, `Rounded`, and `Subnormal`.

Because many of the traps are enabled, this context is useful for debugging.

class `decimal.ExtendedContext`¶

This is a standard context defined by the General Decimal Arithmetic Specification. Precision is set to nine. Rounding is set to `ROUND_HALF_EVEN`. All flags are cleared. No traps are enabled (so that exceptions are not raised during computations).

Because the traps are disabled, this context is useful for applications that prefer to have result value of `NaN` or `Infinity` instead of raising exceptions. This allows an application to complete a run in the presence of conditions that would otherwise halt the program.

class `decimal.DefaultContext`¶

This context is used by the `Context` constructor as a prototype for new contexts. Changing a field (such a precision) has the effect of changing the default for new contexts creating by the `Context` constructor.

This context is most useful in multi-threaded environments. Changing one of the fields before threads are started has the effect of setting system-wide defaults. Changing the fields after threads have started is not recommended as it would require thread synchronization to prevent race conditions.

In single threaded environments, it is preferable to not use this context at all. Instead, simply create contexts explicitly as described below.

The default values are precision=28, rounding=ROUND_HALF_EVEN, and enabled traps for Overflow, InvalidOperation, and DivisionByZero.

In addition to the three supplied contexts, new contexts can be created with the `Context` constructor.

class `decimal.Context`(*prec=None*, *rounding=None*, *traps=None*, *flags=None*, *Emin=None*, *Emax=None*, *capitals=1*)¶

Creates a new context. If a field is not specified or is `None`, the default values are copied from the `DefaultContext`. If the *flags* field is not specified or is `None`, all flags are cleared.

The *prec* field is a positive integer that sets the precision for arithmetic operations in the context.

The *rounding* option is one of:

- `ROUND_CEILING` (towards `Infinity`),
- `ROUND_DOWN` (towards zero),
- `ROUND_FLOOR` (towards `-Infinity`),
- `ROUND_HALF_DOWN` (to nearest with ties going towards zero),
- `ROUND_HALF_EVEN` (to nearest with ties going to nearest even integer),
- `ROUND_HALF_UP` (to nearest with ties going away from zero), or
- `ROUND_UP` (away from zero).
- `ROUND_05UP` (away from zero if last digit after rounding towards zero would have been 0 or 5; otherwise towards zero)

The *traps* and *flags* fields list any signals to be set. Generally, new contexts should only set traps and leave the flags clear.

The *Emin* and *Emax* fields are integers specifying the outer limits allowable for exponents.

The *capitals* field is either `0` or `1` (the default). If set to `1`, exponents are printed with a capital `E`; otherwise, a lowercase `e` is used: `Decimal('6.02e+23')`.

Changed in version 2.6: The `ROUND_05UP` rounding mode was added.

The `Context` class defines several general purpose methods as well as a large number of methods for doing arithmetic directly in a given context. In addition, for each of the `Decimal` methods described above (with the exception of the `adjusted()` and `as_tuple()` methods) there is a corresponding `Context` method. For example, `C.exp(x)` is equivalent to `x.exp(context=C)`.

`clear_flags`()¶
Resets all of the flags to `0`.

`copy`()¶
Return a duplicate of the context.

`copy_decimal`(*num*)¶
Return a copy of the Decimal instance num.

`create_decimal`(*num*)¶

Creates a new Decimal instance from *num* but using *self* as context. Unlike the `Decimal` constructor, the context precision, rounding method, flags, and traps are applied to the conversion.

This is useful because constants are often given to a greater precision than is needed by the application. Another benefit is that rounding immediately eliminates unintended effects from digits beyond the current precision. In the following example, using unrounded inputs means that adding zero to a sum can change the

result:

```
>>> getcontext().prec = 3
>>> Decimal('3.4445') + Decimal('1.0023')
Decimal('4.45')
>>> Decimal('3.4445') + Decimal(0) + Decimal('1.0023')
Decimal('4.44')
```

This method implements the to-number operation of the IBM specification. If the argument is a string, no leading or trailing whitespace is permitted.

`Etiny`()¶
Returns a value equal to `Emin - prec + 1` which is the minimum exponent value for subnormal results. When underflow occurs, the exponent is set to `Etiny`.

`Etop`()¶
Returns a value equal to `Emax - prec + 1`.

The usual approach to working with decimals is to create `Decimal` instances and then apply arithmetic operations which take place within the current context for the active thread. An alternative approach is to use context methods for calculating within a specific context. The methods are similar to those for the `Decimal` class and are only briefly recounted here.

`abs`(*x*)¶
Returns the absolute value of *x*.

`add`(*x*, *y*)¶
Return the sum of *x* and *y*.

`canonical`(*x*)¶
Returns the same Decimal object *x*.

`compare`(*x*, *y*)¶
Compares *x* and *y* numerically.

`compare_signal`(*x*, *y*)¶
Compares the values of the two operands numerically.

`compare_total`(*x*, *y*)¶
Compares two operands using their abstract representation.

`compare_total_mag`(*x*, *y*)¶
Compares two operands using their abstract representation, ignoring sign.

`copy_abs`(*x*)¶
Returns a copy of *x* with the sign set to 0.

`copy_negate`(*x*)¶
Returns a copy of *x* with the sign inverted.

`copy_sign`(*x*, *y*)¶
Copies the sign from *y* to *x*.

`divide`(*x*, *y*)¶
Return *x* divided by *y*.

`divide_int`(*x*, *y*)¶
Return *x* divided by *y*, truncated to an integer.

`divmod`(*x*, *y*)¶
Divides two numbers and returns the integer part of the result.

`exp`(*x*)¶
Returns e ** x.

`fma`(*x*, *y*, *z*)¶
Returns *x* multiplied by *y*, plus *z*.

`is_canonical`(*x*)¶
Returns True if *x* is canonical; otherwise returns False.

`is_finite`(*x*)¶
Returns True if *x* is finite; otherwise returns False.

`is_infinite`(*x*)¶
Returns True if *x* is infinite; otherwise returns False.

`is_nan`(*x*)¶
Returns True if *x* is a qNaN or sNaN; otherwise returns False.

`is_normal`(*x*)¶
Returns True if *x* is a normal number; otherwise returns False.

`is_qnan`(*x*)¶
Returns True if *x* is a quiet NaN; otherwise returns False.

`is_signed`(*x*)¶
Returns True if *x* is negative; otherwise returns False.

`is_snan`(*x*)¶
Returns True if *x* is a signaling NaN; otherwise returns False.

`is_subnormal`(*x*)¶
Returns True if *x* is subnormal; otherwise returns False.

`is_zero`(*x*)¶
Returns True if *x* is a zero; otherwise returns False.

`ln`(*x*)¶
Returns the natural (base e) logarithm of *x*.

`log10`(*x*)¶
Returns the base 10 logarithm of *x*.

`logb`(*x*)¶
Returns the exponent of the magnitude of the operand's MSD.

`logical_and`(*x*, *y*)¶
Applies the logical operation *and* between each operand's digits.

`logical_invert`(*x*)¶
Invert all the digits in *x*.

`logical_or`(*x*, *y*)¶
Applies the logical operation *or* between each operand's digits.

`logical_xor`(*x*, *y*)¶
Applies the logical operation *xor* between each operand's digits.

`max`(*x*, *y*)¶
Compares two values numerically and returns the maximum.

`max_mag`(*x*, *y*)¶
Compares the values numerically with their sign ignored.

`min`(*x*, *y*)¶
Compares two values numerically and returns the minimum.

`min_mag`(*x*, *y*)¶
Compares the values numerically with their sign ignored.

`minus`(*x*)¶
Minus corresponds to the unary prefix minus operator in Python.

`multiply`(*x*, *y*)¶
Return the product of *x* and *y*.

`next_minus`(*x*)¶
Returns the largest representable number smaller than *x*.

`next_plus`(*x*)¶
Returns the smallest representable number larger than *x*.

`next_toward`(*x*, *y*)¶
Returns the number closest to *x*, in direction towards *y*.

`normalize`(*x*)¶
Reduces *x* to its simplest form.

`number_class`(*x*)¶
Returns an indication of the class of *x*.

`plus`(*x*)¶
Plus corresponds to the unary prefix plus operator in Python. This operation applies the context precision and rounding, so it is *not* an identity operation.

`power`(*x*, *y*[, *modulo*])¶

Return x to the power of y, reduced modulo `modulo` if given.

With two arguments, compute `x**y`. If x is negative then y must be integral. The result will be inexact unless y is integral and the result is finite and can be expressed exactly in 'precision' digits. The result should always be correctly rounded, using the rounding mode of the current thread's context.

With three arguments, compute `(x**y) % modulo`. For the three argument form, the following restrictions on the arguments hold:

- all three arguments must be integral
- y must be nonnegative
- at least one of x or y must be nonzero
- `modulo` must be nonzero and have at most 'precision' digits

The value resulting from `Context.power(x, y, modulo)` is equal to the value that would be obtained by computing `(x**y) % modulo` with unbounded precision, but is computed more efficiently. The exponent of the result is zero, regardless of the exponents of `x`, `y` and `modulo`. The result is always exact.

Changed in version 2.6: `y` may now be nonintegral in `x**y`. Stricter requirements for the three-argument version.

`quantize`(*x*, *y*)¶
Returns a value equal to *x* (rounded), having the exponent of *y*.

`radix`()¶
Just returns 10, as this is Decimal, :)

`remainder`(*x*, *y*)¶

Returns the remainder from integer division.

The sign of the result, if non-zero, is the same as that of the original dividend.

`remainder_near`(*x*, *y*)¶
Returns `x - y * n`, where *n* is the integer nearest the exact value of `x / y` (if the result is 0 then its sign will be the sign of *x*).

`rotate`(*x*, *y*)¶
Returns a rotated copy of *x*, *y* times.

`same_quantum`(*x*, *y*)¶
Returns True if the two operands have the same exponent.

`scaleb`(*x*, *y*)¶
Returns the first operand after adding the second value its exp.

`shift`(*x*, *y*)¶
Returns a shifted copy of *x*, *y* times.

`sqrt`(*x*)¶
Square root of a non-negative number to context precision.

`subtract`(*x*, *y*)¶
Return the difference between *x* and *y*.

`to_eng_string`(*x*)¶
Converts a number to a string, using scientific notation.

`to_integral_exact`(*x*)¶
Rounds to an integer.

`to_sci_string`(*x*)¶
Converts a number to a string using scientific notation.

10.4.4. Signals¶

Signals represent conditions that arise during computation. Each corresponds to one context flag and one context trap enabler.

The context flag is set whenever the condition is encountered. After the computation, flags may be checked for informational purposes (for instance, to determine whether a computation was exact). After checking the flags, be sure to clear all flags before starting the next computation.

If the context's trap enabler is set for the signal, then the condition causes a Python exception to be raised. For example, if the `DivisionByZero` trap is set, then a `DivisionByZero` exception is raised upon encountering the condition.

class `decimal.Clamped`¶

Altered an exponent to fit representation constraints.

Typically, clamping occurs when an exponent falls outside the context's `Emin` and `Emax` limits. If possible, the exponent is reduced to fit by adding zeros to the coefficient.

class `decimal.DecimalException`¶
Base class for other signals and a subclass of `ArithmeticError`.
class `decimal.DivisionByZero`¶

Signals the division of a non-infinite number by zero.

Can occur with division, modulo division, or when raising a number to a negative power. If this signal is not trapped, returns `Infinity` or `-Infinity` with the sign determined by the inputs to the calculation.

class `decimal.Inexact`¶

Indicates that rounding occurred and the result is not exact.

Signals when non-zero digits were discarded during rounding. The rounded result is returned. The signal flag or trap is used to detect when results are inexact.

class `decimal.InvalidOperation`¶

An invalid operation was performed.

Indicates that an operation was requested that does not make sense. If not trapped, returns `NaN`. Possible causes include:

```
Infinity - Infinity
0 * Infinity
Infinity / Infinity
x % 0
Infinity % x
x._rescale( non-integer )
sqrt(-x) and x > 0
0 ** 0
x ** (non-integer)
x ** Infinity
```

class `decimal.Overflow`¶

Numerical overflow.

Indicates the exponent is larger than `Emax` after rounding has occurred. If not trapped, the result depends on the rounding mode, either pulling inward to the largest representable finite number or rounding outward to `Infinity`. In either case, `Inexact` and `Rounded` are also signaled.

class `decimal.Rounded`¶

Rounding occurred though possibly no information was lost.

Signaled whenever rounding discards digits; even if those digits are zero (such as rounding `5.00` to `5.0`). If not trapped, returns the result unchanged. This signal is used to detect loss of significant digits.

class `decimal.Subnormal`¶

Exponent was lower than `Emin` prior to rounding.

Occurs when an operation result is subnormal (the exponent is too small). If not trapped, returns the result unchanged.

class `decimal.Underflow`¶

Numerical underflow with result rounded to zero.

Occurs when a subnormal result is pushed to zero by rounding. `Inexact` and `Subnormal` are also signaled.

The following table summarizes the hierarchy of signals:

```
exceptions.ArithmeticError(exceptions.StandardError)
   DecimalException
       Clamped
       DivisionByZero(DecimalException, exceptions.ZeroDivisionError)
       Inexact
           Overflow(Inexact, Rounded)
           Underflow(Inexact, Rounded, Subnormal)
       InvalidOperation
       Rounded
       Subnormal
```

10.4.5. Floating Point Notes¶

10.4.5.1. Mitigating round-off error with increased precision¶

The use of decimal floating point eliminates decimal representation error (making it possible to represent `0.1` exactly); however, some operations can still incur round-off error when non-zero digits exceed the fixed precision.

The effects of round-off error can be amplified by the addition or subtraction of nearly offsetting quantities resulting in loss of significance. Knuth provides two instructive examples where rounded floating point arithmetic with insufficient precision causes the breakdown of the associative and distributive properties of addition:

```
# Examples from Seminumerical Algorithms, Section 4.2.2.
>>> from decimal import Decimal, getcontext
>>> getcontext().prec = 8

>>> u, v, w = Decimal(11111113), Decimal(-11111111), Decimal('7.51111111')
```

```
>>> (u + v) + w
Decimal('9.5111111')
>>> u + (v + w)
Decimal('10')

>>> u, v, w = Decimal(20000), Decimal(-6), Decimal('6.0000003')
>>> (u*v) + (u*w)
Decimal('0.01')
>>> u * (v+w)
Decimal('0.0060000')
```

The `decimal` module makes it possible to restore the identities by expanding the precision sufficiently to avoid loss of significance:

```
>>> getcontext().prec = 20
>>> u, v, w = Decimal(11111113), Decimal(-11111111), Decimal('7.51111111')
>>> (u + v) + w
Decimal('9.51111111')
>>> u + (v + w)
Decimal('9.51111111')
>>>
>>> u, v, w = Decimal(20000), Decimal(-6), Decimal('6.0000003')
>>> (u*v) + (u*w)
Decimal('0.0060000')
>>> u * (v+w)
Decimal('0.0060000')
```

10.4.5.2. Special values¶

The number system for the `decimal` module provides special values including `NaN`, `sNaN`, `-Infinity`, `Infinity`, and two zeros, `+0` and `-0`.

Infinities can be constructed directly with: `Decimal('Infinity')`. Also, they can arise from dividing by zero when the `DivisionByZero` signal is not trapped. Likewise, when the `Overflow` signal is not trapped, infinity can result from rounding beyond the limits of the largest representable number.

The infinities are signed (affine) and can be used in arithmetic operations where they get treated as very large, indeterminate numbers. For instance, adding a constant to infinity gives another infinite result.

Some operations are indeterminate and return `NaN`, or if the `InvalidOperation` signal is trapped, raise an exception. For example, `0/0` returns `NaN` which means “not a number”. This variety of `NaN` is quiet and, once created, will flow through other computations always resulting in another `NaN`. This behavior can be useful for a series of computations that occasionally have missing inputs — it allows the calculation to proceed while flagging specific results as invalid.

A variant is `sNaN` which signals rather than remaining quiet after every operation. This is a useful return value when an invalid result needs to interrupt a calculation for special handling.

The behavior of Python’s comparison operators can be a little surprising where a `NaN` is involved. A test for equality where one of the operands is a quiet or signaling `NaN` always returns `False` (even when doing `Decimal('NaN')==Decimal('NaN')`), while a test for inequality always returns `True`. An attempt to compare two Decimals using any of the <, <=, > or >= operators will raise the `InvalidOperation` signal if either operand is a `NaN`, and return `False` if this signal is not trapped. Note that the General Decimal Arithmetic specification does not specify the behavior of direct comparisons; these rules for comparisons involving a `NaN` were taken from the IEEE 854 standard (see Table 3 in section 5.7). To ensure strict standards-compliance, use the `compare()` and `compare-signal()` methods instead.

The signed zeros can result from calculations that underflow. They keep the sign that would have resulted if the calculation had been carried out to greater precision. Since their magnitude is zero, both positive and negative zeros are treated as equal and their sign is informational.

In addition to the two signed zeros which are distinct yet equal, there are various representations of zero with differing precisions yet equivalent in value. This takes a bit of getting used to. For an eye accustomed to normalized floating point representations, it is not immediately obvious that the following calculation returns a value equal to zero:

```
>>> 1 / Decimal('Infinity')
Decimal('0E-1000000026')
```

10.4.6. Working with threads¶

The `getcontext()` function accesses a different `Context` object for each thread. Having separate thread contexts means that threads may make changes (such as `getcontext.prec=10`) without interfering with other threads.

Likewise, the `setcontext()` function automatically assigns its target to the current thread.

If `setcontext()` has not been called before `getcontext()`, then `getcontext()` will automatically create a new context for use in the current thread.

The new context is copied from a prototype context called *DefaultContext*. To control the defaults so that each thread will use the same values throughout the application, directly modify the *DefaultContext* object. This should be done *before* any threads are started so that there won’t be a race condition between threads

calling getcontext(). For example:

```
# Set applicationwide defaults for all threads about to be launched
DefaultContext.prec = 12
DefaultContext.rounding = ROUND_DOWN
DefaultContext.traps = ExtendedContext.traps.copy()
DefaultContext.traps[InvalidOperation] = 1
setcontext(DefaultContext)

# Afterwards, the threads can be started
t1.start()
t2.start()
t3.start()
. . .
```

10.4.7. Recipes¶

Here are a few recipes that serve as utility functions and that demonstrate ways to work with the Decimal class:

```
def moneyfmt(value, places=2, curr='', sep=',', dp='.',
             pos='', neg='-', trailneg=''):
   """Convert Decimal to a money formatted string.

   places:  required number of places after the decimal point
   curr:    optional currency symbol before the sign (may be blank)
   sep:     optional grouping separator (comma, period, space, or blank)
   dp:      decimal point indicator (comma or period)
            only specify as blank when places is zero
   pos:     optional sign for positive numbers: '+', space or blank
   neg:     optional sign for negative numbers: '-', '(', space or blank
   trailneg:optional trailing minus indicator:  '-', ')', space or blank

   >>> d = Decimal('-1234567.8901')
   >>> moneyfmt(d, curr='$')
   '-$1,234,567.89'
   >>> moneyfmt(d, places=0, sep='.', dp='', neg='', trailneg='-')
   '1.234.568-'
   >>> moneyfmt(d, curr='$', neg='(', trailneg=')')
   '($1,234,567.89)'
   >>> moneyfmt(Decimal(123456789), sep=' ')
   '123 456 789.00'
   >>> moneyfmt(Decimal('-0.02'), neg='<', trailneg='>')
   '<0.02>'

   """
   q = Decimal(10) ** -places      # 2 places --> '0.01'
   sign, digits, exp = value.quantize(q).as_tuple()
   result = []
   digits = map(str, digits)
   build, next = result.append, digits.pop
   if sign:
       build(trailneg)
   for i in range(places):
       build(next() if digits else '0')
   build(dp)
   if not digits:
       build('0')
   i = 0
   while digits:
       build(next())
       i += 1
       if i == 3 and digits:
           i = 0
           build(sep)
   build(curr)
   build(neg if sign else pos)
   return ''.join(reversed(result))

def pi():
```

```
   """Compute Pi to the current precision.

   >>> print pi()
   3.141592653589793238462643383

   """
   getcontext().prec += 2  # extra digits for intermediate steps
   three = Decimal(3)      # substitute "three=3.0" for regular floats
   lasts, t, s, n, na, d, da = 0, three, 3, 1, 0, 0, 24
   while s != lasts:
       lasts = s
       n, na = n+na, na+8
       d, da = d+da, da+32
       t = (t * n) / d
       s += t
   getcontext().prec -= 2
   return +s               # unary plus applies the new precision

def exp(x):
   """Return e raised to the power of x.  Result type matches input type.

   >>> print exp(Decimal(1))
   2.718281828459045235360287471
   >>> print exp(Decimal(2))
   7.389056098930650227230427461
   >>> print exp(2.0)
   7.38905609893
   >>> print exp(2+0j)
   (7.38905609893+0j)

   """
   getcontext().prec += 2
   i, lasts, s, fact, num = 0, 0, 1, 1, 1
   while s != lasts:
       lasts = s
       i += 1
       fact *= i
       num *= x
       s += num / fact
   getcontext().prec -= 2
   return +s

def cos(x):
   """Return the cosine of x as measured in radians.

   >>> print cos(Decimal('0.5'))
   0.8775825618903727161162815826
   >>> print cos(0.5)
   0.87758256189
   >>> print cos(0.5+0j)
   (0.87758256189+0j)

   """
   getcontext().prec += 2
   i, lasts, s, fact, num, sign = 0, 0, 1, 1, 1, 1
   while s != lasts:
       lasts = s
       i += 2
       fact *= i * (i-1)
       num *= x * x
       sign *= -1
       s += num / fact * sign
   getcontext().prec -= 2
   return +s

def sin(x):
   """Return the sine of x as measured in radians.

   >>> print sin(Decimal('0.5'))
```

```
0.47942553860420300027328793521
>>> print sin(0.5)
0.479425538604
>>> print sin(0.5+0j)
(0.479425538604+0j)

"""
getcontext().prec += 2
i, lasts, s, fact, num, sign = 1, 0, x, 1, x, 1
while s != lasts:
    lasts = s
    i += 2
    fact *= i * (i-1)
    num *= x * x
    sign *= -1
    s += num / fact * sign
getcontext().prec -= 2
return +s
```

10.4.8. Decimal FAQ¶

Q. It is cumbersome to type `decimal.Decimal('1234.5')`. Is there a way to minimize typing when using the interactive interpreter?

A. Some users abbreviate the constructor to just a single letter:

```
>>> D = decimal.Decimal
>>> D('1.23') + D('3.45')
Decimal('4.68')
```

Q. In a fixed-point application with two decimal places, some inputs have many places and need to be rounded. Others are not supposed to have excess digits and need to be validated. What methods should be used?

A. The `quantize()` method rounds to a fixed number of decimal places. If the `Inexact` trap is set, it is also useful for validation:

```
>>> TWOPLACES = Decimal(10) ** -2       # same as Decimal('0.01')

>>> # Round to two places
>>> Decimal('3.214').quantize(TWOPLACES)
Decimal('3.21')

>>> # Validate that a number does not exceed two places
>>> Decimal('3.21').quantize(TWOPLACES, context=Context(traps=[Inexact]))
Decimal('3.21')

>>> Decimal('3.214').quantize(TWOPLACES, context=Context(traps=[Inexact]))
Traceback (most recent call last):
  ...
Inexact: None
```

Q. Once I have valid two place inputs, how do I maintain that invariant throughout an application?

A. Some operations like addition, subtraction, and multiplication by an integer will automatically preserve fixed point. Others operations, like division and non-integer multiplication, will change the number of decimal places and need to be followed-up with a `quantize()` step:

```
>>> a = Decimal('102.72')           # Initial fixed-point values
>>> b = Decimal('3.17')
>>> a + b                           # Addition preserves fixed-point
Decimal('105.89')
>>> a - b
Decimal('99.55')
>>> a * 42                          # So does integer multiplication
Decimal('4314.24')
>>> (a * b).quantize(TWOPLACES)     # Must quantize non-integer multiplication
Decimal('325.62')
>>> (b / a).quantize(TWOPLACES)     # And quantize division
Decimal('0.03')
```

In developing fixed-point applications, it is convenient to define functions to handle the `quantize()` step:

```
>>> def mul(x, y, fp=TWOPLACES):
...     return (x * y).quantize(fp)
```

```
>>> def div(x, y, fp=TWOPLACES):
...     return (x / y).quantize(fp)

>>> mul(a, b)                       # Automatically preserve fixed-point
Decimal('325.62')
>>> div(b, a)
Decimal('0.03')
```

Q. There are many ways to express the same value. The numbers `200`, `200.000`, `2E2`, and `02E+4` all have the same value at various precisions. Is there a way to transform them to a single recognizable canonical value?

A. The `normalize()` method maps all equivalent values to a single representative:

```
>>> values = map(Decimal, '200 200.000 2E2 .02E+4'.split())
>>> [v.normalize() for v in values]
[Decimal('2E+2'), Decimal('2E+2'), Decimal('2E+2'), Decimal('2E+2')]
```

Q. Some decimal values always print with exponential notation. Is there a way to get a non-exponential representation?

A. For some values, exponential notation is the only way to express the number of significant places in the coefficient. For example, expressing `5.0E+3` as `5000` keeps the value constant but cannot show the original's two-place significance.

If an application does not care about tracking significance, it is easy to remove the exponent and trailing zeroes, losing significance, but keeping the value unchanged:

```
>>> def remove_exponent(d):
...     return d.quantize(Decimal(1)) if d == d.to_integral() else d.normalize()

>>> remove_exponent(Decimal('5E+3'))
Decimal('5000')
```

Q. Is there a way to convert a regular float to a `Decimal`?

A. Yes, all binary floating point numbers can be exactly expressed as a Decimal. An exact conversion may take more precision than intuition would suggest, so we trap `Inexact` to signal a need for more precision:

```
def float_to_decimal(f):
   "Convert a floating point number to a Decimal with no loss of information"
   n, d = f.as_integer_ratio()
   numerator, denominator = Decimal(n), Decimal(d)
   ctx = Context(prec=60)
   result = ctx.divide(numerator, denominator)
   while ctx.flags[Inexact]:
       ctx.flags[Inexact] = False
       ctx.prec *= 2
       result = ctx.divide(numerator, denominator)
   return result

>>> float_to_decimal(math.pi)
Decimal('3.141592653589793115997963468544185161590576171875')
```

Q. Why isn't the `float_to_decimal()` routine included in the module?

A. There is some question about whether it is advisable to mix binary and decimal floating point. Also, its use requires some care to avoid the representation issues associated with binary floating point:

```
>>> float_to_decimal(1.1)
Decimal('1.100000000000000088817841970012523233890533447265625')
```

Q. Within a complex calculation, how can I make sure that I haven't gotten a spurious result because of insufficient precision or rounding anomalies.

A. The decimal module makes it easy to test results. A best practice is to re-run calculations using greater precision and with various rounding modes. Widely differing results indicate insufficient precision, rounding mode issues, ill-conditioned inputs, or a numerically unstable algorithm.

Q. I noticed that context precision is applied to the results of operations but not to the inputs. Is there anything to watch out for when mixing values of different precisions?

A. Yes. The principle is that all values are considered to be exact and so is the arithmetic on those values. Only the results are rounded. The advantage for inputs is that "what you type is what you get". A disadvantage is that the results can look odd if you forget that the inputs haven't been rounded:

```
>>> getcontext().prec = 3
>>> Decimal('3.104') + Decimal('2.104')
```

```
Decimal('5.21')
>>> Decimal('3.104') + Decimal('0.000') + Decimal('2.104')
Decimal('5.20')
```

The solution is either to increase precision or to force rounding of inputs using the unary plus operation:

```
>>> getcontext().prec = 3
>>> +Decimal('1.23456789')      # unary plus triggers rounding
Decimal('1.23')
```

Alternatively, inputs can be rounded upon creation using the `Context.create_decimal()` method:

```
>>> Context(prec=5, rounding=ROUND_DOWN).create_decimal('1.2345678')
Decimal('1.2345')
```


Previous topic

10.3. `cmath` — Mathematical functions for complex numbers

Next topic

10.5. `fractions` — Rational numbers

This Page

- Show Source

Navigation

- index
- modules |
- next |
- previous |
- Python v2.6.4 documentation »
- The Python Standard Library »
- 10. Numeric and Mathematical Modules »