Decimal

Austin Bingham
COFOUNDER - SIXTY NORTH

@austin_bingham

Robert Smallshire
COFOUNDER - SIXTY NORTH

@robsmallshire

Overview

`decimal.Decimal`

Represent decimal values exactly

Relationship between `Decimal` and other types

The pitfalls of mixing numeric types

Preservation of precision

Control calculations with context

Special values

Surprising behavior of certain operators

Working with Decimal Values

`decimal.Decimal`

Fast, correctly-rounded number type for base-10 arithmetic

Floating-point type with finite, configurable precision

Useful for domains like finance

decimal.Decimal

```
>>> import decimal
>>> decimal.getcontext()
Context(prec=28, rounding=ROUND_HALF_EVEN, Emin=-999999, Emax=999999, capitals=1
, clamp=0, flags=[], traps=[InvalidOperation, DivisionByZero, Overflow])
>>> decimal.Decimal(5)
Decimal('5')
>>> from decimal import Decimal
>>> Decimal(7)
Decimal('7')
>>> Decimal('0.8')
Decimal('0.8')
>>> Decimal('0.8') - Decimal('0.7')
Decimal('0.1')
>>>
```

Construction with Fractional Values

```
>>> Decimal(0.8) - Decimal(0.7)
Decimal('0.10000000000000008881784197001')
>>> type(0.8)
<class 'float'>
>>> type(0.7)
<class 'float'>
>>> decimal.getcontext().traps[decimal.FloatOperation] = True
>>> Decimal(0.8)
Traceback (most recent call last):
  File "<stdin>", line 1, in <module>
decimal.FloatOperation: [<class 'decimal.FloatOperation'>]
>>> Decimal('0.8') > 0.7
Traceback (most recent call last):
  File "<stdin>", line 1, in <module>
decimal.FloatOperation: [<class 'decimal.FloatOperation'>]
>>>
```

Specify fractional `Decimal` literal arguments as strings to avoid intermediate `float`s.

Preserving Decimal Precision

```
>>> a = Decimal(3)
>>> b = Decimal('3.0')
>>> c = Decimal('3.00')
>>> a
Decimal('3')
>>> b
Decimal('3.0')
>>> c
Decimal('3.00')
>>> a * 2
Decimal('6')
>>> b * 2
Decimal('6.0')
>>> c * 2
Decimal('6.00')
>>> decimal.getcontext().prec = 6
>>> d = Decimal('1.234567')
>>> d
Decimal('1.234567')
>>> d + Decimal(1)
Decimal('2.23457')
>>>
```

Special Values

```
>>> Decimal('Infinity')
Decimal('Infinity')
>>> Decimal('-Infinity')
Decimal('-Infinity')
>>> Decimal('NaN')
Decimal('NaN')
>>> Decimal('NaN') + Decimal('1.414')
Decimal('NaN')
>>>
```

`Decimal`s can be combined safely with Python integers

This is not generally true for `float` and other numeric types

Decimal and float

```
>>> Decimal('1.4') + 0.6
Traceback (most recent call last):
  File "<stdin>", line 1, in <module>
TypeError: unsupported operand type(s) for +: 'decimal.Decimal' and 'float'
>>> Decimal('1.4') > 0.6
True
>>>
```

`Decimal` and Python Code

1. **`Decimal` plays well with most of Python**
2. **Code for other number types will generally work for `Decimal`**
3. **But there are a few differences to be aware of**

The result of modulus with `Decimal` takes its sign from the first operand.

Modulus with `int`

```
>>> (-7) % 3
2
>>>
```

Modulus with `int`

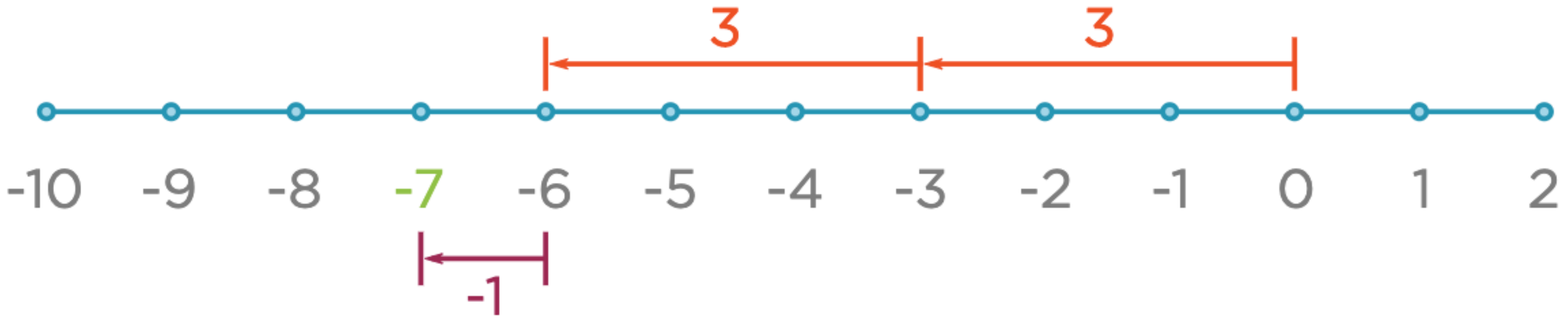
-7 % 3 == 2

For **`int`**, the modulus has the same sign as the divisor

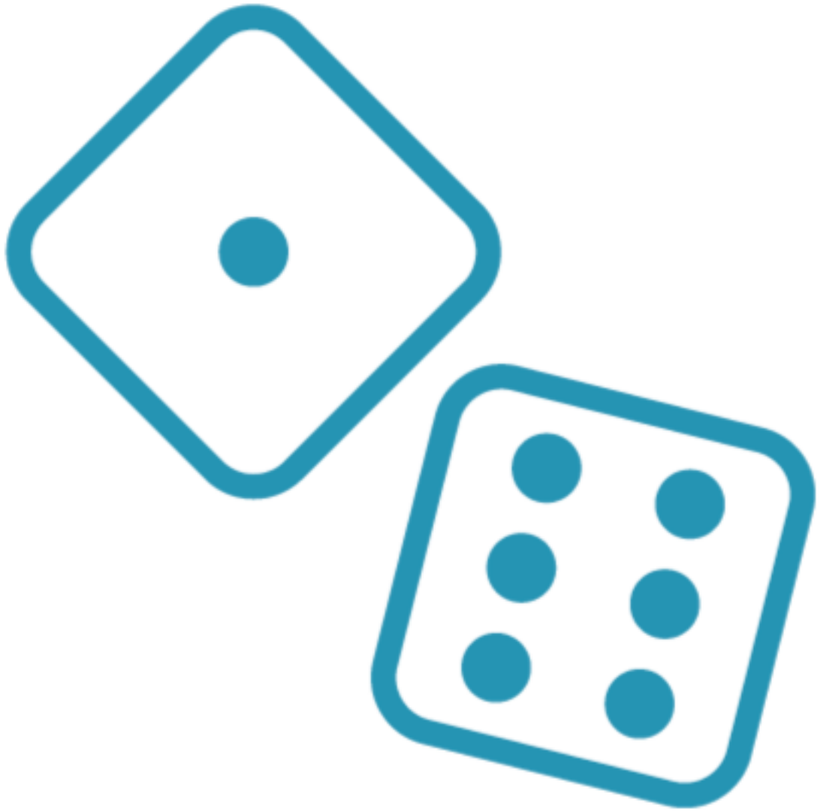
Modulus with `Decimal`

```
>>> from decimal import Decimal
>>> Decimal(-7) % Decimal(3)
Decimal('-1')
>>>
```

Modulus with `Decimal`




```
Decimal(-7) % Decimal(3) == Decimal(-1)
```



This difference may seem capricious

Retains `float` compatibility with legacy Python versions

`Decimal` implements the IEEE 854 floating-point standard

Broken Expectations

```
>>> is_odd(2.0)
False
>>> is_odd(3.0)
True
>>> is_odd(-2.0)
False
>>> is_odd(-3.0)
True
>>> is_odd(Decimal(2))
False
>>> is_odd(Decimal(3))
True
>>> is_odd(Decimal(-2))
False
>>> is_odd(Decimal(-3))
False
>>> Decimal(-3) % 2
Decimal('-1')
>>> def is_odd(n):
...     return n % 2 != 0
...
>>> is_odd(Decimal(-3))
True
>>>
```

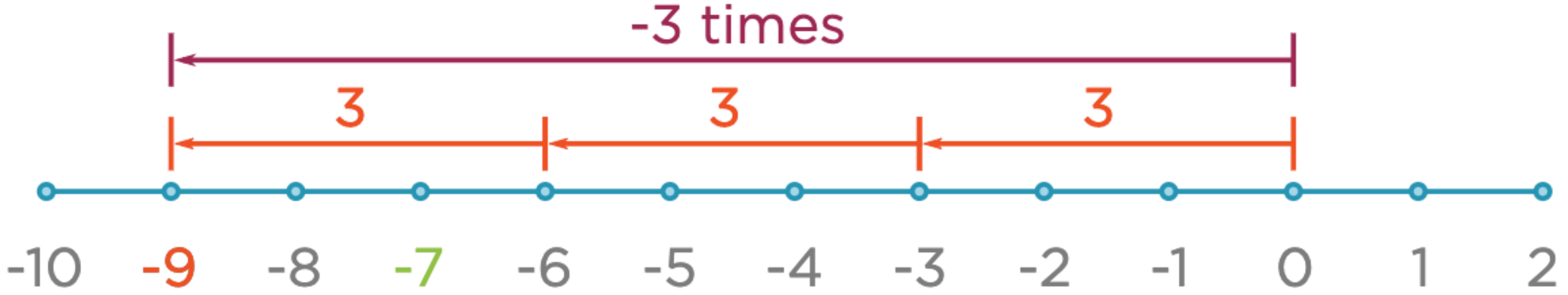
This Identity Is Preserved

$$x == (x // y) * y + x \% y$$

Floor Division for `int`

```
>>> -7 // 3
-3
>>>
```

Floor Division with `int`

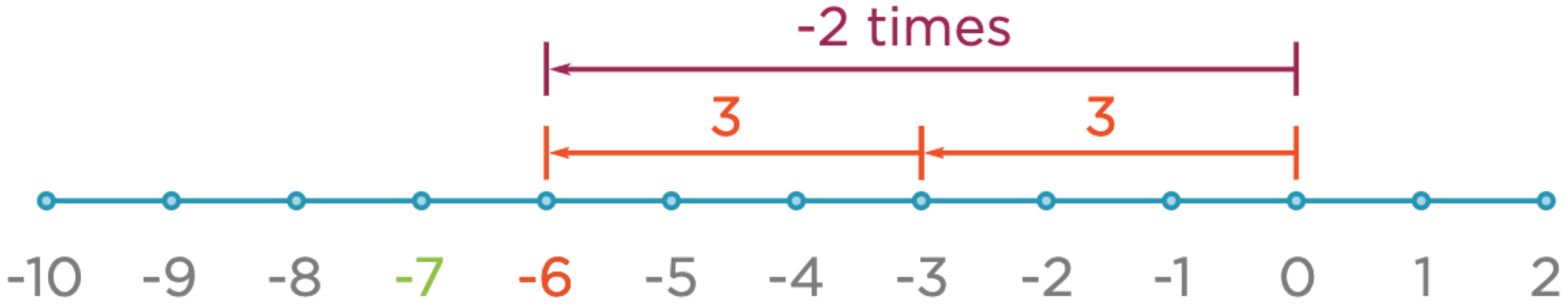


$$-7 \mathbin{//} 3 == -3$$

Floor Division for `Decimal`

```
>>> Decimal(-7) // Decimal(3)
Decimal('-2')
>>>
```

Floor Division with `Decimal`




```
Decimal(-7) // Decimal(3) == Decimal(-2)
```

"Floor division" is confusingly named

`Decimal` truncates towards zero, not "down"

The semantics of `//` are type-dependent

Math Functions

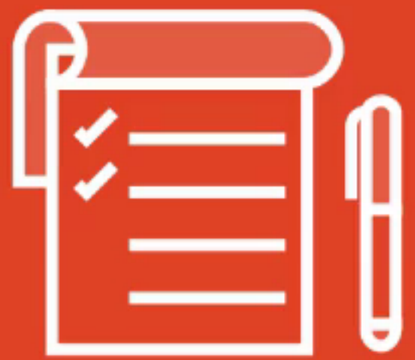
Math Functions

```
>>> Decimal('0.81').sqrt()
Decimal('0.9')
>>>
```

Recap of Number Types

`float`	`Decimal`	Incomplete	New types
Cannot exactly represent some decimal values such as 0.7	Can exactly represent decimal values such as 0.7	Neither can represent some numbers such as two-thirds	We need new number types for more complete coverage of the number line

Summary



`Decimal` uses a base-10 floating-point number representation

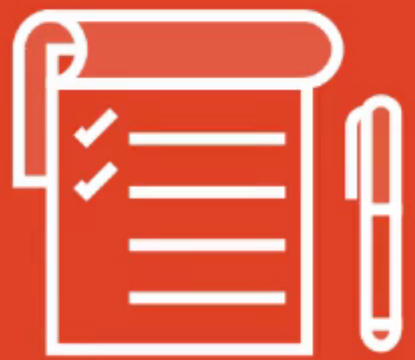
`Decimal` can exactly represent decimal values

`Decimal` can be safely constructed with string and integers

Constructing `Decimal` from `float` can lead to loss of data

The `decimal` module can be configured to disallow operations with `float`

Summary



`Decimal` preserves precision across calculations

The `decimal` context can limit the precision of computations

`Decimal` supports the special "infinity" and "not-a-number" values

Some operators like modulus behave differently for `Decimal`

Functions in `math` don't work with `Decimal`