### Important!

<table>
<thead>
<tr>
<th>YES</th>
<th>NO</th>
</tr>
</thead>
<tbody>
<tr>
<td><strong>class Point3</strong>(object):</td>
<td><strong>class Point3</strong>:</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** |

### What Does `str()` Do On Objects?

- Does NOT display contents
  ```python
  >>> p = Point(1,2,3)
  >>> str(p)
  '<Point x: 1, y: 2, z: 3>'
  ```
- Must add a special method
  ```python
  def __str__(self):
      """Return string with contents"""
      return "<Point x: {self.x}, y: {self.y}, z: {self.z}>"
  ```
- Could get away with just one
  - `str()` can use `__repr__` (if `__str__` is not there)
  - `__repr__` for backquotes

### 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: \(\frac{1}{2} \times \frac{3}{4} = \frac{3}{8}\)
- Can do this with a class
  - Objects are fractions
  - Have built-in methods to implement `+`, `*`, `/`, etc...
  - Operator overloading

### Converting Values to Strings

- **str() Function**
  - Evaluates the expression
  - Converts it into a string
  - How does it convert?
    - `str(1) → '1'`
    - `str(True) → 'True'`
    - `str('abc') → 'abc'`
    - `str(Fraction(2,3)) → '('0.666...0.666...,0.666...0.666...)'`

- **Backquotes**
  - Evaluates the expression
  - Converts it into a string
  - How does it convert?
    - `1 → '1'`
    - `True → 'True'`
    - `'abc' → 'abc'`
    - `Fraction(2,3) → "class Point: (0.666...0.666...0.666...0.666...0.666...0.666...)'`

### Operator Overloading: Multiplication

- **class Fraction(object):**
  - """Instances attributes numerator: top [int] denominator: bottom [int > 0]"""
  ```python
  def __mul__(self, q):
      """Return Product of self, q Makes a new Fraction; does not modify contents of self or q"
      numerator = self.numerator * q.numerator
denominator = self.denominator * q.denominator
  return Fraction(numerator, denominator)
  ```

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

- **Operator overloading uses method in object on left.**

### Comparing Objects for Equality

- **Earlier in course, we saw**
  - **class Fraction(object):**
    - """Instances attributes numerator: top [int] denominator: bottom [int > 0]"""
  ```python
  def __eq__(self, q):
      """Return 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
  ```
- **Ex**: cross multiplying
  - \(\frac{1}{2} \times \frac{2}{3} = \frac{4}{6}\)
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 is 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 is bad)

Broken statements to the attribute
Replaces all assignments to the attribute
Used to change an attribute
Do not allow direct access to attributes

Idea
Can do same for attributes
Hidden methods
Method will make it hidden
Put underscore in front of a class

Hidden 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 of 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

Enforcing Invariants

- Idea: Restrict direct access
  - Only access via methods
  - Use asserts to enforce them
- Examples:
  - def __eq__(self, q):
    - """Returns True if self, q equal, False if not, or q not a Fraction"
    - if self == q: return True
    - if not (self, q) is Fraction:
      - return False
  - def __ne__(self, q):
    - """Returns True if self, q equal, False if not, or q not a Fraction"
    - return not self == q

- These are just comments!

Evaluation

- __eq__ evaluates to True
  - But only because method __eq__ compares contents
- __ne__ evaluates to False
  - Compares folder names
  - Cannot change this

Data Encapsulation

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

setter method
- Used to change an attribute
- Replaces all assignment statements to the attribute
Bad:
  >>> f.numerator = 5
Good:
  >>> f.setNumerator(5)

getter method
- Used to access an attribute
- Replaces all usage of attribute in an expression
Bad:
  >>> x = 3*f.numerator
Good:
  >>> x = 3*f.getNumerator()

Structure of a Proper Python Class

class Fraction(object):
    """Instance attributes:
    numerator: [int]   denominator: [int > 0]"

    def __init__(self, numerator, denominator):
        self.numerator = numerator
        self.denominator = denominator

    @property
    def numerator(self):
        return self.numerator

    @numerator.setter
    def numerator(self, value):
        type(value) == int and self.denominator > 0

    @property
    def denominator(self):
        return self.denominator

    @denominator.setter
    def denominator(self, value):
        type(value) == int

    def __eq__(self, q):
        """Returns: if q is a Fraction"
        return type(q) != Fraction:
    return False

    def __ne__(self, q):
        """Returns: if q is a Fraction"
        return not self == q

    def normalize(self):
        """This Fraction in reduced form?"

    def _is_denominator(self, d):
        """Return True if d valid denom"
        return type(d) == int and d > 0

    def __add__(self, q):
        """Returns: Sum of self, q"

    def __sub__(self, q):
        """Returns: Difference of self, q"

    def __mul__(self, q):
        """Returns: Product of self, q"

    def __div__(self, q):
        """Returns: Quotient of self, q"

    def normalize(self):
        """This Fraction is reduced form?"

    def getNumerator(self):
        """Returns: numerator"
        return self.numerator

    def setNumerator(self, value):
        type(value) == int and self.denominator > 0

    def getDenominator(self):
        """Returns: denominator"
        return self.denominator

    def setDenominator(self, value):
        type(value) == int

    def normalize(self):
        """This Fraction is reduced form?"

    def __eq__(self, q):
        """Returns True if self, q equal, False if not, or q not a Fraction"
        if self == q: return True
        if not (self, q) is Fraction:
            return False
        return not self == q

    def __ne__(self, q):
        """Returns True if self, q equal, False if not, or q not a Fraction"
        return not self == q

    def normalize(self):
        """This Fraction is reduced form?"

    def __eq__(self, q):
        """Returns True if self, q equal, False if not, or q not a Fraction"
        if self == q: return True
        if not (self, q) is Fraction:
            return False
        return not self == q

    def __ne__(self, q):
        """Returns True if self, q equal, False if not, or q not a Fraction"
        return not self == q

    def normalize(self):
        """This Fraction is reduced form?"

    def __eq__(self, q):
        """Returns True if self, q equal, False if not, or q not a Fraction"
        if self == q: return True
        if not (self, q) is Fraction:
            return False
        return not self == q

    def __ne__(self, q):
        """Returns True if self, q equal, False if not, or q not a Fraction"
        return not self == q

    def normalize(self):
        """This Fraction is reduced form?"

    def __eq__(self, q):
        """Returns True if self, q equal, False if not, or q not a Fraction"
        if self == q: return True
        if not (self, q) is Fraction:
            return False
        return not self == q

    def __ne__(self, q):
        """Returns True if self, q equal, False if not, or q not a Fraction"
        return not self == q

    def normalize(self):
        """This Fraction is reduced form?"

    def __eq__(self, q):
        """Returns True if self, q equal, False if not, or q not a Fraction"
        if self == q: return True
        if not (self, q) is Fraction:
            return False
        return not self == q

    def __ne__(self, q):
        """Returns True if self, q equal, False if not, or q not a Fraction"
        return not self == q

    def normalize(self):
        """This Fraction is reduced form?"

    def __eq__(self, q):
        """Returns True if self, q equal, False if not, or q not a Fraction"
        if self == q: return True
        if not (self, q) is Fraction:
            return False
        return not self == q

    def __ne__(self, q):
        """Returns True if self, q equal, False if not, or q not a Fraction"
        return not self == q

    def normalize(self):
        """This Fraction is reduced form?"

    def __eq__(self, q):
        """Returns True if self, q equal, False if not, or q not a Fraction"
        if self == q: return True
        if not (self, q) is Fraction:
            return False
        return not self == q

    def __ne__(self, q):
        """Returns True if self, q equal, False if not, or q not a Fraction"
        return not self == q

    def normalize(self):
        """This Fraction is reduced form?"

    def __eq__(self, q):
        """Returns True if self, q equal, False if not, or q not a Fraction"
        if self == q: return True
        if not (self, q) is Fraction:
            return False
        return not self == q

    def __ne__(self, q):
        """Returns True if self, q equal, False if not, or q not a Fraction"
        return not self == q

    def normalize(self):
        """This Fraction is reduced form?"

    def __eq__(self, q):
        """Returns True if self, q equal, False if not, or q not a Fraction"
        if self == q: return True
        if not (self, q) is Fraction:
            return False
        return not self == q

    def __ne__(self, q):
        """Returns True if self, q equal, False if not, or q not a Fraction"
        return not self == q

    def normalize(self):
        """This Fraction is reduced form?"