Lecture 12

Methods and Operations
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

10/7/16  Methods and Operations  2
Case Study: Fractions

- Want to add a new \textit{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 \textit{objects}
  - Operations are \textit{methods}

- \textbf{Example:} simplefrac.py

\begin{verbatim}

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

\end{verbatim}
### Problem: Doing Math is Unwieldy

<table>
<thead>
<tr>
<th>What We Want</th>
<th>What We Get</th>
</tr>
</thead>
</table>
| \[
\left(\frac{1}{2} + \frac{1}{3} + \frac{1}{4}\right) \times \frac{5}{4}
\] | >>> 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

<table>
<thead>
<tr>
<th>What We Want</th>
<th>What We Get</th>
</tr>
</thead>
</table>
| \[
\left( \frac{1}{2} + \frac{1}{3} + \frac{1}{4} \right) \times \frac{5}{4}
\] | >>> p = Fraction(1,2)                            |
|                                           | >>> q = Fraction(1,3)                            |
|                                           | >>> r = Fraction(1,4)                            |
|                                           | >>> s = Fraction(5,4)                            |
|                                           | >>> (p.add(q.add(r))).mult(s)                     |

Why not use the standard Python math operations?

This is confusing!
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"
```

\`\texttt{w = Worker('Garoppolo', 1234, None)}\`

Called by the constructor

<table>
<thead>
<tr>
<th>id8</th>
<th>Worker</th>
</tr>
</thead>
<tbody>
<tr>
<td>lname</td>
<td>‘Garoppolo'</td>
</tr>
<tr>
<td>ssn</td>
<td>1234</td>
</tr>
<tr>
<td>boss</td>
<td>None</td>
</tr>
</tbody>
</table>
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?

```python
w = Worker('Plunkett', 1234, None)
```
**Example: 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> <code>str(&lt;expression&gt;)</code></td>
<td><strong>Usage:</strong> <code>\&lt;expression&gt;</code></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>▪ <code>str(2)</code> → '2'</td>
<td>▪ <code>\2</code> → '2'</td>
</tr>
<tr>
<td>▪ <code>str(True)</code> → 'True'</td>
<td>▪ <code>\True</code> → 'True'</td>
</tr>
<tr>
<td>▪ <code>str('True')</code> → 'True'</td>
<td>▪ <code>\'True'</code> → &quot;'True'&quot;</td>
</tr>
<tr>
<td>▪ <code>str(Point3())</code> → <code>(0.0,0.0,0.0)</code></td>
<td>▪ <code>\Point3()</code> → &quot;&lt;class 'Point3'&gt; (0.0,0.0,0.0)&quot;</td>
</tr>
</tbody>
</table>
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)`

What type is this value? The value’s type is clear
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)

```python
class Point3(object):
  
  
  
  
  
  
  
  
  
  
  
  
  def \_\_str\_(self):
    return '(+self.x + ',', ' +
            '    self.y + ',', ' +
            '    self.z + ')'  # for unambiguous string

  def \_\_repr\_(self):
    return str(self.__class__) +
          str(self)
```

10/7/16 Methods and Operations
What Does \texttt{str()} Do On Objects?

- Does **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)

```python
class Point3(object):
    '''Instances are points in 3d space'''

    ...

    def __str__(self):
        '''Returns: string with contents'''
        return '(' + str(self.x) + ',' + str(self.y) + ',' + str(self.z) + ')

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

\texttt{\_\_repr\_} using \texttt{\_\_str\_} as helper

Gives the class name
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""
        ...
```
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
  - \texttt{\_\_add\_\_} corresponds to +
  - \texttt{\_\_mul\_\_} corresponds to *
    - Not implemented by default
- Implementing one allows you to use that op on your objects
  - Called operator overloading
  - Changes operator meaning
**Operator Overloading: Multiplication**

```python
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

```python
>>> r = p.__mul__(q)
```

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

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

10/7/16 Methods and Operations 16
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
```

10/7/16 Methods and Operations 17
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** \((x \text{ is None})\) **not** \((x == \text{None})\)
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

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

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

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

Helper method

10/7/16 Methods and Operations 19
Enforcing Invariants

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

    Invariants:
    Properties that are always true.

10/7/16 Methods and Operations 20
Data Encapsulation

**Idea**: Force the user to only use methods

**Do not allow direct access of attributes**

<table>
<thead>
<tr>
<th>Setter Method</th>
<th>Getter Method</th>
</tr>
</thead>
<tbody>
<tr>
<td>• Used to change an attribute</td>
<td>• Used to access an attribute</td>
</tr>
<tr>
<td>• Replaces all assignment statements to the attribute</td>
<td>• Replaces all usage of attribute in an expression</td>
</tr>
<tr>
<td><strong>Bad</strong>:</td>
<td><strong>Bad</strong>:</td>
</tr>
<tr>
<td>&gt;&gt;&gt; f.numerator = 5</td>
<td>&gt;&gt;&gt; x = 3*f.numerator</td>
</tr>
<tr>
<td><strong>Good</strong>:</td>
<td><strong>Good</strong>:</td>
</tr>
<tr>
<td>&gt;&gt;&gt; f.setNumerator(5)</td>
<td>&gt;&gt;&gt; x = 3*f.getNumerator()</td>
</tr>
</tbody>
</table>
Data Encapsulation

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

def getDenominator(self):
    '''Returns: numerator attribute'''
    return self._denominator

def setDenominator(self, d):
    '''Alters denominator to be d
    Pre: d is an int > 0'''
    assert type(d) == int
    assert 0 < d
    self._denominator = d
```

- **Getter**
- **Setter**

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

- Precondition is same as attribute invariant.
## Mutable vs. Immutable Attributes

<table>
<thead>
<tr>
<th>Mutable</th>
<th>Immutable</th>
</tr>
</thead>
<tbody>
<tr>
<td>• Can change value directly</td>
<td>• Can’t change value directly</td>
</tr>
<tr>
<td>▪ If class invariant met</td>
<td>▪ May change “behind scenes”</td>
</tr>
<tr>
<td>▪ <strong>Example</strong>: t.color</td>
<td>▪ <strong>Example</strong>: t.x</td>
</tr>
<tr>
<td>• Has both getters and setters</td>
<td>• Has only a getter</td>
</tr>
<tr>
<td>▪ Setters allow you to change</td>
<td>▪ No setter means no change</td>
</tr>
<tr>
<td>▪ Enforce invariants w/ asserts</td>
<td>▪ Getter allows limited access</td>
</tr>
</tbody>
</table>

May ask you to differentiate on the exam
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"
    ...