Lecture 20

Operators and Abstraction
Announcements for Today

Two Weeks Out!

- **Prelim, Nov 21st at 7:30**
  - Same rooms as last time
  - Day after A6 is due
- **Material up to Nov. 12**
  - Study guide this weekend
  - Recursion + Loops + Classes
- **Conflict with Prelim?**
  - Prelim 2 Conflict on CMS
  - SDS students must submit!

Assignments

- A4 graded by Saturday
  - Will cover survey next week
- A5 also graded by Saturday
  - Returned via Gradescope
  - Similar policies to A2
- Need to be working on A6
  - Finish Image this weekend
  - Finish Filter by next Thurs
  - Best way to study for exam

11/7/19  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: an int
    # _denominator: an int > 0
    def __init__(self, n=0, d=1):
        """Init: makes a Fraction""
        self._numerator = n
        self._denominator = d
```

11/7/19 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: an int
    # _denominator: an 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

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)
```

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""
    ...
```

11/7/19 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

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

Operator Overloading

- Python has methods that correspond to built-in ops
  - \texttt{__add__} corresponds to +
  - \texttt{__mul__} corresponds to *
  - \texttt{__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 is a fraction n/d"""
    # _numerator: an int
    # _denominator: an 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.
class Fraction(object):
    """Instance is a fraction n/d"""
    # _numerator: an int
    # _denominator: an 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
    \[
    \frac{4}{2} \times \frac{1}{2} = \frac{4}{4} = 1
    \]

```python
class Fraction(object):
    """Instance is a fraction n/d"""
    # _numerator: an int
    # _denominator: an 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
```
is Versus ==

- p is q evaluates to \textbf{False}
  - Compares folder names
  - Cannot change this

- p == q evaluates to \textbf{True}
  - But only because method \_\_eq\_\_ compares contents

Always use (x is None) \textbf{not} (x == None)
class Fraction(object):

    '''Instance is a fraction n/d'''
    
    # _numerator: an int
    # _denominator: an 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 is a fraction n/d"""
    # _numerator: an int
    # _denominator: an 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)
```

```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**
  - \( p \cdot q \Rightarrow p._{\text{mul}}(q) \)
  - 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)
```

11/7/19
Operators and Abstraction
A Better Multiplication

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 = p*q
# OK!

Python converts to

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

See frac3.py for a full example of this method
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?

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

\[ a[1:2] = b \]

is translated to

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

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.
We Have Come Full Circle

• On the first day, saw that a type is both
  ▪ a set of values, and
  ▪ the operations on them

• In Python, all values are objects
  ▪ Everything has a folder in the heap
  ▪ Just ignore it for immutable, basic types

• In Python, all operations are methods
  ▪ Each operator has a double-underscore helper
  ▪ Looks at type of object on left to process
Advanced Topic Warning!

The following will not be on the exam

If you ask “Will this be on the Exam”

we will be 🙄
class Fraction(object):
    """Instance is a fraction n/d"""
    _numerator: an int
    _denominator: an 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()

>>> x = p.numerator(2)

Python converts to

>>> p.numerator = 2

Python converts to

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

```python
class Fraction(object):
    """Instance is a fraction n/d"""
    # _numerator: an int
    # _denominator: an 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

```python
class Fraction(object):
    """Instance is a fraction n/d"""
    # _numerator:  an int
    # _denominator: an 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)