Lecture 19

Operators and Abstraction
Announcements

Reading

• **Tuesday**: Chapter 18
• **Thursday** reading online

Assignments

• A4 due **tonight** at Midnight
  ▪ 10 pts per day late
  ▪ Consultants available tonight
• A5 posted today, A6 on Sat.
  ▪ See included *micro*-deadlines

Regrades

• Today is last day to request
  ▪ Show it to me after class
  ▪ I will verify if it is valid
• **Then** request regrade in CMS

• Prelim, Nov 9th 7:30-9:00
  ▪ Material up to November 2nd
  ▪ Recursion + Loops + Classes
• **S/U Students are exempt**
• **Conflict with Prelim time?**
  ▪ Prelim 2 Conflict on CMS
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**: frac1.py

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

10/26/17 Operators and Abstraction 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 with a class
  - Values are fraction objects
  - Operations are methods
- **Example:** `frac1.py`

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

Reminder: Hide attributes, use getters/setters
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

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

Why not use the standard Python math operations?

This is confusing!
Special Methods in Python

• Have seen three so far
  ▪ `__init__` for initializer
  ▪ `__str__` for `str()`
  ▪ `__repr__` for `repr()`

• Start/end with 2 underscores
  ▪ This is standard in Python
  ▪ Used in all special methods
  ▪ Also for special attributes

• We can **overload operators**
  ▪ Give new meaning to +, *, -

```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):
    """Returns: string with contents""
    ...

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

10/26/17
Operator Overloading

- Many operators in Python are special symbols
  - +, -, /, *, ** for mathematics
  - ==, !=, <, > for comparisons
- The meaning of these symbols depends on type
  - 1 + 2 vs 'Hello' + 'World'
  - 1 < 2 vs 'Hello' < 'World'
- Our new type might want to use these symbols
  - We overload them to support our new type
Returning to Fractions

What We Want

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

Operator Overloading

- Python has methods that correspond to built-in ops
  - `__add__` corresponds to +
  - `__mul__` corresponds to *
  - `__eq__` corresponds to `==`
  - Not implemented by default
- To overload operators you implement these methods

Why not use the standard Python math operations?
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
>>> r = p.__add__(q)

Python converts to

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):
    """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
```

10/26/17  Operators and Abstraction  12
is Versus ==

• **p is q** evaluates to **False**
  - Compares folder names
  - Cannot change this

  ![Diagram](image)

  p  id2
    - id2
      - Point
      - x 2.2
      - y 5.4
      - z 6.7

  q  id3
    - id3
      - Point
      - x 2.2
      - y 5.4
      - z 6.7

• **p == q** evaluates to **True**
  - But only because method `__eq__` compares contents

Always use `(x is None)` **not** `(x == None)`
Structure of a Proper Python Class

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"
...
Recall: Overloading Multiplication

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

    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 = 2  # an int
>>> r = p * q

>>> r = p.__mul__(q)  # ERROR

Python converts to

Can only multiply fractions. But ints “make sense” too.
```

10/26/17 Operators and Abstraction
Solution: Look at Argument Type

- Overloading use **left** type
  - \( p \times q \Rightarrow p.__mul__(q) \)
  - Done for us automatically
  - Looks in class definition
- What about type on **right**?
  - Have to handle ourselves
- Can implement with ifs
  - Write helper for each type
  - Check type in method
  - Send to appropriate helper

```python
class Fraction(object):
    ...
    def __mul__(self,q):
        """Returns: Product of self, q
        Precondition: q a Fraction or int"
        if type(q) == Fraction:
            return self.__mulFrac(q)
        elif type(q) == int:
            return self.__mulInt(q)
    ...
    def __mulInt(self,q):
        # Hidden method
        return Fraction(self.__numerator*q, self.__denominator)
```

10/26/17

Operators and Abstraction
A Better Multiplication

```python
class Fraction(object):
    ...

    def __mul__(self, q):
        """Returns: Product of self, q
        Precondition: q a Fraction or int"
        if type(q) == Fraction:
            return self._mulFrac(q)
        elif type(q) == int:
            return self._mulInt(q)
    ...

    def _mulInt(self, q):
        return Fraction(self._numerator * q, self._denominator)
```

```python
>>> p = Fraction(1,2)
>>> q = 2  # an int
>>> r = p * q
>>> r = p.__mul__(q)  # OK!
```

Python converts to

>>> r = p.__mul__(q)  # OK!

See frac3.py for a full example of this method
What Do We Get This Time?

```python
class Fraction(object):
...

def __mul__(self, q):
    """Returns: Product of self, q
    Precondition: q a Fraction or int""
    if type(q) == Fraction:
        return self._mulFrac(q)
    elif type(q) == int:
        return self._mulInt(q)
...

def _mulInt(self, q): # Hidden method
    return Fraction(self._numerator * q, self._denominator)

>>> p = Fraction(1,2)
>>> q = 2 # an int
>>> r = q*p
```

A: Fraction(2,2)
B: Fraction(1,1)
C: Fraction(2,4)
D: Error
E: I don’t know
class Fraction(object):
...

def __mul__(self,q):
    """Returns: Product of self, q
    Precondition: q a Fraction or int"""
    if type(q) == Fraction:
        return self._mulFrac(q)
    elif type(q) == int:
        return self._mulInt(q)
...

def _mulInt(self,q): # Hidden method
    return Fraction(self._numerator*q,
                    self._denominator)

>>> p = Fraction(1,2)
>>> q = 2 # an int
>>> r = q*p

Meaning determined by left. Variable q stores an int.

B: Fraction(1,1)
C: Fraction(2,4)
D: Error  CORRECT
E: I don’t know
The Python Data Model

**Note:** Slicing is done exclusively with the following three methods. A call like

```python
a[1:2] = b
```

is translated to

```python
a[slice(1, 2, None)] = b
```

and so forth. Missing slice items are always filled in with `None`.

**object.** `__getitem__(self, key)`

Called to implement evaluation of `self[key]`. For sequence types, the accepted keys should be integers and slice objects. Note that the special interpretation of negative indexes (if the class wishes to emulate a sequence type) is up to the `__getitem__()` method. If `key` is of an inappropriate type, `TypeError` may be raised; if of a value outside the set of indexes for the sequence (after any special interpretation of negative values), `IndexError` should be raised. For mapping types, if `key` is missing (not in the container), `KeyError` should be raised.

**Note:** `for` loops expect that an `IndexError` will be raised for illegal indexes to allow proper detection of the end of the sequence.

**object.** `__missing__(self, key)`

Called by `dict.__getitem__()` to implement `self[key]` for dict subclasses when key is not in the dictionary.

**object.** `__setitem__(self, key, value)`

Called to implement assignment to `self[key]`. Same note as for `__getitem__()`.

**object.** `__delitem__(self, key)`

Called to implement deletion of `self[key]`. Same note as for `__getitem__()`.
Advanced Example: A6 Pixels

• **Image is list of list of RGB**
  - But this is really slow
  - **Faster**: byte buffer (???)
  - Beyond scope of course

• **Compromise**: Pixels class
  - Has byte buffer attribute
  - *Pretends* to be list of tuples
  - You can slice/iterate/etc…

• Uses data model to do this

[(255,255,255), (255,255,255), ...]
Advanced Example: A6 Pixels

- Image is list of list of RGB
  - But this is really slow
  - Faster: byte buffer (???)
  - Beyond scope of course
- Compromise: Pixels class
  - Has byte buffer attribute
  - Pretends to be list of tuples
  - You can slice/iterate/etc…
- Uses data model to \textbf{lie to you!}

\textbf{Abstraction}: Making a type easier to use by hiding details from the user
Advanced Topic Warning!

The following will not be on the exam

If you ask “Will this be on the Exam”

we will be 😠
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

>>> p = Fraction(1, 2)

```
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
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

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)

10/26/17