Lecture 20

Operators and Abstraction
## Announcements for Today

### Reading
- **Prelim, Nov 8\textsuperscript{th} 5:15 or 7:30**
  - Same break-up as last time
  - But will swap times assigned
- **Material up to November 1**
  - Review posted this weekend
  - Recursion + Loops + Classes
- **Conflict with Prelim time?**
  - Prelim 2 Conflict on CMS
  - SDS students must submit!
  - LAST DAY TO SUBMIT

### Assignments
- A4 graded by tomorrow
  - Will cover survey next week
- A5 to be graded Saturday
  - Returned via Gradescope
  - Similar policies to A2
- Need to be working on A6
  - Should have Dataset done
  - Cluster finished by Sunday
  - Best way to study for exam

11/1/18  Operators and Abstraction  2
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
```

11/1/18
Operators and Abstraction
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**
  - 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

11/1/18 Operators and Abstraction
# Problem: Doing Math is Unwieldy

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

This is confusing!
Problem: Doing Math is Unwieldy

What We Want

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

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)
```

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, q):
        """Returns: string with contents""
        ...

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

11/1/18 Operators and Abstraction
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

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

Why not use the standard Python math operations?

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
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

>>> r = p.__mul__(q)

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)

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

```
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
```

11/1/18
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)*
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"
    ...

Docstring describing class
Attributes are all hidden

Getters and Setters.

Initializer for the class.
Defaults for parameters.

Python operator overloading

Normal method definitions
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)
```

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

Python converts to

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

Can only multiply fractions. But ints “make sense” too.
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 if statements
  - 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)
```

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

```python
>>> r = p._mul__(q)  # OK!
```

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

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
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)
```

```python
>>> 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`.

```python
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.

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

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

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

Called to implement assignment to `self[key]`. Same note as for `__getitem__()`. This should only be implemented for mappings if the objects support changes to the values for keys, or if new keys can be added, or for sequences if elements can be replaced. The same exceptions should be raised for improper `key` values as for the `__getitem__()` method.

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

Called to implement deletion of `self[key]`. Same note as for `__getitem__()`. This should only be implemented for mappings if the objects support removal of keys, or for sequences if elements can be removed from the sequence. The same exceptions should be raised for improper `key` values as for the `__getitem__()` method.
Advanced Example: 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: 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 *lie to you!*
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

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)

Python converts to

>>> x = p.numerator

Python converts to

>>> x = p.numerator()

>>> p.numerator = 2

>>> p.numerator(2)
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

Decorator specifies that next method is getter for property of the same name as method

Docstring describing property

Property uses hidden attribute.

Decorator specifies that next method is the setter for property whose name is numerator.
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)