Lecture 18

Advanced Class Design
## Announcements for This Lecture

### This Week

- **Assignment 4 due tonight**
  - Consultants 4:30-9:30
  - TAs and I have office hours
  - Late penalty is -10pts/day
- **Survey due by next class**
  - Do as individuals!
  - Same format as before
- **Reading**: Chapter 18

### Assignment 5

- **Brand new assignment**
  - We are still beta testing it
  - There will probably be bugs
  - Expect regular fixes/updates
  - But we will be lenient…
- **Will be longer!**
  - Start the assignment earlier
  - Finish Part I before prelim
- **Will go online tomorrow**
# 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

## 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)'`

Backquotes are for **unambiguous** representation.

What type is this value? It's clear from the representation.
What Does \texttt{str()} Do On Objects?

- Does \textbf{NOT} display contents
  
  ```python
  >>> p = Point(1,2,3)
  >>> str(p)
  '<Point 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)

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

- Does \textbf{NOT} display contents
  
  >>> p = Point(1,2,3)
  >>> \texttt{str(p)}
  
  \texttt{<Point 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)

\textbf{class Point(object)}:

  """Instances are points in 3d space""

  ... 

  \textbf{def \_\_str\_\_(self)}:

    """Returns: string with contents""

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

  \textbf{def \_\_repr\_\_(self)}:

    """Returns: unambiguous string""

    \textbf{return} \texttt{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 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""
    ...
```

10/16/12 Advanced Classes
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))
```

10/16/12
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

>>> 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.

10/16/12
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
  
  \[
  \begin{array}{c}
  4 \\
  -2 \\
  \end{array} \quad \begin{array}{c}
  1 \\
  2 \\
  \end{array} \quad \begin{array}{c}
  2 \\
  4 \\
  \end{array} \quad \begin{array}{c}
  4 \\
  \end{array}
  \]

**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)

```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 Advanced Classes
**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
    ...```
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

```python
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
```

Helper method
From Last Time: **Data Encapsulation**

class Time(object):
    """Instances represent times of day.
    Instance Attributes:
    _hour: hour of day [int in 0..23]
    _min: minute of hour [int in 0..59]"

    def getMin(self):
        """Returns: min attribute"
        return self._min

    def setMin(self, mins):
        """Alters min attribute to be mins
        Pre: mins is in 0..59"
        assert type(mins) == int
        assert 0 <= mins and mins < 60
        self._min = mins

**Getter**

**Setter**

**Naming Convention**

The underscore means “should not access the attribute directly.”

**Do this for all of your attributes**

Precondition is same as attribute invariant.
Properties: Invisible Setters and Getters

class Fraction(object):
    
    """Instance attributes:
    _numerator: [int]
    _denominator: [int > 0]""

@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

>>> x = p.numerator()

>>> p.numerator = 2

>>> p.numerator(2)
Properties: Invisible Setters and Getters

```python
class Fraction(object):
    
    """Instance attributes:
    _numerator:    [int]
    _denominator: [int > 0]"

@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** attribute.
- Specifies that next method is the **setter** for property whose name is numerator.
Properties: Invisible Setters and Getters

```python
class Fraction(object):
    '''Instance attributes:
    _numerator: [int]
    _denominator: [int > 0]'''

@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”.
- Replace **Attributes w/ Properties** (Users cannot tell difference)
class Fraction(object):
    """Instances represent a Fraction
    Attributes:
    _numerator: [int]
    _denominator: [int > 0]"

@property
def numerator(self):
    """Numerator value 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

Properties for *each* attribute.
Put invariants in *getter*.

Initializer for the 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

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

• **Properties** provide invisible **getters**, **setters**
  - Make attributes **hidden**, and use properties instead
  - **Example**: bestfraction.py