Lecture 16

Advanced Class Design
Announcements for This Lecture

Reading

• Today: Chapter 17
• Tuesday: Chapter 18

Assignment 4

• Due next Tuesday
• Consultants on Saturday
• Next assignment short!
  ▪ Only a week to work on it
  ▪ Give more time for later 2

Assignment 3

• A3 is finally graded
  ▪ Mean: 94, Median: 97
  ▪ Mean Time: 7-8 hours
  ▪ Generally successful

• Please do the surveys!
  ▪ A lot did not complete
  ▪ Must finish individually
  ▪ 1% of your final grade
  ▪ Extension for tonight

10/16/12
Exam: Regrading Policy

• For major oversights (do not “point lawyer”)
  ▪ We missed something on the back of a sheet
  ▪ Your code actually does work when you try it
  ▪ We totaled up your score wrong
• Reserve the right to take off points in regrade
  ▪ But not if it is just a point total mistake
• What to do if you want a regrade?
  ▪ Submit a regrade request online
  ▪ Bring exam to Molly Trufant in Upson 415A
Last Time: Saw Several Special Methods

- Each added new features
  - `__init__` for constructor
  - `__str__` for `str()`
  - `__repr__` for backquotes
- Each one started and ended with double underscores
  - This is standard in Python
  - Used in all special methods
- 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):
    """Constructor: makes new Point""
    ...

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

def __repr__(self,q):
    """Returns: unambiguos string""
    ...
```
Getting Information About a Class

- Recall the `help()` function to see 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)`
- Useful to see attributes and methods of the class

```python
class Fraction(__builtin__.object):
    Methods defined here:
    |
    |
    __init__(self)
    |
    __str__(self)
    |
    distanceTo(self,q)
    |
Data and other attributes defined here:
|
|x = 0.0
|y = 0.0
|z = 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: $\frac{1}{2} \times \frac{3}{4} = \frac{3}{8}$
- Can do this with a class
  - Objects are fractions
  - Have built-in methods to implement $+$, $\times$, $/$, etc…
  - **Operator overloading**

```python
class Fraction(object):
    numerator = 0  # int
    denominator = 1  # int > 0

    def __init__(self, n=0, d=1):
        """Constructor: 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):
    numerator = 0  # int
denominator = 1  # 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

>>> r = p.__mul__(q)

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

```python
class Fraction(object):
    numerator = 0  # int
denominator = 1  # 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
>>> r = p.__add__(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

```
class Fraction(object):
    numerator = 0  # int
denominator = 1  # 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
```

4 1 2 4
\[ \frac{2}{2} \times \frac{4}{4} \]
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 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
        right = self.denominator*q.numerator
        return left == right
    
def __ne__(self,q):
        """Returns: False if self, q equal, True if not, or q not a Fraction""
        return not self == q
```

10/16/12
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)
Enforcing Invariants

Invariants:
Properties that are always true.

class Fraction(object):
    numerator = 0  # int
    denominator = 1  # int > 0

• 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

10/16/12  Advanced Classes
Hiding Fields From Access

- Put underscore in front of field name to make it **hidden**
  - Will not show up in `help()`
  - But it is still there…

```python
>>> help(Fraction)
class Fraction(__builtinc__object):
  Methods defined here:
  ...
  def getNumerator(self):
    """Returns: numerator""
    return self._numerator
  ...
  ...
  (No data attributes shown)
```

```python
class Fraction(object):
  _numerator = 0  # int, hidden
  _denominator = 1  # int > 0, hidden
  ...
  def getNumerator(self):
    """Returns: numerator""
    return self._numerator
  ...
  def setNumerator(self, value):
    """Sets numerator to value""
    assert type(value) == int
    self._numerator = value
```

10/16/12
Advanced Classes 13
Properties: Invisible Setters and Getters

class Fraction(object):
    _numerator = 0  # int, hidden
    _denominator = 1  # int > 0, hidden
...
@property
def numerator(self):
    """Numerator value of Fraction
    Invariant: must be an int""
    return self._numerator

@numerator.setter
def numerator(self, value):
    assert type(value) == int
    self._numerator = value

>>> p = Fraction(1,2)
>>> x = p.numerator()
>>> p.numerator = 2
>>> p.numerator(2)

Python converts to

10/16/12  Advanced Classes  14
Properties: Invisible Setters and Getters

```python
class Fraction(object):
    _numerator = 0  # int, hidden
    _denominator = 1  # int > 0, hidden
...
@property
def numerator(self):
    """Numerator value of Fraction
    Invariant: must be an int"
    return self._numerator
@numerator.setter
def numerator(self, value):
    assert type(value) == int
    self._numerator = value
```

- Specifies that next method is the **getter** for property of the same name as the method.
- Docstring describing property.
- Property uses **hidden** field.
- Specifies that next method is the **setter** for property whose name is numerator.
Properties: Invisible Setters and Getters

**class** Fraction(object):

```python
    _numerator = 0  # int, hidden
    _denominator = 1  # int > 0, hidden
...
@property
    def numerator(self):
        """Numerator value of Fraction
        Invariant: must be an int"
        return self._numerator

@numerator.setter
    def numerator(self, value):
        assert type(value) == int
        self._numerator = value
```

**Goal:** Data Encapsulation
Protecting your data from other, “clumsy” users.

- Only the **getter** is required!
- If no **setter**, then the attribute is “immutable”.

**Attributes = Properties**
(All *fields* should be hidden)
**Structure of a Proper Python Class**

```python
class Fraction(object):
    '''Instances represent a Fraction'''
    _numerator = 0  # int, hidden
...
@property
    def numerator(self):
        '''Numerator value of Fraction'''
...
    def __init__(self, n=0, d=1):
        '''Constructor: makes a Fraction'''
...
    def __add__(self, q):
        '''Returns: Sum of self, q'''
...
    def normalize(self):
        '''Puts Fraction in reduced form'''
...
```

- **Docstring describing class**
- **Field defaults; all hidden**
- **Properties for each field. Put invariants in getter.**
- **Constructor for class. Defaults for parameters.**
- **Python operator overloading**
- **Normal method definitions**
Summary + Files

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

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

• **Properties** provide invisible getters, setters
  ▪ Attributes = properties + non-hidden fields
  ▪ **Example**: bestfraction.py