

Lecture 22

Operators and Abstraction

Announcements for Today

Prelim 2

- **Prelim, Nov 21st at 7:30**
 - See webpage for rooms
 - Review **Sun Nov. 19 (TBA)**
- **Material up to Nov. 9**
 - Recursion + Loops + Classes
 - But need to know early stuff
- **Conflict with Prelim?**
 - Prelim 2 Conflict on CMS
 - SDS students must submit!

Assignments

- A4 graded by tomorrow
- A5 graded by Sunday
 - Returned via Gradescope
 - Similar policies to A2
 - Survey is now posted
- Need to be working on A6
 - Hopefully done with Task 3
 - Should be now on Task 4
 - Start Task 5 by Thursday

Announcements for Today

Prelim 2

- **Prelim, Nov 21st at 7:30**

- See webpage for rooms
- Review **Sum**

- **Material up**

- Recursion
- But need to

- **Conflict with Prelim?**

- Prelim 2 Conflict on CMS
- SDS students must submit!

Assignments

- A4 graded by tomorrow

- A5 graded by Sunday

Video Lessons

- **Videos 23.1-23.7** for **today**

- **Lessons 24, 25** for next time

via Gradescope

olicies to A2

ow posted

working on A6

property done with Task 3

- Should be now on Task 4

- Start Task 5 by Thursday

Quick Announcement on A6

- We ask you to use `isinstance` instead of `type`
 - In paragraph on enforcing preconditions
 - What do we mean by this?
- **Recall:** Type-based preconditions
 - **Precondition:** `x` is a `Player` object
 - `assert type(x) == Player`
 - `assert isinstance(x, Player)`
- More on this next lecture (**Thursday**)

This is *almost*
the same

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**: $\frac{1}{2} * \frac{3}{4} = \frac{3}{8}$
- Can do this with a class
 - **Values** are fraction **objects**
 - **Operations** are **methods**
- **Example**: `frac1.py`

```
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
```

Case Study: Fractions

- Want to add a new *type*
 - Values are fractions: $\frac{1}{2}$, $\frac{3}{4}$
 - Operations are methods
 - Example: `frac1.py`
- Can do
 - Values are fraction *objects*
 - Operations are *methods*
- **Example:** `frac1.py`

Reminder: Hide attributes, use getters/setters

```
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
```

Problem: Doing Math is Unwieldy

What We Want

$$\left(\frac{1}{2} + \frac{1}{3} + \frac{1}{4}\right) * \frac{5}{4}$$

What We Get

```
>>> 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

$$\left(\frac{1}{2} + \frac{1}{3} + \frac{1}{4}\right) * \frac{5}{4}$$

Why not use the standard Python math operations?

What We Get

```
>>> 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!

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 `+`, `*`, `-`

```
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"""
        ...
```

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

$$\left(\frac{1}{2} + \frac{1}{3} + \frac{1}{4}\right) * \frac{5}{4}$$

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

Operator Overloading: Multiplication

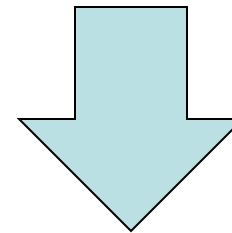
```
class Fraction(object):
    """Instance is a fraction n/d"""
    # _numerator:  an int
    # _denominator: an int > 0

    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.

Operator Overloading: Addition

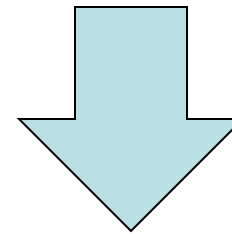
```
class Fraction(object):
    """Instance is a fraction n/d"""
    # _numerator:  an int
    # _denominator: an int > 0

    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
```



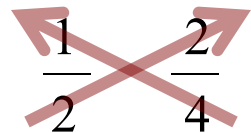
Python
converts to

```
>>> 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

$$4 \quad \frac{1}{2} \quad \frac{2}{4} \quad 4$$


```
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
```

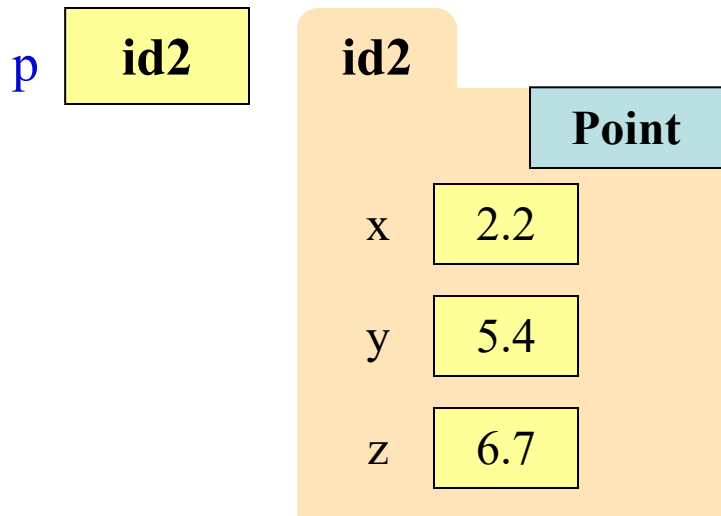
```
        right = self._denominator*q._numerator
```

```
        return left == right
```

is Versus ==

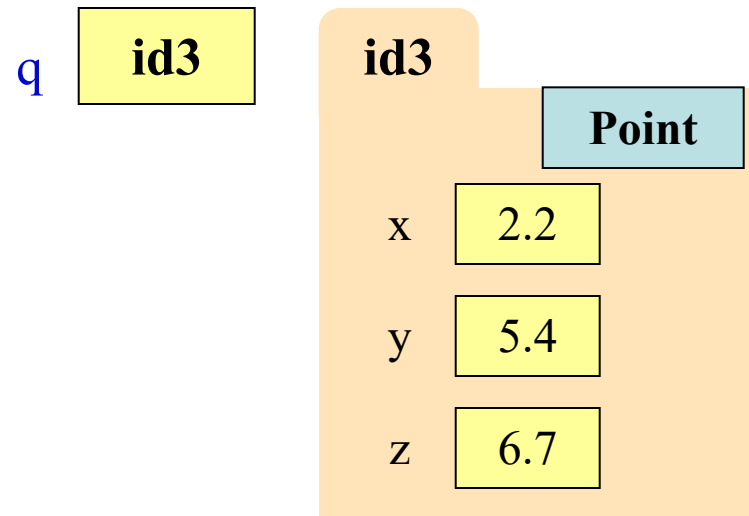
- `p is q` evaluates to **False**

- 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):
```

```
    """Instance is a fraction n/d"""
```

```
    # _numerator:  an int
```

```
    # _denominator: an int > 0
```

Docstring describing class
Attributes are all **hidden**

```
def getNumerator(self):
```

```
    """Returns: Numerator of Fraction"""
```

```
...
```

Getters and Setters.

```
def __init__(self,n=0,d=1):
```

```
    """Initializer: makes a Fraction"""
```

```
...
```

Initializer for the class.
Defaults for parameters.

```
def __add__(self,q):
```

```
    """Returns: Sum of self, q"""
```

```
...
```

Python operator overloading

```
def normalize(self):
```

```
    """Puts Fraction in reduced form"""
```

```
...
```

Normal method definitions

Recall: Overloading Multiplication

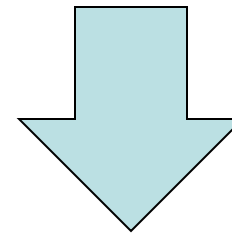
```
class Fraction(object):
    """Instance is a fraction n/d"""
    # _numerator:  an int
    # _denominator: an int > 0

    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 = 2 # an int
```

```
>>> r = p*q
```



Python
converts to

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

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

Solution: Look at Argument Type

- Overloading use **left** type
 - $p * q \Rightarrow p.__mul__(q)$
 - 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

```
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)
```

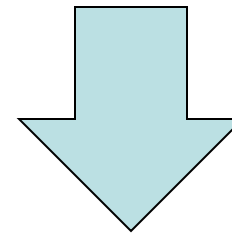
A Better Multiplication

```
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
```



Python
converts to

```
>>> 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?

```
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

<http://docs.python.org/3/reference/datamodel.html>

```
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__()`. 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.

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

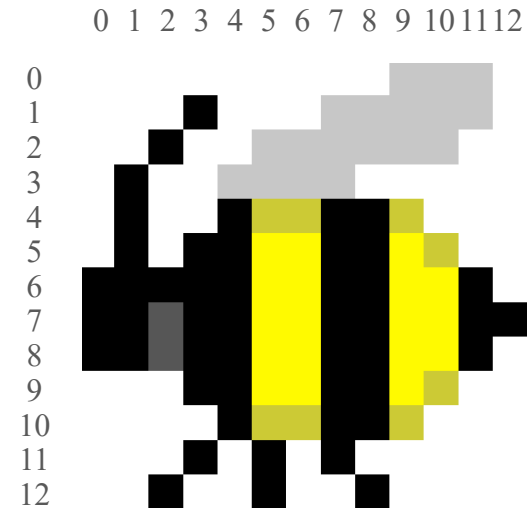
If you ask “Will this be on the Exam”

we will be



Example from Old 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), ...]

Example from Old A6: Pixels

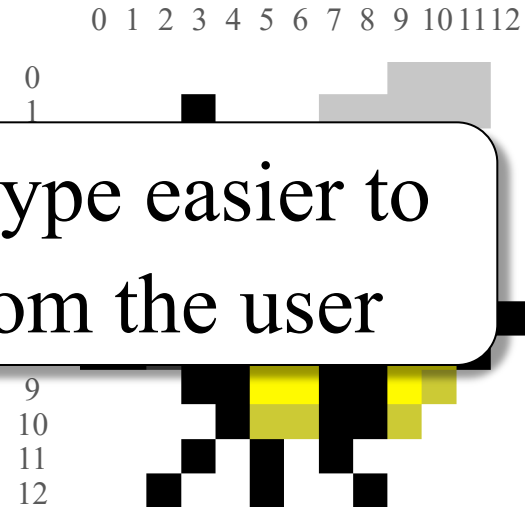
- Image is list of list of RGB

- **Abstraction:** Making a type easier to use by hiding details from the user

- **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!** [(255,255,255), (255,255,255), ...]

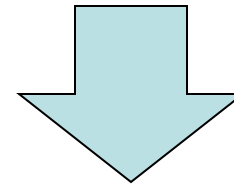


Properties: Invisible Setters and Getters

```
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)
```

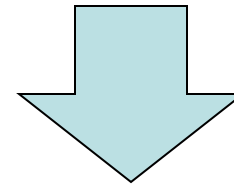
```
>>> x = p.numerator
```



Python
converts to

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

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



Python
converts to

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

Properties: Invisible Setters and Getters

```
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

```
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)