Why bother with operator overloading?

- What Does `str()` Do On Objects?
  - Does **not** display contents
  - `str(p)`
  - `<Point object at 0x1007a90>`
  - Must add a special method
    - `__str__` for `str()`
    - `__repr__` for backquotes
  - Could get away with just one
    - Backquotes require `__repr__`
    - `str()` can use `__repr__` (if `__str__` is not there)

- Converting Values to Strings
  - **str()** Function
    - Usage: `str(<expression>)`
      - Evaluates the expression
      - Converts it into a string
    - How does it convert?
      - `str(1) → '1'`
      - `str(True) → 'True'`
      - `str('abc') → 'abc'`
      - `str(Point((0.0,0.0,0.0))) → '<Point (0.0,0.0,0.0)>'`
  - Backquotes
    - Usage: `'<expression>'`
      - Evaluates the expression
      - Converts it into a string
    - How does it convert?
      - `'1' → '1'`
      - `'True' → 'True'`
      - `'abc' → 'abc'`
      - `Point(') → '<class 'Point'> (0.0,0.0,0.0)'`

- Challenge: Implementing Fractions
  - Python has many built-in math types, but not all
    - Want to add a new type
    - Want to be able to add, multiply, divide etc.
    - Example: `1/3 * 3/2 = ½`
  - Can do this with a class
    - Objects are fractions
    - Have built-in methods to implement `+`, `/`, etc...
    - **Operator overloading**

- Operator Overloading: Multiplication
  - `class Fraction(object):
    """Instance attributes:
    numerator: top   [int]
    denominator: bottom [int > 0]"
    def __mul__(self, q):
      """Returns: Product of self, q
      Makes a new Fraction, does not modify contents of self or q"
      assert type(q) == Fraction
      top = self.numerator * q.numerator
      bot = self.denominator * q.denominator
      return Fraction(top,bot)
  >>> 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.

- 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 4 x 4`
**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 `_` operator on variables
  - `(x is y)` True if `x`, `y` contain the same folder name
- Check if variable is empty: `x is None` (x == None)

**Hiding Methods From Access**

- Put underscore in front of a method will make it hidden
- Will not show up in `help()`
- But it is still there…
- Hidden methods
  - Can be used as helpers inside the same class
  - But it is bad style to use them outside of this class
- Can do same for attributes
  - Underscore makes it hidden
  - Do not use outside of class

**Data Encapsulation**

- Idea: Force the user to only use methods
- Do not allow direct access of attributes

<table>
<thead>
<tr>
<th>Setter Method</th>
<th>Getter Method</th>
</tr>
</thead>
<tbody>
<tr>
<td>Used to change an attribute</td>
<td>Used to access an attribute</td>
</tr>
<tr>
<td>Replaces all assignment statements to the attribute</td>
<td>Replaces all usage of attribute in an expression</td>
</tr>
<tr>
<td><strong>Bad:</strong> <code>&gt;&gt;&gt; f.numerator = 5</code></td>
<td><strong>Bad:</strong> <code>&gt;&gt;&gt; x = 3*f.numerator</code></td>
</tr>
<tr>
<td><strong>Good:</strong> <code>&gt;&gt;&gt; f._setNumerator(5)</code></td>
<td><strong>Good:</strong> <code>&gt;&gt;&gt; x = 3*f._getNumerator()</code></td>
</tr>
</tbody>
</table>

**Enforcing Invariants**

- Idea: Restrict direct access
  - Only access via methods
  - Use asserts to enforce them
- Examples:
  - `def getNumerator(self):`  
  - `return self.numerator`  
  - `def setNumerator(self,value):`  
  - `assert type(value) == int`  
  - `self.numerator = value`

**Structure of a Proper Python Class**

- Docstring describing class
- Attributes are all hidden

- Getters and Setters  
- Initializer for the class
- Defaults for parameters.
- Python operator overloading
- Normal method definitions