Lecture 17

Methods and Encapsulation
Announcements for This Lecture

Assignment 4

• Due on Sunday at midnight
  ▪ Turned on in CMS today
• Looking at Consultant Hours
  ▪ Thursday was very busy
  ▪ Sat. hours might be possible
• Survey extra important!
• Will post A5 at end of week
  ▪ Written assignment like A2
  ▪ Can do at the same time

Lab this Week

• Simple class exercise
  ▪ Fill in predefined methods
  ▪ Setting you up for A6…
• Exams are handed back
  ▪ Organized by lab section
  ▪ Unclaimed exams will go to handback room on Thurs
• Regrades turned on in CMS
  ▪ For major mistakes only
Recursion and A4

Wrong

- Recursion on pmap keys
  - Argument must get smaller
  - pmap should never change
- Also do not loop over keys
- Example: Autocomplete

```python
keys = pmap.keys()
accum = []
for word in keys:
    # Add word if it extends prefix
    return accum
```

Right

- Recursion on prefix
  - Extend prefix via pmap
  - Compute extended answer
  - Combine with others
- Example: Autocomplete

```python
pmap = {'a':['t','x',''], ...

Extensions of 'a' are
- 'a', plus
- Extensions of 'at', plus
- Extensions of 'ax'
```
Important!

YES

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

    ...
```

3.0-Style Classes
Well-Designed

NO

```python
class Point:

    """Instances are 3D points\n    Attributes:
        x: x-coord [float]
        y: y-coord [float]
        z: z-coord [float]\n    """

    ...
```

“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)"`
# Converting Values to Strings

<table>
<thead>
<tr>
<th><strong>str() Function</strong></th>
<th><strong>Backquotes</strong></th>
</tr>
</thead>
<tbody>
<tr>
<td><strong>Usage:</strong> ( \text{str(&lt;expression&gt;)} )</td>
<td><strong>Usage:</strong> Backquotes are for <em>unambiguous</em> representation</td>
</tr>
<tr>
<td>▪ Evaluates the expression</td>
<td>▪ Evaluates the expression</td>
</tr>
<tr>
<td>▪ Converts it into a string</td>
<td>▪ Converts it into a string</td>
</tr>
<tr>
<td><strong>How does it convert?</strong></td>
<td><strong>How does it convert?</strong></td>
</tr>
<tr>
<td>▪ ( \text{str(1)} ) → '1'</td>
<td>▪ <code>'1</code> → '1'</td>
</tr>
<tr>
<td>▪ ( \text{str(True)} ) → 'True'</td>
<td>▪ <code>'True</code> → 'True'</td>
</tr>
<tr>
<td>▪ ( \text{str('abc')} ) → 'abc'</td>
<td>▪ <code>'abc</code> → 'abc'</td>
</tr>
<tr>
<td>▪ ( \text{str(Point())} ) → '(0.0,0.0,0.0)'</td>
<td>▪ <code>Point()</code> → &quot;&lt;class 'Point'&gt; (0.0,0.0,0.0)&quot;</td>
</tr>
</tbody>
</table>

What type is this value?

The value’s type is clear

`methods and encapsulation`
What Does \texttt{str()} Do On Objects?

- Does \textbf{NOT} display contents
  
  \begin{verbatim}
  >>> p = Point(1,2,3)
  >>> str(p)
  '<Point object at 0x1007a90>'
  \end{verbatim}

- 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)

---

```python
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)
```

10/27/14

Methods and Encapsulation
What Does \texttt{str()} Do On Objects?

\begin{itemize}
  \item Does \textbf{NOT} display contents
    \begin{verbatim}
    >>> p = Point(1,2,3)
    >>> str(p)
    '<Point object at 0x1007a90>'
    \end{verbatim}
  \item Must add a special method
    \begin{itemize}
      \item \texttt{\_\_str\_} for \texttt{str()}
      \item \texttt{\_\_repr\_} for backquotes
    \end{itemize}
  \item Could get away with just one
    \begin{itemize}
      \item Backquotes require \texttt{\_\_repr\_}
      \item \texttt{str()} can use \texttt{\_\_repr\_} \\
           (if \texttt{\_\_str\_} is not there)
    \end{itemize}
\end{itemize}

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

\begin{verbatim}
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)
\end{verbatim}
Special Methods in Python

- Have seen three so far
  - __init__ for initializer
  - __str__ for str()
  - __repr__ for backquotes
- Start/end w/ two 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 Point(object):
    """Instances are points in 3D space""
    ...
    def __init__(self,x=0,y=0,z=0):
        """Initializer: makes new Point""
        ...
    def __str__(self,q):
        """Returns: string with contents""
        ...
    def __repr__(self,q):
        """Returns: unambiguous string""
        ...
```

10/27/14 Methods and Encapsulation 9
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**

```python
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))
```

10/27/14

Methods and Encapsulation
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
>>> r = p.__mul__(q)

Python converts to

Operator overloading uses method in object on left.
**Operator Overloading: Addition**

```python
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)
```

```python
>>> p = Fraction(1,2)
>>> q = Fraction(3,4)
>>> r = p+q
```

Python converts to

```python
>>> 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

```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
        right = self.denominator * q.numerator
        return left == right
```

10/27/14

Methods and Encapsulation
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
```

10/27/14
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)`
Getting Information About a Class

- Recall the `help()` function shows module contents
  - Works on classes too
  - **Example**: `help(Point)`
- Can even use on object
  - In that case, runs help on the class of that object
  - **Example**: `help(p)`
- Shows all methods
  - And `class` attributes

```python
class Fraction(__builtin__.object):
    """Instance is a fraction n/d"""
    
    Instance Attributes:
    numerator:    top part    [int]
    denominator: bottom part [int > 0]

    Methods defined here:
    __add__(self, other)
    """Returns: Sum of self and other as a new Fraction. Does not modify contents of self or other."""

    Precondition: other is a Fraction
```

...
Summary + Files

• Methods with double underscores are special
  ▪ Used to implement operators (e.g. +, ==, <)
  ▪ Great for implementing mathematical objects
  ▪ **Example:** fraction.py

• Attributes cannot enforce invariants
  ▪ Want to wrap them in getters, setters
  ▪ Setters use asserts to enforce invariants
  ▪ **Example:** betterfraction.py

• **But how does class RGB work?**
  ▪ No setters, getters but enforces its invariants
  ▪ **Advanced programming topic.** Ask outside of class.