**Important!**

<table>
<thead>
<tr>
<th>YES</th>
<th>NO</th>
</tr>
</thead>
<tbody>
<tr>
<td><strong>class</strong> Point3(object):</td>
<td><strong>class</strong> Point3:</td>
</tr>
</tbody>
</table>
| """Instances are 3D points Attributes: x: x-coord [float] y: y-coord [float] z: z-coord [float]""" | """Instances are 3D points Attributes: x: x-coord [float] y: y-coord [float] z: z-coord [float]"""
| 3.0-Style Classes Well-Designed | "Old-Style" Classes Very, Very Bad |

**Problem: Doing Math is Unwieldy**

<table>
<thead>
<tr>
<th>What We Want</th>
<th>What We Get</th>
</tr>
</thead>
</table>
| \[
\frac{1}{2} + \frac{1}{3} + \frac{1}{4} \times \frac{5}{4}
\] | \[
(p = \text{Fraction}(1,2))
\]
| \[
\frac{1}{2} + \frac{1}{3} + \frac{1}{4} \times \frac{5}{4}
\] | \[
\frac{1}{2} + \frac{1}{3} + \frac{1}{4} \times \frac{5}{4}
\]

Why not use the standard Python math operations?

This is confusing!

**Case Study: Fractions**

- Want to add a new type
  - Values are fractions: \(\frac{1}{2}, \frac{3}{4}\)
  - Operations are standard multiply, divide, etc.
- **Example:** \(\frac{1}{2} \times \frac{3}{4} = \frac{3}{8}\)
- Can do this with a class
  - Values are fraction objects
  - Operations are methods
- **Example:** simplefrac.py

**Example: Converting Values to Strings**

- **str() Function**
  - Usage: `str(<expression>)`
    - Evaluates the expression
    - Converts it into a string
  - How does it convert?
    - `str(2)` → `'2'`
    - `str(True)` → `'True'`
    - `str(-1)` → `'True'`
    - `str(Point3())` → `(0.0,0.0,0.0)`

- **Backquotes**
  - Usage: `'<expression>'`
    - Evaluates the expression
    - Converts it into a string
    - How does it convert?
      - `2` → `'2'`
      - `True` → `'True'`
      - `True` → `'True'`
      - `Point3()` → `<class Point3>(0.0,0.0,0.0)`

**What Does str() Do On Objects?**

- Does NOT display contents
  - `p = Point3(1,2,3)`
  - `str(p)`
    - `<Point3 object at 0x1007a90>`
- Must add a special method
  * `__str__(self)` for `str()`
  * `__repr__(self)` for backquotes
- Could get away with just one
  - Backquotes require `__repr__`
  - `str()` can use `__repr__`
    - (if `__str__` is not there)

**Special Methods in Python**

- Have seen three so far
  - `__init__` for initializer
  - `__str__` for `str()`
  - `__repr__` for backquotes
- Start/end with 2 underscores
  - This is standard in Python
  - Used in all special methods
    - Also for special attributes
    - For a complete list, see http://docs.python.org/reference/datamodel.html
### Returning to Fractions

#### What We Want

\[
\left( \frac{1}{2} + \frac{1}{3} + \frac{1}{4} \right) \times \frac{5}{4}
\]

Why not use the standard Python math operations?

#### Operator Overloading

- Python has methods that correspond to built-in ops
- \( \text{\_\_add\_\_} \) corresponds to +
- \( \text{\_\_mul\_\_} \) corresponds to *
- Not implemented by default
- Implementing one allows you to use that op on your objects
- Called operator overloading
- Changes operator meaning

---

### Operator Overloading: Addition

#### class Fraction(object):

```python
'''Instance attributes:
    numerator: top [int]
denominator: bottom [int > 0]'''

def \_\_add\_\_(self, q):
    '''Returns: Sum of self, q
    Makes a new Fraction
    Precondition: q a Fraction'''
    assert type(q) == Fraction,
    top = self.numerator * q.denominator
    bot = self.denominator * q.numerator
    return Fraction(top, bot)
```

```python
>>> p = Fraction(1,2)
>>> q = Fraction(3,4)
>>> r = p + q
```

Python converts to

Operator overloading uses method in object on left.

---

### Issues With Overloading ==

- Overloading == does not also overload comparison !=
  - Must implement \_\_ne__
  - Why? Will see later
  - But (not x == y) is okay!
- What if you still want to compare Folder names?
  - Use \_\_eq__ on variables
  - (x is y) True if x, y contain the same folder name
  - Check if variable is empty; \( x \text{ is } \text{None} \) \( \Rightarrow \) \( x \text{ is } \text{None} \)

---

### Operator Overloading: Multiplication

#### class Fraction(object):

```python
'''Instance attributes:
    numerator: top [int]
denominator: bottom [int > 0]'''

def \_\_mul\_\_(self, q):
    '''Returns: Product of self, q
    Makes a new Fraction
    Precondition: q a Fraction'''
    assert type(q) == Fraction,
    top = self.numerator * q.numerator
    bot = self.denominator * q.denominator
    return Fraction(top, bot)
```

```python
>>> p = Fraction(1,2)
>>> q = Fraction(3,4)
>>> r = p * q
```

Python converts to

Operator overloading uses method in object on left.

---

### Comparing Objects for Equality

- Earlier in course, we saw == compare object contents
  - This is not the default
  - Default: folder names
- Must implement \_\_eq__
  - Operator overloading!
  - Not limited to simple attribute comparison
  - Ex: cross multiplying

#### class Fraction(object):

```python
'''Instance attributes:
    numerator: top [int]
denominator: bottom [int > 0]'''

def \_\_eq\_\_(self, q):
    '''Returns: True if self, q equal,
    False if not, or q not a Fraction'''
    if type(q) != Fraction:
        return False
    left = self.numerator\*q.denominator
    right = self.denominator\*q.numerator
    return left == right
```

```python
>>> p = Fraction(1,2)
>>> q = Fraction(3,4)
>>> p == q
```

---

### is Versus ==

- \( p \text{ is } q \text{ evaluates to False} \)
  - Compares folder names
  - Cannot change this
- \( p \text{ == } q \text{ evaluates to True} \)
  - But only because method \_\_eq__ compares contents

Always use \( x \text{ is None} \) not \( x == \text{None} \)