Review 2

Classes and Subclasses
class <name>(<superclass>):

"""Class specification"""
definitions of fields
definitions of properties
constructor (__init__)
definition of operators
definition of methods

Class type to extend (may need module name)

- Every class must extend something
- Most classes will extended object
Attribute Invariants

• What are the attribute invariants below?
• Why are they there?

class Time(object):
    """An instance is a time of day"""
    hr = 0  # hour of the day; int in range 0..23
    min = 0 # minute of the hour; int in range 0..59
...
Attribute Invariants

- Attribute invariants are important for programmer
  - Can look at them when writing methods
  - Any reader of the code will benefit as well

```python
class Time(object):
    """An instance is a time of day""
    hr = 0  # hour of the day; int in range 0..23
    min = 0  # minute of the hour; int in range 0..59
    ...
```
Enforcing Invariants

- Attribute invariants are the purpose of constructors
- They initialize the attributes to satisfy invariants

```python
class Time(object):
    ...
    def __init__(self, t):
        """Constructor: an instance with time t, in minutes, in range 0..24*60-1""
        self.hr = t / 60
        self.min = t % 60
```

- Without seeing the invariants, might write `self.min = t`
Enforcing Invariants

- **Restrict access to fields**
  - Make fields hidden
  - Force access through methods: getter & setter

- **Getter**: Read attribute
  - Just return the field

- **Setter**: Change attribute
  - Checks that new value satisfies the invariant
  - If so, changes field

```python
class Time(object):
    _hr = 0  # int in range 0..23
    _min = 0  # int in range 0..59

    def gethour(self):
        """Returns: hour of the day""
        return self._hr

    def setHour(self, value):
        """Sets hour to value""
        assert type(value) == int
        assert value >= 0 and value <= 23
        self._hr = value
```
Properties: Getters and Setters

- Properties are preferred way to prevent access
  - Pair of getter and setter
  - Put invariant in getter
- Written as methods, but not called like methods

```python
>>> t.hr = 2
```

```python
class Time(object):
    _hr = 0  # int in range 0..23
...
@property
def hr(self):
    """Hour of the day
    Invariant: int in range 0..23"
    return self._hr

@hr.setter
def hr(self, value):
    assert type(value) == int
    assert value >= 0 and value <= 23
    self._hr = value
```

Python converts to

```python
>>> t.hr(2)
```
Special Methods

• Start/end with underscores
  ▪ `__init__` for constructor
  ▪ `__str__` for `str()`
  ▪ `__repr__` for backquotes

• Actually defined in `object`
  ▪ You are overriding them
  ▪ Many more of them

• 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):
        """Returns: string with contents""
        ...

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

12/2/12 Review 2
An object of class `Course` (next slide) maintains a course name, the instructors involved, and the list of registered students, sometimes called the roster.

1. State the purpose of a constructor. Then complete the body of the constructor of `Course`, fulfilling this purpose.

2. Complete the body of method `add` of `Course`.

3. Complete the body of method `__eq__` of `Course`. If you write a loop, you do not need to give a loop invariant.

4. Complete the body of method `__ne__` of `Course`. Your implementation should be a single line.
Modified Question from Fall 2010

class Course(object):
    """An instance is a course at Cornell. Maintains the name of the course, the roster (list of netIDs of students registered for it), and a list of netIDs of instructors."""
    name = ''  # Course name. Must be a String.
    instructors = None  # Must be a list of netids  # Cannot be empty.
    roster = None  # Must be a list of netids  # Allowed to be empty.

def __init__(self,name,b):
    """Instance w/ name, instructors b, no students. It must COPY b. Do not assign b to instructors. Pre: name is a string, b is a nonempty list"""
    # IMPLEMENT ME

def add(self,n):
    """If student with netID n is not in roster, add student. Do nothing if student is already there. Precondition: n is a valid netID."""
    # IMPLEMENT ME

def __eq__(self,ob):
    """Return True if ob is a Course with the same name and same set of instructors as this; otherwise return False"""
    # IMPLEMENT ME

def __ne__(self,ob):
    """Return False if ob is a Course with the same name and same set of instructors as this; otherwise return True"""
    # IMPLEMENT ME IN ONE LINE
1. State the purpose of a constructor. Complete the body of the constructor of Course, fulfilling this purpose.

- The purpose is to initialize fields so that the attribute invariants in the class are all satisfied.

```python
def __init__(self, name, b):
    """Instance w/ name, instructors b, no students.
    Pre: name is a string, b is a nonempty list""
    self.name = name
    self.instructors = b[:]  # Copies b
    self.roster = []  # Satisfy the invariant!
```
2. Complete the body of method add of Course

```python
def add(self, n):
    """If student with netID n is not in roster, add student. Do nothing if student is already there.
    Precondition: n is a valid netID.""
    if not n in self.roster:
        self.roster.append(n)
```
3. Complete body of method \_\_eq\_\_ of Course.

```python
def __eq__(self, ob):
    """Return True if ob is a Course with the same name and same set of instructors as this; otherwise return False"""
    if not (isinstance(ob, Course)):
        return False
    # Check if instructors in ob are in this
    for inst in ob.instructors:
        if not inst in self.instructors:
            return False
    # If instructors of ob are those in self, same if length is same
    return self.name == ob.name and len(self.instructors) == len(ob.instructors)
```
4. Complete body of method `__ne__` of `Course`. Your implementation should be a single line.

```python
def __ne__(self, ob):
    
    """Return False if ob is a Course with the same name and same set of instructors as this; otherwise return True""

    # IMPLEMENT ME IN ONE LINE
    return not self == ob # Calls __eq__
```
An instance of Course always has a lecture, and it may have a set of recitation or lab sections, as does CS 1110. Students register in the lecture and in a section (if there are sections). For this we have two other classes: Lecture and Section. We show only components that are of interest for this question

Do the following:

- Complete the constructor in class Section
- Complete the method add in Section

Make sure invariants are enforced at all times
class Lecture(Course):
    """Instance is a lecture, with list of sections""
    # List of sections associated with lecture.
    seclist = None # Must be a list; can be empty

    def __init__(self, n, ls):
        """Instance w/ name, instructors ls, no students.
        It must COPY ls. Do not assign ls to instructors.
        Pre: name is a string, ls is a nonempty list""
        super(Lecture,self).__init__(n,ls)
        self.seclist = []

class Section(Course):
    """Instance is a section associated w/ a lecture""
    # Lecture with which this section is associated.
    mainlecture = None # Should not be None.

    def __init__(self, n, ls, lec):
        """Instance w/ name, instructors ls, no students AND primary lecture lec.
        Pre: name a string, ls list, lec a Lecture""
        # IMPLEMENT ME

    def add(self,n):
        """If student with netID n is not in roster of section, add student to this section AND the main lecture. Do nothing if already there.
        Precondition: n is a valid netID.""
        # IMPLEMENT ME
def __init__(self, n, ls, lec):
    """Instance w/ name, instructors ls
    no students AND main lecture lec.
    Pre: name a string, ls list,
    lec a Lecture"
    super(Section, self).__init__(n, ls)
    self.mainlecture = lec

def add(self, n):
    """If student with netID n is not in
    roster of section, add student to
    this section AND the main lecture.
    Do nothing if already there.
    Precondition: n is a valid netID.""
    # Calls old version of add to
    # add to roster
    super(Section, self).add(n)
    # Add to lecture roster
    self.mainlecture.add(n)
Diagramming Subclasses

Important Details:

- Attributes should go in correct partition
- If property is overloaded, put in both partitions
- Do not need Object partition unless asked
- Methods must have parameter names
- Give parameter defaults
Example: Class Point

The object **Partition**

Provides the default constructor

**Point**

If field associated with a property, list **one** of them

**Point**

Specify non-self arguments

__class__ Point

...  
__init__() __str__()  
...  

__init__(x=0.0,y=0.0,z=0.0)  
__str__() __repr__()  
distanceTo(other)
Example: Class **Point**

Because it is always there, typically omit the object partition.
Two Classes

```python
class CongressMember(object):
    _name = "" # Member's name

    @property
    def name(self):
        return self._name

    @name.setter
    def name(self, value):
        assert type(value) == str
        self._name = value

    def __init__(self, n):
        self.name = n # Use the setter

    def __str__(self):
        return 'Honorable ' + self.name

class Senator(CongressMember):
    _state = "" # Senator's state

    @property
    def state(self):
        return self._state

    @property
    def name(self):
        return self._name

    @name.setter
    def name(self, value):
        assert type(value) == str
        self._name = 'Senator ' + value

    def __init__(self, n, s):
        assert type(s) == str and len(s) == 2
        super(Senator, self).__init__(n)
        self._state = s

    def __str__(self):
        return (super(Senator, self).__str__() +
                ' of ' + self.state)
```

12/2/12
Review 2
Execute the Following Code

```python
>>> b = CongressMember('Jack')
>>> c = Senator('John', 'NY')
>>> d = c
>>> d.name = 'Clint'
```

Remember:
Commands outside of a function definition happen in global space

- Draw two columns:
  - Global space
  - Heap space
- Draw both the
  - Variables created
  - Objects (folders) created
- Put each in right space
- If a variable changes
  - Mark out the old value
  - Write in the new value
If property overridden, same in both partitions

Note setter always puts string 'Senator' in front
class Senator(CongressMember):
    _state = "" # Senator's state

@property
def state(self):
    return self._state

@property
def name(self):
    return self._name

@name.setter
def name(self, value):
    assert type(value) == str
    self._name = 'Senator ' + value

def __init__(self, n, s):
    assert type(s) == str and len(s) == 2
    super(Senator, self).__init__(n)
    self._state = s

def __str__(self):
    return (super(Senator, self).__str__() + ' of ' + self.state)