Challenge: Implementing Fractions

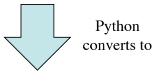
- Python has many built-in math types, but not all
  - Want to add a new type
  - Want to be able to add, multiply, divide etc.
  - Example: ½*¾ = ⅜
- Can do this with a class
  - Objects are fractions
  - Have built-in methods to implement +, *, /, etc…
  - **Operator overloading**

```
class Fraction(object):
    numerator = 0     # int
    denominator = 1  # int > 0

    def __init__(self,n=0,d=1):
        """Constructor: makes a Frac"""
        self.numerator = n
        self.denominator = d

    def __str__(self):
        """Returns: Fraction as string"""
        return (str(self.numerator)
                +'/'+str(self.denominator))
```

Operator Overloading: Multiplication

```
class Fraction(object):
    numerator = 0     # int
    denominator = 1  # 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

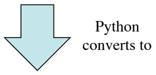
```
>>> r = p.__mul__(q)
```

Operator overloading uses method in object on left.

Operator Overloading: Addition

```
class Fraction(object):
    numerator = 0     # int
    denominator = 1  # 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
```



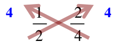
Python converts to

```
>>> 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



```
class Fraction(object):
    numerator = 0     # int
    denominator = 1  # 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
```

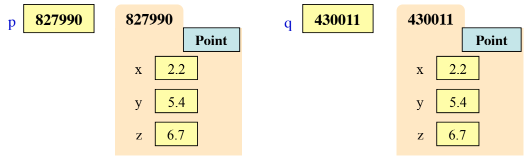
Issues With Overloading ==

- 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)

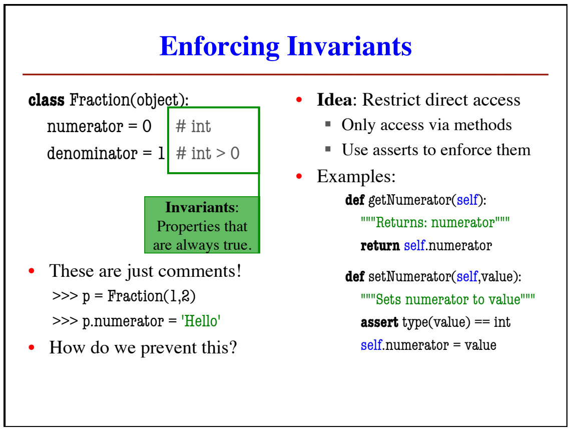
```
class Fraction(object):
    …
    def __eq__(self,q):
        """Returns: True if self, q equal,
        False if not, or q not a Fraction"""
        if type(q) != Fraction:
            return False
        left = self.numerator*q.denominator
        rght = self.denominator*q.numerator
        return left == rght

    def __ne__(self,q):
        """Returns: False if self, q equal,
        True if not, or q not a Fraction"""
        return not self == q
```

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)



Enforcing Invariants

```
class Fraction(object):
    numerator = 0    # int
    denominator = 1  # int > 0
```

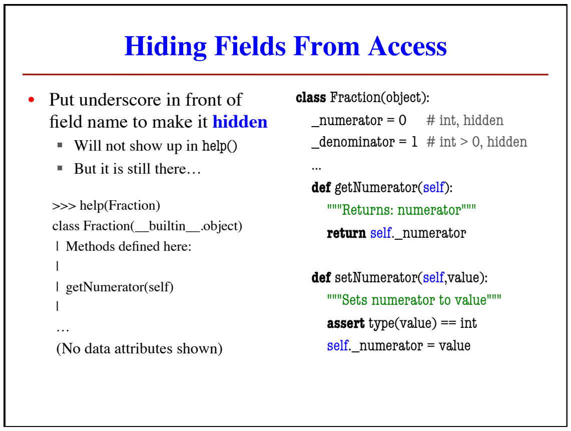
Invariants: Properties that are always true.

- 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
```

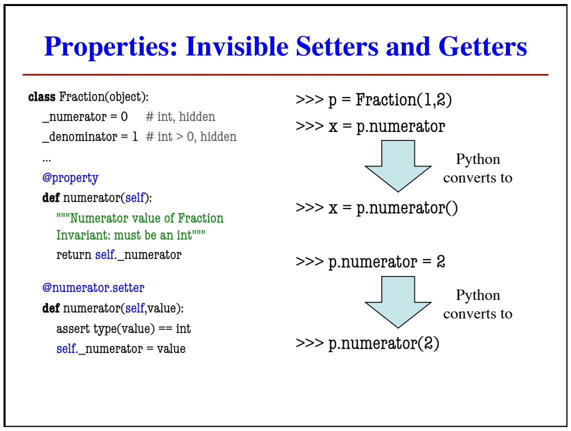


Hiding Fields From Access

- Put underscore in front of field name to make it **hidden**
  - Will not show up in help()
  - But it is still there…

```
>>> help(Fraction)
class Fraction(__builtin__.object)
 |  Methods defined here:
 |
 |  getNumerator(self)
 |
 …
 (No data attributes shown)
```

```
class Fraction(object):
    _numerator = 0    # int, hidden
    _denominator = 1  # int > 0, hidden
    …
    def getNumerator(self):
        """Returns: numerator"""
        return self._numerator

    def setNumerator(self,value):
        """Sets numerator to value"""
        assert type(value) == int
        self._numerator = value
```



Properties: Invisible Setters and Getters

```
class Fraction(object):
    _numerator = 0    # int, hidden
    _denominator = 1  # int > 0, hidden
    …
    @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
```

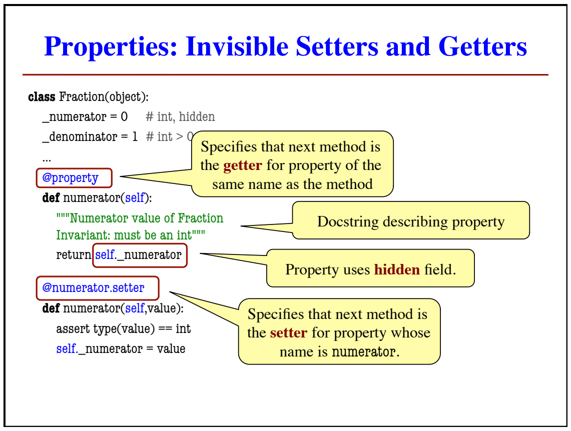
Python converts to

```
>>> x = p.numerator()
```

```
>>> p.numerator = 2
```

Python converts to

```
>>> p.numerator(2)
```



Properties: Invisible Setters and Getters

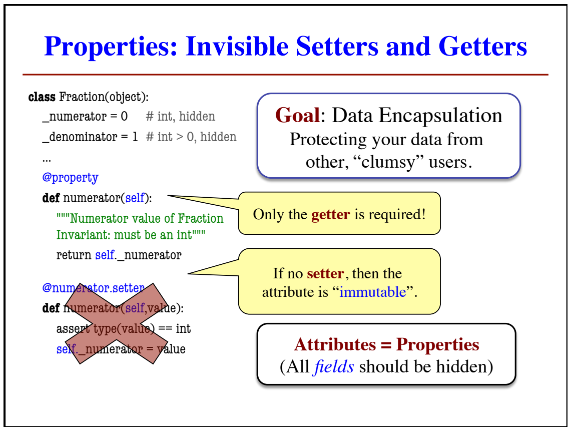
```
class Fraction(object):
    _numerator = 0    # int, hidden
    _denominator = 1  # int > 0
    …
    @property
    def numerator(self):
        """Numerator value of Fraction
        Invariant: must be an int"""
        return self._numerator

    @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** field.

Specifies that next method is the **setter** for property whose name is numerator.



Properties: Invisible Setters and Getters

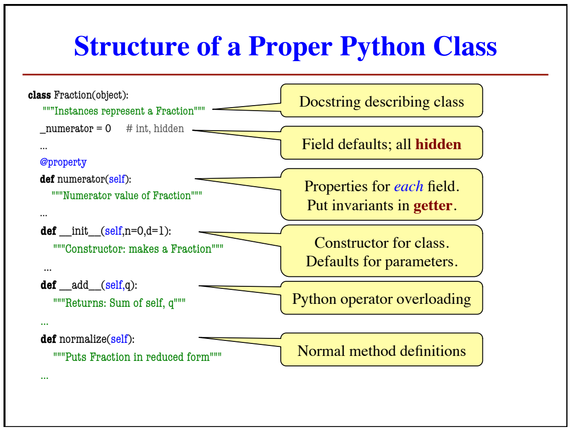
```
class Fraction(object):
    _numerator = 0    # int, hidden
    _denominator = 1  # int > 0, hidden
    …
    @property
    def numerator(self):
        """Numerator value of Fraction
        Invariant: must be an int"""
        return self._numerator

    @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”.

Attributes = Properties
(All *fields* should be hidden)



Structure of a Proper Python Class

```
class Fraction(object):
    """Instances represent a Fraction"""
    _numerator = 0    # int, hidden
    …
    @property
    def numerator(self):
        """Numerator value of Fraction"""
    …
    def __init__(self,n=0,d=1):
        """Constructor: makes a Fraction"""
     …
    def __add__(self,q):
        """Returns: Sum of self, q"""
    …
    def normalize(self):
        """Puts Fraction in reduced form"""
    …
```

Docstring describing class

Field defaults; all **hidden**

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

Constructor for class.
Defaults for parameters.

Python operator overloading

Normal method definitions