**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`

```python
class Fraction(object):
    ""
    Instance is a fraction n/d
    ""
    INSTANCE ATTRIBUTES:
    _numerator: top [int]
    _denominator: bottom [int > 0]
    ""
    def __init__(self, n=0, d=1):
        ""
        Init: makes a Fraction"
        self._numerator = n
        self._denominator = d
```

**Problem: Doing Math is Unwieldy**

**What We Want**

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

**What We Get**

```python
>>> p = Fraction(1,2)
>>> q = Fraction(1,3)
>>> r = Fraction(1,4)
>>> s = Fraction(5,4)
>>> (p.add(q.add(r))).mult(s)
```

This is confusing!

**Operator Overloading**

- Many operators in Python are 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**

- Python has methods that correspond to built-in ops
  - `__add__` corresponds to `+`
  - `__mul__` corresponds to `*`
  - `__eq__` corresponds to `==`
- Not implemented by default
- To overload operators you implement these methods

**Operator Overloading: Multiplication**

```python
class Fraction(object):
    ""
    Instance attributes:
    _numerator: top [int]
    _denominator: bottom [int > 0]
    ""
    def __mul__(self, q):
        ""
        Return Product of self, q
        Makes a new Fraction; does not modify contents of self or q
        ""
        assert type(q) == Fraction
        bot = self._denominator * q._denominator
        top = (self._numerator * q._denominator)
        return Fraction(top, bot)
```

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

Python converts to

```
>>> r = p.__mul__(q)
```

Operator overloading uses method in object on left.

**Operator Overloading: Addition**

```python
class Fraction(object):
    ""
    Instance attributes:
    _numerator: top [int]
    _denominator: bottom [int > 0]
    ""
    def __add__(self, q):
        ""
        Return sum of self, q
        Makes a new Fraction
        ""
        assert type(q) == Fraction
        bot = self._denominator * q._denominator
        top = self._numerator * q._denominator + self._denominator * q._numerator
        return Fraction(top, bot)
```

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

Python converts to

```
>>> r = p.__add__(q)
```

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

```python
def __eq__(self, q):
    if type(q) == Fraction:
        return left == right
    elif type(q) == int:
        return left * q == right
    else:
        return False
```

```python
def __mul__(self, q):
    if type(q) == Fraction:
        return self._numerator * q._numerator
        * q._denominator
```

```python
top = self._numerator
bot = self._denominator
```

Example of this method

```python
>>> r = p.__mul__(q)
```

is Versus ==

- p == q evaluates to True
  - But only because method __eq__ compares contents
- p is q evaluates to False
  - Compares folder names
  - Cannot change this
  - Always use (x is None) not (x == None)

Solution: Look at Argument Type

- Overloading use left type
  - p*q => p.__mul__(q)
  - Done for us automatically
  - Looks in class definition
- What about type on right?
  - Have to handle ourselves
  - Can implement with ifs
  - Write helper for each type
  - Send to appropriate helper

```python
def __mul__(self, q):
    if type(q) == Fraction:
        return Fraction(self._numerator * q._numerator,
                        self._denominator * q._denominator)
```

A Better Multiplication

- 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

Advanced Example: Pixels

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