Important!

YES

```
class Point(object):
    """Instances are 3D points
    Attributes:
        x: x-coord [float]
        y: y-coord [float]
        z: z-coord [float]"""
    …
```

3.0-Style Classes
Well-Designed

NO

```
class Point:
    """Instances are 3D points
    Attributes:
        x: x-coord [float]
        y: y-coord [float]
        z: z-coord [float]"""
    …
```

"Old-Style" Classes
Very, Very Bad

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)'

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)"

What Does str() Do On Objects?

- Does **NOT** display contents
  ```
  >>> p = Point(1,2,3)
  >>> 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)

```
class Point(object):
    """Instances are points in 3d space"""
    …
    def __str__(self):
        """Returns: string with contents"""
        return '('+self.x + ',' +
                   self.y + ',' +
                   self.z + ')'

    def __repr__(self):
        """Returns: unambiguous string"""
        return str(self.__class__)+
                   str(self)
```

Challenge: Implementing Fractions

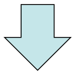
- Python has many built-in math types, but not all
  - Want to add a new type
  - Want to be able to add, multiply, divide etc.
  - Example: ½*¾ = ⅜
- Can do this with a class
  - Objects are fractions
  - Have built-in methods to implement +, *, /, etc…
  - **Operator overloading**

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

    def __init__(self,n=0,d=1):
        """Initializer: makes a Frac"""
        self.numerator = n
        self.denominator = d

    def __str__(self):
        """Returns: Fraction as string"""
        return (str(self.numerator)
                 +'/'+str(self.denominator))
```

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



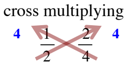
Python converts to

```
>>> 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



```
class Fraction(object):
    """Instance attributes:
        numerator:   top      [int]
        denominator: bottom [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
        rght = self.denominator*q.numerator
        return left == rght
```

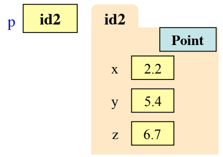
Issues With Overloading ==

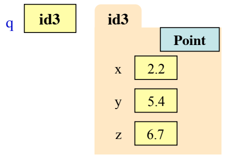
- Overloading == **does not** also overload comparison !=
  - Must implement __ne__
  - Why? Will see later
  - But (not x == y) is okay!
- What if you still want to compare Folder names?
  - Use is operator on variables
  - (x is y) True if x, y contain the same folder name
  - Check if variable is empty: x is None (x == None is bad)

```
class Fraction(object):
    ...
    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
        rght = self.denominator*q.numerator
        return left == rght

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

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)

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 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

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

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

    def __init__(self,n=0,d=1):
        assert self._is_denominator(d)
        self.numerator = n
        self.denominator = d
```

HIDDEN

Helper method

Enforcing Invariants

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

Invariants: Properties that are always true.

- These are just comments!

  >>> p = Fraction(1,2)

  >>> p.numerator = 'Hello'
- How do we prevent this?

- **Idea**: Restrict direct access
  - Only access via methods
  - Use asserts to enforce them
- Examples:

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

def setNumerator(self,value):
    """Sets numerator to value"""
    assert type(value) == int
    self.numerator = value
```

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):
    """Instances represent a Fraction
      Attributes:
        _numerator: [int]
        _denominator: [int > 0]"""

    def getNumerator(self):
        """Returns: Numerator of Fraction"""
    ...
    def __init__(self,n=0,d=1):
        """Initializer: makes a Fraction"""
    ...
    def __add__(self,q):
        """Returns: Sum of self, q"""
    ...
    def normalize(self):
        """Puts Fraction in reduced form"""
    ...
```

Docstring describing class
Attributes are all **hidden**

Getters and Setters.

Initializer for the class.
Defaults for parameters.

Python operator overloading

Normal method definitions