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:** `frac1.py`

```
class Fraction(object):
    '''Instance is a fraction n/d'''
    INSTANCE ATTRIBUTES:
    _numerator: top [int]
    _denominator: bottom [int > 0]

    __init__(self, n=0, d=1):
        '''Init: makes a Fraction'''
        self._numerator = n
        self._denominator = d
```

Problem: Doing Math is Unwieldy

<table>
<thead>
<tr>
<th>What We Want</th>
<th>What We Get</th>
</tr>
</thead>
<tbody>
<tr>
<td>( \frac{1}{2} + \frac{1}{3} + \frac{1}{4} ) * ( \frac{5}{4} )</td>
<td>( \frac{1}{2} \times \frac{3}{4} = \frac{3}{8} )</td>
</tr>
</tbody>
</table>

Why not use the standard Python math operations?

This is confusing!

Operator Overloading

- Many operators in Python use special symbols
  - `+`, `-`, `/`, `*`, `**` for mathematics
  - `==`, `!=`, `<`, `>` for comparisons
- The meaning of these symbols depends on type
  - `1 + 2` vs `'Hello' + 'World'`
  - `1 < 2` vs `'Hello' < 'World'`
- Our new type might want to use these symbols
  - We **overload** them to support our new type

Returning to Fractions

```
class Fraction(object):
    '''Instance attributes:
    _numerator: top [int]
    _denominator: bottom [int > 0]'''

    __mul__(self, q):
        '''Returns Product of self, q
        Makes a new Fraction; does not modify contents of self or q
        Precondition: q a Fraction'''
        assert type(q) == Fraction
        top = self._numerator * q._numerator
        bot = self._denominator * q._denominator
        return Fraction(top, bot)
```

```
class Fraction(object):
    '''Instance attributes:
    _numerator: top [int]
    _denominator: bottom [int > 0]'''

    __add__(self, q):
        '''Returns Sum of self, q
        Makes a new Fraction
        Precondition: q a Fraction'''
        assert type(q) == Fraction
        bot = self._denominator * q._denominator
        top = self._numerator * q._denominator + self._denominator * q._numerator
        return Fraction(top, bot)
```

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

Python converts to

```
def __mul__(self, q):
    '''Returns Product of self, q
    Makes a new Fraction; does not modify contents of self or q
    Precondition: q a Fraction'''
    assert type(q) == Fraction
    top = self._numerator * q._numerator
    bot = self._denominator * q._denominator
    return Fraction(top, bot)
```

```
def __add__(self, q):
    '''Returns Sum of self, q
    Makes a new Fraction
    Precondition: q a Fraction'''
    assert type(q) == Fraction
    return Fraction(top, bot)
```

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

Operator overloading uses method in object on left.
```

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

Operator overloading uses method in object on left.
```

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

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

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

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

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

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

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

```
>>> p = Fraction(1,2)
>>> q = 2 # an int
>>> r = p*q
```

* `p` \(\sim q\) evaluates to False
* Compares folder names
* Cannot change this

```
>>> p == q
False
```

#### is Versus ==

- `p == q` evaluates to True
- But only because method `__eq__` compares contents

```
>>> p == q
False
```

#### Solution: Look at Argument Type

- Overloading use `left` type
  - `p*q` \(\sim p\_\_mul\_\_\_\_(q)\)
  - Done for us automatically
  - Looks in class definition
- What about type on `right`?
  - Have to handle ourselves
  - Can implement with ifs

```
>>> p = Fraction(1,2)
>>> q = 2 # an int
>>> r = p*q
```

#### Advanced Example: A6 Pixels

- Image is list of list of RGB
  - But this is really slow
  - Faster: byte buffer (???)
  - Beyond scope of course
- **Compromise:** Pixels class
  - Has byte buffer attribute
  - Pretends to be list of tuples
  - You can slice/iterate/etc...
  - Uses data model to do this

```
[(255,255,255), (255,255,255), ...]
```