Lecture 9

Using Classes Effectively
Last Time: Saw Several Special Methods

- Each added new features
  - `__init__` for constructor
  - `__str__` for `str()`
  - `__repr__` for backquotes

- Each one started and ended with double underscores
  - This is standard in Python
  - Used in all special methods

- For a complete list, see http://docs.python.org/reference/datamodel.html

```python
class Point(object):
    """Instances are points in 3D space""
    ...
    def __init__(self, x=0, y=0, z=0):
        """Constructor: makes new Point""
        ...
    def __str__(self, q):
        """Returns: string with contents""
        ...
    def __repr__(self, q):
        """Returns: unambiguous string""
        ...
```

10/6/14 Using Classes Effectively
Important!

**YES**

```python
class Point(object):
    
    """Instances are 3D points
    Attributes:
    x: x-coord [float]
y: y-coord [float]
z: z-coord [float]""

    ...
```

3.0-Style Classes
Well-Designed

**NO**

```python
class Point:

    """Instances are 3D points
    Attributes:
    x: x-coord [float]
y: y-coord [float]
z: z-coord [float]""

    ...
```

“Old-Style” Classes
Very, Very Bad
Designing Types

- **Type**: set of values and the operations on them
  - int: (set: integers; ops: +, −, *, /, …)
  - Time (set: times of day; ops: time span, before/after, …)
  - Worker (set: all possible workers; ops: hire, pay, promote, …)
  - Rectangle (set: all axis-aligned rectangles in 2D; ops: contains, intersect, …)

- To define a class, think of a *real type* you want to make
  - Python gives you the tools, but does not do it for you
  - Physically, any object can take on any value
  - Discipline is required to get what you want
Making a Class into a Type

1. Think about what values you want in the set
   - What attributes? What values can they have?

2. Think about what operations you want
   - Often influences the previous question
   - To make (1) precise: write a class invariant
     - Statement we promise to keep true after every method call
   - To make (2) precise: write method specifications
     - Statement of what method does/what it expects (preconditions)
   - Write your code to make these statements true!
class Time(object):
    """Instances represent times of day.
    Instance Attributes:
    hour: hour of day [int in 0..23]
    min:  minute of hour [int in 0..59]"

    def __init__(self, hour, min):
        """The time hour:min.
        Pre: hour in 0..23; min in 0..59"

    def increment(self, hours, mins):
        """Move this time <hours> hours
        and <mins> minutes into the future.
        Pre: hours is int >= 0; mins in 0..59"

    def isPM(self):
        """Returns: this time is noon or later."""
Planning out a Class

class Rectangle(object):
    """Instances represent rectangular regions of the plane.
Instance Attributes:
    t: y coordinate of top edge [float]
    l: x coordinate of left edge [float]
    b: y coordinate of bottom edge [float]
    r: x coordinate of right edge [float]
For all Rectangles, l <= r and b <= t."""

def __init__(self, t, l, b, r):
    """The rectangle [l, r] x [t, b]
    Pre: args are floats; l <= r; b <= t"""

def area(self):
    """Return: area of the rectangle."""

def intersection(self, other):
    """Return: new Rectangle describing intersection of self with other."""
class Hand(object):
    """Instances represent a hand in cards.    
Instance Attributes:    
    cards: cards in the hand [list of card]    
This list is sorted according to the    
ordering defined by the Card class."""

def __init__(self, deck, n):
    """Draw a hand of n cards.    
Pre: deck is a list of >= n cards"""

def isFullHouse(self):
    """Return: True if this hand is a full house; False otherwise"""

def discard(self, k):
    """Discard the k-th card."""
Implementing a Class

- All that remains is to fill in the methods. (All?!)  
- When implementing methods:
  1. Assume preconditions are true
  2. Assume class invariant is true to start
  3. Ensure method specification is fulfilled
  4. Ensure class invariant is true when done
- Later, when using the class:
  - When calling methods, ensure preconditions are true
  - If attributes are altered, ensure class invariant is true
Implementing an Initializer

def __init__(self, hour, min):
    """The time hour:min.
    Pre: hour in 0..23; min in 0..59"""

    self.hour = hour
    self.min = min

This is true to start

You put code here

This should be true at the end
Implementing a Method

Instance variables:
- hour: hour of day [int in 0..23]
- min: minute of hour [int in 0..59]

```python
def increment(self, hours, mins):
    """Move this time <hours> hours and <mins> minutes into the future.
    Pre: hours [int] >= 0; mins in 0..59""
    self.min = self.min + min
    self.hour = (self.hour + hour +
                 self.min / 60)
    self.min = self.min % 60
    self.hour = self.hour % 24
```

This is true to start
What we are supposed to accomplish
This is also true to start
You put code here
This should be true at the end
Role of Invariants and Preconditions

• They both serve two purposes
  § Help you think through your plans in a disciplined way
  § Communicate to the user* how they are allowed to use the class

• Provide the interface of the class
  § interface btw two programmers
  § interface btw parts of an app

• Important concept for making large software systems
  * …who might well be you!

* …who might well be you!

---

**interface** noun

1. a point where two systems, subjects, organizations, etc., meet and interact: the interface between accountancy and the law.
   - *chiefly Physics* a surface forming a common boundary between two portions of matter or space, e.g., between two immiscible liquids: the surface tension of a liquid at its air/liquid interface.

2. **Computing** a device or program enabling a user to communicate with a computer.
   - a device or program for connecting two items of hardware or software so that they can be operated jointly or communicate with each other.

—The Oxford American Dictionary
Implementing a Class

• All that remains is to fill in the methods. (All?!)  
• When implementing methods:
  1. Assume preconditions are true
  2. Assume class invariant is true to start
  3. Ensure method specification is fulfilled
  4. Ensure class invariant is true when done

• Later, when using the class:
  ▪ When calling methods, ensure preconditions are true
  ▪ If attributes are altered, ensure class invariant is true

Easy(ish) if we are the user. But what if we aren’t?
Assert Statements

assert <boolean> # Creates error if <boolean> false
assert <boolean>, <string> # As above, but displays <String>

- Way to force an error
  - Why would you do this?
- Enforce preconditions!
  - Put precondition as assert.
  - If violate precondition, the program crashes
- Provided code in A3 uses asserts heavily

```python
def exchange(amt, from_c, to_c):
    """Returns: amt from exchange
    Precondition: amt is a float...""
    assert type(amt) == float
    ...
```
def anglicize(n):

    """Returns: the anglicization of int n.

    Precondition: n an int, 0 < n < 1,000,000"""

    assert type(n) == int, str(n)+' is not an int'
    assert 0 < n and n < 1000000, str(n)+' is out of range'

    # Implement method here...

    Check (part of) the precondition

    (Optional) Error message when precondition violated
Enforce Method Preconditions with `assert`

class Time(object):
    """Instances represent times of day."""

    def __init__(self, hour, min):
        """The time hour:min.
        Pre: hour in 0..23; min in 0..59"
        assert type(hour) == int
        assert 0 <= hour and hour < 24
        assert type(min) == int
        assert 0 <= min and min < 60

    def increment(self, hours, mins):
        """Move this time <hours> hours
        and <mins> minutes into the future.
        Pre: hours is int >= 0; mins in 0..59"
        assert type(hour) == int
        assert type(min) == int
        assert hour >= 0 and
        assert 0 <= min and min < 60

Instance Attributes:
    hour: hour of day [int in 0..23]
    min: minute of hour [int in 0..59]

Initializer creates/initializes all
of the instance attributes.
Asserts in initializer guarantee the
initial values satisfy the invariant.

Asserts in other methods enforce
the method preconditions.
What About Attributes?

- User can access instance attributes via assignment
- Example:
  >>> t = Time(2,45)
  >>> t.min = 70
- Nothing we can do
  - Wrote methods assuming invariant always true
  - Our enforcement code is all in method definitions

```
class Time(object):
    """Instances represent times of day.
    Instance Attributes:
    hour: hour of day [int in 0..23]
    min: minute of hour [int in 0..59]"

    def __init__(self, hour, min):
        """The time hour:min.
        Pre: hour in 0..23; min in 0..59"
        assert type(hour) == int
        assert 0 <= hour and hour < 24
        assert type(min) == int
        assert 0 <= min and min < 60
```

Invariant violation!

Only protects inside initializer
Data Encapsulation

- **Idea**: Force the user to only use methods
- Do not allow direct access of attributes

**Setter Method**
- Used to change an attribute
- Replaces all assignment statements to the attribute
- **Bad**: `>>> t.min = 55`
- **Good**: `>>> t.setMin(55)`

**Getter Method**
- Used to access an attribute
- Replaces all usage of attribute in an expression
- **Bad**: `>>> h = 60*t.min`
- **Good**: `>>> h = 60*t.getMin()`
class Time(object):
    """Instances represent times of day.
    Instance Attributes:
    hour: hour of day [int in 0..23]
    min: minute of hour [int in 0..59]"

    def getMin(self):
        """Returns: min attribute"
        return self._min

    def setMin(self, mins):
        """Alters min attribute to be mins
        Pre: mins is in 0..59"
        assert type(mins) == int
        assert 0 <= mins and mins < 60
        self._min = mins

    Getter
    Setter

Do this for all of your attributes

Naming Convention
The underscore means “should not access the attribute directly.”

Precondition is same as attribute invariant.
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: $\frac{1}{2} \times \frac{3}{4} = \frac{3}{8}$
- Can do this with a class
  - Objects are fractions
  - Have built-in methods to implement $+$, $\times$, $/$, etc…
  - **Operator overloading**

```python
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 '{}/*{}'.format(str(self.numerator) + '/', str(self.denominator))
```

Using Classes Effectively
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

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

```python
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
        right = self.denominator * q.numerator
        return left == right
```
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)

```python
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
        right = self.denominator * q.numerator
        return left == right

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

10/6/14 Using Classes Effectively
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

```
<p>| | | |</p>
<table>
<thead>
<tr>
<th></th>
<th></th>
<th></th>
</tr>
</thead>
<tbody>
<tr>
<td>x</td>
<td>2.2</td>
<td></td>
</tr>
<tr>
<td>y</td>
<td>5.4</td>
<td></td>
</tr>
<tr>
<td>z</td>
<td>6.7</td>
<td></td>
</tr>
</tbody>
</table>
```

```
<p>| | | |</p>
<table>
<thead>
<tr>
<th></th>
<th></th>
<th></th>
</tr>
</thead>
<tbody>
<tr>
<td>x</td>
<td>2.2</td>
<td></td>
</tr>
<tr>
<td>y</td>
<td>5.4</td>
<td></td>
</tr>
<tr>
<td>z</td>
<td>6.7</td>
<td></td>
</tr>
</tbody>
</table>
```

Always use `(x is None)` not `(x == None)`
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

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

```python
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
```
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)
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
Summary + Files

• Methods with double underscores are special
  ▪ Used to implement **operators** (e.g. +, ==, <)
  ▪ Great for implementing mathematical objects
  ▪ **Example**: fraction.py

• Fields cannot enforce invariants
  ▪ Want to wrap them in **getters**, **setters**
  ▪ Setters use asserts to enforce invariants
  ▪ **Example**: betterfraction.py

• **Properties** provide invisible **getters**, **setters**
  ▪ Attributes = properties + **non-hidden** fields
  ▪ **Example**: bestfraction.py