

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:** $\frac{1}{2} * \frac{3}{4} = \frac{3}{8}$
- 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: an int
    # _denominator: an int > 0

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

1

Problem: Doing Math is Unwieldy

What We Want	What We Get
$\left(\frac{1}{2} + \frac{1}{3} + \frac{1}{4}\right) * \frac{5}{4}$	<pre>>>> p = Fraction(1,2) >>> q = Fraction(1,3) >>> r = Fraction(1,4) >>> s = Fraction(5,4) >>> (p.add(q.add(r))).mult(s)</pre>

Why not use the standard Python math operations?

This is confusing!

2

Operator Overloading

- Many operators in Python a 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

3

Returning to Fractions

What We Want	Operator Overloading
$\left(\frac{1}{2} + \frac{1}{3} + \frac{1}{4}\right) * \frac{5}{4}$	<ul style="list-style-type: none"> • Python has methods that correspond to built-in ops <ul style="list-style-type: none"> ▪ <code>__add__</code> corresponds to + ▪ <code>__mul__</code> corresponds to * ▪ <code>__eq__</code> corresponds to == ▪ Not implemented by default • To overload operators you implement these methods

Why not use the standard Python math operations?

4

Operator Overloading: Multiplication

<pre>class Fraction(object): """Instance is a fraction n/d""" # _numerator: an int # _denominator: an int > 0 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)</pre>	<pre>>>> p = Fraction(1,2) >>> q = Fraction(3,4) >>> r = p*q Python converts to >>> r = p.__mul__(q)</pre> <div style="border: 1px solid green; border-radius: 15px; padding: 5px; background-color: #e0ffe0; text-align: center; margin-top: 10px;"> <p>Operator overloading uses method in object on left.</p> </div>
---	---

5

Operator Overloading: Addition

<pre>class Fraction(object): """Instance is a fraction n/d""" # _numerator: an int # _denominator: an int > 0 def __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)</pre>	<pre>>>> p = Fraction(1,2) >>> q = Fraction(3,4) >>> r = p+q Python converts to >>> r = p.__add__(q)</pre> <div style="border: 1px solid green; border-radius: 15px; padding: 5px; background-color: #e0ffe0; text-align: center; margin-top: 10px;"> <p>Operator overloading uses method in object on left.</p> </div>
--	---

6

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



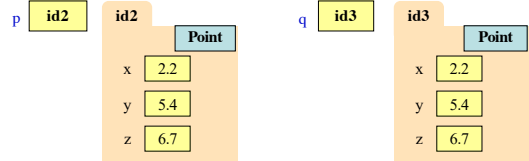
```
class Fraction(object):
    """Instance is a fraction n/d"""
    #_numerator: an int
    #_denominator: an int > 0

    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
```

7

is Versus ==

- `p is q` evaluates to **False**
 - Compares folder names
 - Cannot change this
- `p == q` evaluates to **True**
 - But only because method `__eq__` compares contents



Always use `(x is None)` not `(x == None)`

8

Recall: Overloading Multiplication

```
class Fraction(object):
    """Instance is a fraction n/d"""
    #_numerator: an int
    #_denominator: an int > 0

    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)
```

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

Python converts to

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

Can only multiply fractions. But ints "make sense" too.

9

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
 - Check type in method
 - Send to appropriate helper

```
class Fraction(object):
    ...
    def __mul__(self,q):
        """Returns: Product of self, q
        Precondition: q a Fraction or int"""
        if type(q) == Fraction:
            return self._mulFrac(q)
        elif type(q) == int:
            return self._mulInt(q)
    ...
    def _mulInt(self,q): # Hidden method
        return Fraction(self._numerator*q,
            self._denominator)
```

10

A Better Multiplication

```
class Fraction(object):
    ...
    def __mul__(self,q):
        """Returns: Product of self, q
        Precondition: q a Fraction or int"""
        if type(q) == Fraction:
            return self._mulFrac(q)
        elif type(q) == int:
            return self._mulInt(q)
    ...
    def _mulInt(self,q): # Hidden method
        return Fraction(self._numerator*q,
            self._denominator)
```

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

Python converts to

>>> r = p.__mul__(q) # OK!
```

See frac3.py for a full example of this method

11

We Have Come Full Circle

- On the first day, saw that a **type** is both
 - a set of *values*, and
 - the *operations* on them
- In Python, **all values are objects**
 - Everything has a folder in the heap
 - Just ignore it for immutable, basic types
- In Python, **all operations are methods**
 - Each operator has a double-underscore helper
 - Looks at type of object on left to process

12