Lecture 18

Methods and Operations
Announcements for This Lecture

Assignments

- **A4** Due Thursday at midnight
  - Hopefully you are on Task 4
  - Extra consultants available
- Will post **A5** on Thursday
  - Written assignment like A2
  - Needs material from next Tues
- Will also post **A6** as well
  - Not due until November 19
  - Want to avoid exam crunch

Lab this Week

- Simple class exercise
  - Fill in predefined methods
  - Setting you up for A6…

Exams

- Moved to handback room
  - Located in Gates 216
  - Open 12-4:30 daily
- Regrades still open this week
Important!

**YES**

```python
class Point3(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**

```python
class Point3:
    '''Instances are 3D points
    Attributes:
    x: x-coord [float]
y: y-coord [float]
z: z-coord [float]'''
...
```

“Old-Style” Classes
Very, Very Bad

10/27/15 Methods and Operations 3
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**: simplefrac.py

```python
class Fraction(object):
    """Instance is a fraction n/d
    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
```

10/27/15 Methods and Operations 4
## 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!
Problem: Doing Math is Unwieldy

What We Want

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

Why not use the standard Python math operations?

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!
Recall: The \_\_init\_\_ Method

def \_\_init\_\_(self, n, s, b):
    
    """Initializer: creates a Worker

    Has last name n, SSN s, and boss b

    Precondition: n a string, s an int in range 0..999999999, and b either a Worker or None.

    self.lname = n
    self.ssn = s
    self.boss = b"""

two underscores

\[w = \text{Worker('Obama', 1234, None)}\]
Recall: The __init__ Method

```python
def __init__(self, n, s, b):
    """Initializer: creates a Worker
    ""
    Has last name n, SSN s, and boss b
    Precondition: n a string, s an int in range 0..999999999, and b either a Worker or None.
    self.lname = n
    self.ssn = s
    self.boss = b
```

Are there other special methods that we can use?
### Example: Converting Values to Strings

**str() Function**

- **Usage:** `str(<expression>)`
  - Evaluates the expression
  - Converts it into a string

- **How does it convert?**
  - `str(2) → '2'`
  - `str(True) → 'True'`
  - `str('True') → 'True'`
  - `str(Point3()) → '(0.0,0.0,0.0)'`

**Backquotes**

- **Usage:** `~<expression>`
  - Evaluates the expression
  - Converts it into a string

- **How does it convert?**
  - `~2 → '2'`
  - `~True → 'True'`
  - `~'True' → ''True''`
  - `~Point3() → '<class 'Point3'> (0.0,0.0,0.0)''`
**Example: Converting Values to Strings**

### `str()` Function

- **Usage:** `str(<expression>)`
  - Evaluates the expression
  - Converts it into a string

- **How does it convert?**
  - `str(2) → '2'`
  - `str(True) → 'True'`
  - `str('True') → 'True'`
  - `str(Point3()) → '(0.0,0.0,0.0)'`

### Backquotes

- **Backquotes are for unambiguous representation**

- **How does it convert?**
  - `\`2` → '2'`
  - `\`True` → "True"
  - `\`'True'` → "'True'"
  - `\`Point3()` → "<class 'Point3'> (0.0,0.0,0.0)"
What Does \texttt{str()} Do On Objects?

• Does \textbf{NOT} display contents
  >>> p = Point3(1,2,3)
  >>> \texttt{str(p)}
  
  '\texttt{<Point3 object at 0x1007a90>}'

• Must add a special method
  ▪ \texttt{\_\_str\_\_} for \texttt{str()}
  ▪ \texttt{\_\_repr\_\_} for backquotes

• Could get away with just one
  ▪ Backquotes require \texttt{\_\_repr\_\_}
  ▪ \texttt{str()} can use \texttt{\_\_repr\_\_}
    (if \texttt{\_\_str\_\_} is not there)

\texttt{class Point3(object):}
  
  """Instances are points in 3d space""

  ...

  \texttt{def \_\_str\_\_(self):}
  """Returns: string with contents""

  \texttt{return '(}+self.x + ',', +
  
  \texttt{self.y + ',', +}
  
  \texttt{self.z + ')'}

  \texttt{def \_\_repr\_\_(self):}
  """Returns: unambiguous string""

  \texttt{return str(self.\_\_class\_\_)+}
  
  \texttt{str(self)}
What Does \texttt{str()} Do On Objects?

- Does \textbf{NOT} display contents
  
  ```python
  >>> p = Point3(1,2,3)
  >>> str(p)
  '<Point3 object at 0x1007a90>'
  ```

- Must add a special method
  - \texttt{\_\_str\_\_} for \texttt{str()}
  - \texttt{\_\_repr\_\_} for backquotes

- Could get away with just one
  - Backquotes require \texttt{\_\_repr\_\_}
  - \texttt{str()} can use \texttt{\_\_repr\_\_}
    (if \texttt{\_\_str\_\_} is not there)

```
class Point3(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)
```

\texttt{\_\_repr\_\_} using \texttt{\_\_str\_\_} as helper
Special Methods in Python

- Have seen three so far
  - __init__ for initializer
  - __str__ for str()
  - __repr__ for backquotes
- Start/end with 2 underscores
  - This is standard in Python
  - Used in all special methods
  - Also for special attributes
- For a complete list, see
  http://docs.python.org/reference/datamodel.html

```python
class Point3(object):
    """Instances are points in 3D space""
    ...

def __init__(self, x=0, y=0, z=0):
    """Initializer: makes new Point3""
    ...

def __str__(self, q):
    """Returns: string with contents""
    ...

def __repr__(self, q):
    """Returns: unambiguous string""
    ...
```

10/27/15 Methods and Operations 13
Returning to Fractions

What We Want

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

Why not use the standard Python math operations?

Operator Overloading

- Python has methods that correspond to built-in ops
  - \(__add__\) corresponds to +
  - \(__mul__\) corresponds to *
  - Not implemented by default
- Implementing one allows you to use that op on your objects
  - Called operator overloading
  - Changes operator meaning
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.
class Fraction(object):
    """Instance attributes:
    numerator:    top       [int]
    denominator: bottom [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)

>>> 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

```
4  1  2  4
2   4
```

```python
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
```

10/27/15  Methods and Operations  17
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)

```python
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
```

10/27/15
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)