DECIMALS

The `decimal` module (PEP 327)

`float 0.1` → infinite binary expansion

$$(0.1)_{10} = (0.0\,0011\,0011\,0011\;\ldots)_2$$
$$= \frac{1}{16} + \frac{1}{32} + \frac{1}{256} + \frac{1}{512} + \frac{1}{4096} + \frac{1}{8192} + \;\ldots$$

→ finite decimal expansion

$$(0.1)_{10} = \frac{1}{10}$$

alternative to using the (binary) `float` type → avoids the approximation issues with floats

finite number of significant digits → rational number (see videos on rationals)

So why not just use the `Fraction` class?

to add two fractions → common denominator

→ complex, requires extra memory

Why do we even care? Why not just use binary floats?

finance, banking, and any other field where exact finite representations are highly desirable

let's say we are adding up all the financial transactions that took place over a certain time period

amount = `$100.01`   `1,000,000,000` transactions   NYSE: 2-6 billion shares traded **daily**

`100.01` → `100.01000000000000511590769750`

sum → `$100010000000.00` (exact decimal)

`$100009998761.14639282226562500000000` (approximate binary float)

`$1238.85`… off!!

Decimals have a context that controls certain aspects of working with decimals

precision during arithmetic operations

rounding algorithm

This context can be global → the default context

or temporary (local) → sets temporary settings without affecting the global settings

```
import decimal
```

default context → `decimal.getcontext()`

local context → `decimal.localcontext(ctx=None)`

creates a new context, copied from ctx
or from default if ctx not specified

returns a context manager (use a `with` statement)

Precision and Rounding

`ctx = decimal.getcontext()` → context (global in this case)

`ctx.prec` → get or set the precision (value is an int)

`ctx.rounding` → get or set the rounding mechanism (value is a string)

ROUND_UP	rounds away from zero
ROUND_DOWN	rounds towards zero
ROUND_CEILING	rounds to ceiling (towards $+\infty$)
ROUND_FLOOR	rounds to floor (towards $-\infty$)
ROUND_HALF_UP	rounds to nearest, ties away from zero
ROUND_HALF_DOWN	rounds to nearest, ties towards zero
ROUND_HALF_EVEN	rounds to nearest, ties to even (least significant digit)

float rounding algorithm

Working with Global and Local Contexts

Global

```
decimal.getcontext().rounding = decimal.ROUND_HALF_UP
//decimal operations performed here will use the current default context
```

Local

```
with decimal.localcontext() as ctx:
    ctx.prec = 2
    ctx.rounding = decimal.ROUND_HALF_UP

    //decimal operations performed here
    //will use the ctx context
```

Code