decimal — Fixed and Floating Point Math

Purpose: Decimal arithmetic using fixed and floating point numbers

The `decimal` module implements fixed and floating point arithmetic using the model familiar to most people, rather than the IEEE floating point version implemented by most computer hardware and familiar to programmers. A `Decimal` instance can represent any number exactly, round up or down, and apply a limit to the number of significant digits.

Decimal

Decimal values are represented as instances of the `Decimal` class. The constructor takes as argument one integer or string. Floating point numbers can be converted to a string before being used to create a `Decimal`, letting the caller explicitly deal with the number of digits for values that cannot be expressed exactly using hardware floating point representations. Alternately, the class method `from_float()` converts to the exact decimal representation.

```
# decimal_create.py

import decimal

fmt = '{0:<25} {1:<25}'

print(fmt.format('Input', 'Output'))
print(fmt.format('-' * 25, '-' * 25))

# Integer
print(fmt.format(5, decimal.Decimal(5)))

# String
print(fmt.format('3.14', decimal.Decimal('3.14')))

# Float
f = 0.1
print(fmt.format(repr(f), decimal.Decimal(str(f))))
print('{:<0.23g} {:<25}'.format(
    f,
    str(decimal.Decimal.from_float(f))[:25])
)
```

The floating point value of `0.1` is not represented as an exact value in binary, so the representation as a `float` is different from the `Decimal` value. The full string representation is truncated to 25 characters in the last line of this output.

```
$ python3 decimal_create.py

Input                     Output
------------------------- -------------------------
5                         5
3.14                      3.14
0.1                       0.1
0.10000000000000000555112 0.10000000000000000555111
```

`Decimals` can also be created from tuples containing a sign flag (`0` for positive, `1` for negative), a `tuple` of digits, and an integer exponent.

```
# decimal_tuple.py

import decimal

# Tuple
t = (1, (1, 1), -2)
print('Input  :', t)
print('Decimal:', decimal.Decimal(t))
```

The tuple-based representation is less convenient to create, but does offer a portable way of exporting decimal values without losing precision. The tuple form can be transmitted through the network or stored in a database that does not support accurate decimal values, then turned back into a `Decimal` instance later.

```
$ python3 decimal_tuple.py

Input  : (1, (1, 1), -2)
Decimal: -0.11
```

Formatting

`Decimal` responds to Python's string formatting protocol, using the same syntax and options as other numerical types.

```
# decimal_format.py

import decimal

d = decimal.Decimal(1.1)
print('Precision:')
print('{:.1}'.format(d))
print('{:.2}'.format(d))
print('{:.3}'.format(d))
print('{:.18}'.format(d))

print('\nWidth and precision combined:')
print('{:5.1f} {:5.1g}'.format(d, d))
print('{:5.2f} {:5.2g}'.format(d, d))
print('{:5.2f} {:5.2g}'.format(d, d))

print('\nZero padding:')
print('{:05.1}'.format(d))
print('{:05.2}'.format(d))
print('{:05.3}'.format(d))
```

The format strings can control the width of the output, the precision (number of significant digits), and how to pad the value to fill the width.

```
$ python3 decimal_format.py

Precision:
1
1.1
1.10
1.10000000000000009

Width and precision combined:
  1.1     1
 1.10   1.1
 1.10   1.1

Zero padding:
00001
001.1
01.10
```

Arithmetic

`Decimal` overloads the simple arithmetic operators so instances can be manipulated in much the same way as the built-in numeric types.

```
# decimal_operators.py

import decimal

a = decimal.Decimal('5.1')
b = decimal.Decimal('3.14')
c = 4
d = 3.14
```

```
print('a     =', repr(a))
print('b     =', repr(b))
print('c     =', repr(c))
print('d     =', repr(d))
print()

print('a + b =', a + b)
print('a - b =', a - b)
print('a * b =', a * b)
print('a / b =', a / b)
print()

print('a + c =', a + c)
print('a - c =', a - c)
print('a * c =', a * c)
print('a / c =', a / c)
print()

print('a + d =', end=' ')
try:
    print(a + d)
except TypeError as e:
    print(e)
```

`Decimal` operators also accept integer arguments, but floating point values must be converted to `Decimal` instances.

```
$ python3 decimal_operators.py

a     = Decimal('5.1')
b     = Decimal('3.14')
c     = 4
d     = 3.14

a + b = 8.24
a - b = 1.96
a * b = 16.014
a / b = 1.624203821656050955414012739

a + c = 9.1
a - c = 1.1
a * c = 20.4
a / c = 1.275

a + d = unsupported operand type(s) for +: 'decimal.Decimal' and
 'float'
```

Beyond basic arithmetic, `Decimal` includes the methods to find the base 10 and natural logarithms. The return values from `log10()` and `ln()` are `Decimal` instances, so they can be used directly in formulas with other values.

Special Values

In addition to the expected numerical values, `Decimal` can represent several special values, including positive and negative values for infinity, “not a number”, and zero.

```
# decimal_special.py

import decimal

for value in ['Infinity', 'NaN', '0']:
    print(decimal.Decimal(value), decimal.Decimal('-' + value))
print()

# Math with infinity
print('Infinity + 1:', (decimal.Decimal('Infinity') + 1))
print('-Infinity + 1:', (decimal.Decimal('-Infinity') + 1))

# Print comparing NaN
print(decimal.Decimal('NaN') == decimal.Decimal('Infinity'))
print(decimal.Decimal('NaN') != decimal.Decimal(1))
```

```
print(decimal.Decimal('NaN') != decimal.Decimal(1))
```

Adding to infinite values returns another infinite value. Comparing for equality with `NaN` always returns false and comparing for inequality always returns true. Comparing for sort order against `NaN` is undefined and results in an error.

```
$ python3 decimal_special.py

Infinity -Infinity
NaN -NaN
0 -0

Infinity + 1: Infinity
-Infinity + 1: -Infinity
False
True
```

Context

So far, all of the examples have used the default behaviors of the `decimal` module. It is possible to override settings such as the precision maintained, how rounding is performed, error handling, etc. by using a *context*. Contexts can be applied for all `Decimal` instances in a thread or locally within a small code region.

Current Context

To retrieve the current global context, use `getcontext`.

```
# decimal_getcontext.py
```

```
import decimal

context = decimal.getcontext()

print('Emax     =', context.Emax)
print('Emin     =', context.Emin)
print('capitals =', context.capitals)
print('prec     =', context.prec)
print('rounding =', context.rounding)
print('flags    =')
for f, v in context.flags.items():
    print('  {}: {}'.format(f, v))
print('traps    =')
for t, v in context.traps.items():
    print('  {}: {}'.format(t, v))
```

This example script shows the public properties of a `Context`.

```
$ python3 decimal_getcontext.py

Emax     = 999999
Emin     = -999999
capitals = 1
prec     = 28
rounding = ROUND_HALF_EVEN
flags    =
  <class 'decimal.InvalidOperation'>: False
  <class 'decimal.FloatOperation'>: False
  <class 'decimal.DivisionByZero'>: False
  <class 'decimal.Overflow'>: False
  <class 'decimal.Underflow'>: False
  <class 'decimal.Subnormal'>: False
  <class 'decimal.Inexact'>: False
  <class 'decimal.Rounded'>: False
  <class 'decimal.Clamped'>: False
traps    =
  <class 'decimal.InvalidOperation'>: True
  <class 'decimal.FloatOperation'>: False
  <class 'decimal.DivisionByZero'>: True
  <class 'decimal.Overflow'>: True
  <class 'decimal.Underflow'>: False
  <class 'decimal.Subnormal'>: False
```

```
  <class 'decimal.Subnormal'>: False
  <class 'decimal.Inexact'>: False
  <class 'decimal.Rounded'>: False
  <class 'decimal.Clamped'>: False
```

Precision

The `prec` attribute of the context controls the precision maintained for new values created as a result of arithmetic. Literal values are maintained as described.

```
# decimal_precision.py

import decimal

d = decimal.Decimal('0.123456')

for i in range(1, 5):
    decimal.getcontext().prec = i
    print(i, ':', d, d * 1)
```

To change the precision, assign a new value between 1 and `decimal.MAX_PREC` directly to the attribute.

```
$ python3 decimal_precision.py

1 : 0.123456 0.1
2 : 0.123456 0.12
3 : 0.123456 0.123
4 : 0.123456 0.1235
```

Rounding

There are several options for rounding to keep values within the desired precision.

`ROUND_CEILING`
: Always round upwards towards infinity.

`ROUND_DOWN`
: Always round toward zero.

`ROUND_FLOOR`
: Always round down towards negative infinity.

`ROUND_HALF_DOWN`
: Rounds away from zero if the last significant digit is greater than or equal to 5, otherwise toward zero.

`ROUND_HALF_EVEN`
: Like `ROUND_HALF_DOWN` except that if the value is 5 then the preceding digit is examined. Even values cause the result to be rounded down and odd digits cause the result to be rounded up.

`ROUND_HALF_UP`
: Like `ROUND_HALF_DOWN` except if the last significant digit is 5 the value is rounded away from zero.

`ROUND_UP`
: Round away from zero.

`ROUND_05UP`
: Round away from zero if the last digit is `0` or 5, otherwise towards zero.

```
# decimal_rounding.py

import decimal

context = decimal.getcontext()

ROUNDING_MODES = [
    'ROUND_CEILING',
    'ROUND_DOWN',
    'ROUND_FLOOR',
    'ROUND_HALF_DOWN',
    'ROUND_HALF_EVEN',
    'ROUND_HALF_UP',
    'ROUND_UP',
    'ROUND_05UP',
]

header_fmt = '{:10} ' + ' '.join(['{:^8}'] * 6)
```

```
print(header_fmt.format(
    ' ',
    '1/8 (1)', '-1/8 (1)',
    '1/8 (2)', '-1/8 (2)',
    '1/8 (3)', '-1/8 (3)',
))
for rounding_mode in ROUNDING_MODES:
    print('{0:10}'.format(rounding_mode.partition('_')[-1]),
          end=' ')
    for precision in [1, 2, 3]:
        context.prec = precision
        context.rounding = getattr(decimal, rounding_mode)
        value = decimal.Decimal(1) / decimal.Decimal(8)
        print('{0:^8}'.format(value), end=' ')
        value = decimal.Decimal(-1) / decimal.Decimal(8)
        print('{0:^8}'.format(value), end=' ')
    print()
```

This program shows the effect of rounding the same value to different levels of precision using the different algorithms.

```
$ python3 decimal_rounding.py

           1/8 (1)  -1/8 (1) 1/8 (2)  -1/8 (2) 1/8 (3)  -1/8 (3)
CEILING      0.2      -0.1     0.13    -0.12    0.125    -0.125

DOWN         0.1      -0.1     0.12    -0.12    0.125    -0.125

FLOOR        0.1      -0.2     0.12    -0.13    0.125    -0.125

HALF_DOWN    0.1      -0.1     0.12    -0.12    0.125    -0.125

HALF_EVEN    0.1      -0.1     0.12    -0.12    0.125    -0.125

HALF_UP      0.1      -0.1     0.13    -0.13    0.125    -0.125

UP           0.2      -0.2     0.13    -0.13    0.125    -0.125

05UP         0.1      -0.1     0.12    -0.12    0.125    -0.125
```

Local Context

The context can be applied to a block of code using the `with` statement.

```
# decimal_context_manager.py

import decimal

with decimal.localcontext() as c:
    c.prec = 2
    print('Local precision:', c.prec)
    print('3.14 / 3 =', (decimal.Decimal('3.14') / 3))

print()
print('Default precision:', decimal.getcontext().prec)
print('3.14 / 3 =', (decimal.Decimal('3.14') / 3))
```

The `Context` supports the context manager API used by `with`, so the settings only apply within the block.

```
$ python3 decimal_context_manager.py

Local precision: 2
3.14 / 3 = 1.0

Default precision: 28
3.14 / 3 = 1.046666666666666666666666667
```

Per-Instance Context

Contexts also can be used to construct `Decimal` instances, which then inherit the precision and rounding arguments to the conversion from the context.

```
# decimal_instance_context.py

import decimal

# Set up a context with limited precision
c = decimal.getcontext().copy()
c.prec = 3

# Create our constant
pi = c.create_decimal('3.1415')

# The constant value is rounded off
print('PI    :', pi)

# The result of using the constant uses the global context
print('RESULT:', decimal.Decimal('2.01') * pi)
```

This lets an application select the precision of constant values separately from the precision of user data, for example.

```
$ python3 decimal_instance_context.py

PI    : 3.14
RESULT: 6.3114
```

Threads

The “global” context is actually thread-local, so each thread can potentially be configured using different values.

```
# decimal_thread_context.py

import decimal
import threading
from queue import PriorityQueue


class Multiplier(threading.Thread):
    def __init__(self, a, b, prec, q):
        self.a = a
        self.b = b
        self.prec = prec
        self.q = q
        threading.Thread.__init__(self)

    def run(self):
        c = decimal.getcontext().copy()
        c.prec = self.prec
        decimal.setcontext(c)
        self.q.put((self.prec, a * b))


a = decimal.Decimal('3.14')
b = decimal.Decimal('1.234')
# A PriorityQueue will return values sorted by precision,
# no matter what order the threads finish.
q = PriorityQueue()
threads = [Multiplier(a, b, i, q) for i in range(1, 6)]
for t in threads:
    t.start()

for t in threads:
    t.join()

for i in range(5):
    prec, value = q.get()
    print('{}  {}'.format(prec, value))
```

This example creates a new context using the specified, then installs it within each thread.

```
$ python3 decimal_thread_context.py

1  4
2  3.9
3  3.87
4  3.875
5  3.8748
```

See also

- Standard library documentation for decimal
- Python 2 to 3 porting notes for decimal
- Wikipedia: Floating Point – Article on floating point representations and arithmetic.
- Floating Point Arithmetic: Issues and Limitations – Article from the Python tutorial describing floating point math representation issues.

Mathematics

fractions — Rational Numbers

Quick Links

Decimal
Formatting
Arithmetic
Special Values
Context
Current Context
Precision
Rounding
Local Context
Per-Instance Context
Threads

This page was last updated 2016-12-29.

Navigation

Mathematics
fractions — Rational Numbers

The output from all the example programs from PyMOTW-3 has been generated with Python 3.7.1, unless otherwise noted. Some of the features described here may not be available in earlier versions of Python.

Looking for examples for Python 2?

This Site

Module Index
Index

Other Writing

- Blog
- The Python Standard Library By Example