Python Decimal

Summary: in this tutorial, you'll learn about the Python `decimal` module that supports fast correctly-rounded decimal floating-point arithmetic.

Introduction to the Python decimal module

Many decimal numbers don't have exact representations in binary floating-point (https://www.pythontutorial.net/advanced-python/python-float/) such as 0.1. When using these numbers in arithmetic operations, you'll get a result that you would not expect. For example:

```
x = 0.1
y = 0.1
z = 0.1

s = x + y + z

print(s)
```

Output:

```
0.30000000000000004
```

The result is 0.30000000000000004, not 0.3.

To solve this problem, you use the `Decimal` class from the `decimal` module as follows:

```
import decimal
from decimal import Decimal


x = Decimal('0.1')
y = Decimal('0.1')
z = Decimal('0.1')

s = x + y + z

print(s)
```

Output:

```
0.3
```

The output is as expected.

The Python `decimal` module supports arithmetic that works the same as the arithmetic you learn at school.

Unlike floats (https://www.pythontutorial.net/advanced-python/python-float/) , Python represents decimal numbers exactly. And the exactness carries over into arithmetic. For example, the following expression returns exactly 0.0:

```
Decimal('0.1') + Decimal('0.1') + Decimal('0.1') - Decimal('0.3')
```

Decimal context

Decimal always associates with a context (https://www.pythontutorial.net/advanced-python/python-context-managers/) that controls the following aspects:

- Precision during an arithmetic operation
- Rounding algorithm

By default, the context is global. The global context is the default context. Also, you can set a temporary context that will take effect locally without affecting the global context.

To get the default context, you call the `getcontext()` function from the `decimal` module:

```
decimal.getcontext()
```

The `getcontext()` function returns the default context, which can be global or local.

To create a new context copied from another context, you use the `localcontext()` function:

```
decimal.localcontext(ctx=None)
```

The `localcontext()` returns a new context copied from the context `ctx` if specified.

Once getting the context object, you can access the precision and rouding via the `prec` and `rounding` property respectively:

- `ctx.pre` : get or set the precision. The ctx.pre is an integer which defaults to 28
- `ctx.rounding` : get or set the rounding mechanism. The rounding is a string. It defaults to `'ROUND_HALF_EVEN'` . Note floats also use this rounding mechanism.

Python provides the following rounding mechanisms:

Rounding	**Description**
ROUND_UP	round away from zero

Rounding	Description
ROUND_DOWN	round towards zero
ROUND_CEILING	round to ceiling (towards positive infinity)
ROUND_FLOOR	round to floor (towards negative infinity)
ROUND_HALF_UP	round to nearest, ties away from zero
ROUND_HALF_DOWN	round to nearest, ties towards zero
ROUND_HALF_EVEN	round to nearest, ties to even (least significant digit)

This example illustrates how to get the default precision and rounding of the default context:

```
import decimal

ctx = decimal.getcontext()

print(ctx.prec)
print(ctx.rounding)
```

Output:

```
28
ROUND_HALF_EVEN
```

The following example shows how the `'ROUND_HALF_EVEN'` rounding mechanism takes effect:

```
import decimal
from decimal import Decimal
```

```
x = Decimal('2.25')
y = Decimal('3.35')

print(round(x, 1))
print(round(y, 1))
```

Output:

```
2.2
3.4
```

If you change the rounding to `'ROUND_HALF_UP'`, you'll get a different result:

```
import decimal
from decimal import Decimal


ctx = decimal.getcontext()
ctx.rounding = decimal.ROUND_HALF_UP

x = Decimal('2.25')
y = Decimal('3.35')

print(round(x, 1))
print(round(y, 1))
```

Output:

```
2.3
3.4
```

The following example shows you how to copy the default context and change the rounding to `'ROUND_HALF_UP'`:

```
import decimal
from decimal import Decimal


x = Decimal('2.25')
y = Decimal('3.35')

with decimal.localcontext() as ctx:
    print('Local context:')
    ctx.rounding = decimal.ROUND_HALF_UP
    print(round(x, 1))
    print(round(y, 1))

print('Global context:')
print(round(x, 1))
print(round(y, 1))
```

Output:

```
Local context:
2.3
3.4
Global context:
2.2
3.4
```

Notice that the local context doesn't affect the global context. After the with block, Python uses the default rounding mechanism.

Decimal constructor

The `Decimal` constructor allows you to create a new `Decimal` object based on a value:

```
Decimal(value='0', context=None)
```

The `value` argument can be an integer, string, tuple, float, or another Decimal object. If you don't provide the value argument, it defaults to `'0'`.

If the 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:

```
(sign, (digit1,digit2, digit3,...), exponent)
```

For example:

```
3.14 = 314 x 10^-2
```

The tuple has three elements as follows:

- sign is 0
- digits is (3,1,4)
- exponent is -2

Therefore, you'll need to pass the following tuple to the `Decimal` constructor:

```
import decimal
from decimal import Decimal

x = Decimal((0, (3, 1, 4), -2))
print(x)
```

Output:

```
3.14
```

Notice that the decimal context precision only affects the arithmetic operation, not the `Decimal` constructor. For example:

```
import decimal
from decimal import Decimal



decimal.getcontext().prec = 2

pi = Decimal('3.14159')
radius = 1

print(pi)

area = pi * radius * radius
print(area)
```

When you use a float that doesn't have an exact binary float representation, the `Decimal` constructor cannot create an accurate decimal representation. For example:

```
import decimal
from decimal import Decimal

x = Decimal(0.1)
print(x)
```

Output:

```
0.1000000000000000055511151231257827021181583404541015625
```

In practice, you'll use a string or a tuple to construct a `Decimal` .

Decimal arithmetic operations

Some arithmetic operators don't work the same as floats or integers (https://www.pythontutorial.net/advanced-python/python-integers/) , such as div (https://www.pythontutorial.net/advanced-python/python-floor-division/) (`//`) and mod (`%)` .

For decimal numbers, the `//` operator performs a truncated division:

```
x // y = trunc( x / y)
```

The `Decimal` class provides some mathematical operations such as `sqrt` and `log` . However, it doesn't have all the functions defined in the `math` module.

When you use functions from the `math` module for decimal numbers, Python will cast the `Decimal` objects to floats before carrying arithmetic operations. This results in losing the precision built in the decimal objects.

Summary

- Use the Python `decimal` module when you want to support fast correctly-rounded decimal floating-point arithmetic.
- Use the `Decimal` class from the `decimal` module to create Decimal object from strings, integers, and tuples.
- The `Decimal` numbers have a context that controls the precision and rounding mechanism.
- The `Decimal` class doesn't have all methods defined in the `math` module. However, you should use the Decimal's arithmetic methods if they're available.