Lecture 10

Subclasses & Inheritance
Announcements: Finishing Up

Assignment 2
- Due this evening at Midnight
  - Do as much as you can
- Graded by Wednesday
  - TAs are also on break
  - Earliest we can do
- Done if make 85 or higher
- One week to revise
  - Similar policy to A1
  - Get a final grade at end

Assignment 3
- Posted on Sunday
  - We are behind in 1110
- Will have two weeks to do
  - No more classes or labs
  - Can use Consultant hours
- Done if make 85 or higher
- Also one week to revise
  - Finished at the end

10/10/14
A Interesting Challenge

• How do we add new methods to class Fraction?
  ▪ Open up the .py module and add them!

• But Python has many “built-in” classes
  ▪ **Examples:** string, list, time, date (in datetime)
  ▪ **GUI Examples:** Button, Slider, Image

• What if we want to add methods to these?
  ▪ Where is the module to modify?
  ▪ It is even a good idea to modify it?
An Application

- **Goal**: Presentation program (e.g. PowerPoint)
- **Problem**: There are many types of content
  - **Examples**: text box, rectangle, image, etc.
  - Have to write code to display each one
- **Solution**: Use object oriented features
  - Define class for every type of content
  - Make sure each has a `draw` method:
    ```python
    for x in slide[i].contents:
        x.draw(window)
    ```
Sharing Work

• These classes will have a lot in common
  ▪ Drawing handles for selection
  ▪ Background and foreground color
  ▪ Current size and position
  ▪ And more (see the formatting bar in PowerPoint)

• **Result**: A lot of repetitive code

• **Solution**: Create one class with shared code
  ▪ All content are *subclasses* of the *parent* class
Defining a Subclass

Abbreviate as SC to right

class SlideContent(object):
    """Any object on a slide."""
    def __init__(self, x, y, w, h): ...
    def draw_frame(self): ...
    def select(self): ...

class TextBox(SlideContent):
    """An object containing text."""
    def __init__(self, x, y, text): ...
    def draw(self): ...

class Image(SlideContent):
    """An image."""
    def __init__(self, x, y, image_file): ...
    def draw(self): ...
Class Definition: Revisited

class <name>(<superclass>):

"""Class specification"""
getters and setters
initializer (__init__)
definition of operators
definition of methods
anything else

Class type to extend (may need module name)

- Every class must extend *something*
- Previous classes all extended *object*
object and the Subclass Hierarchy

- Subclassing creates a **hierarchy** of classes
  - Each class has its own super class or parent
  - Until object at the “top”
- object has many features
  - Special built-in fields: `__class__`, `__dict__`
  - Default operators: `__str__`, `__repr__`

**Kivy Example**

- object
- `kivy.uix.widget.WidgetBase`
- `kivy.uix.widget.Widget`
- `kivy.uix.label.Label`
- `kivy.uix.button.Button`

**Module**

**Class**
Subclassing creates a **hierarchy** of classes

- Each class has its own super class or parent
- Until object at the “top”

Object has many features

- Special built-in fields: __class__, __dict__
- Default operators: __str__, __repr__

### Kivy Example

- object
- kivy.uix.widge.WidgetBase
- kivy.uix.widget.Widget
- kivy.uix.label.Label
- kivy.uix.button.Button

---

**Built-in class**

- object
- kivy.uix.widge.WidgetBase
- kivy.uix.widget.Widget
- kivy.uix.label.Label
- kivy.uix.button.Button

**Module**

- kivy

**Class**

- WidgetBase
- Widget
- Label
- Button
Name Resolution Revisited

To look up attribute/method name:
1. Look first in instance (object folder)
2. Then look in the class (folder)

Subclasses add two more rules:
3. Look in the superclass
4. Repeat 3. until reach object

```python
class TextBox:
    def __init__(self, x, y, text):
        self.text = text
    def draw(self):
        pass

class SC:
    def __init__(self, x, y, w, h):
        pass
    def draw_frame(self):
        pass
    def select(self):
        pass

object = SC(object)
```

```
p = id3
id3 = TextBox('Hi!')
```
Name Resolution Revisited

- To look up attribute/method name
  1. Look first in instance (object folder)
  2. Then look in the class (folder)
- Subclasses add two more rules:
  3. Look in the superclass
  4. Repeat 3. until reach object

```
TextBox
  p.text
  p.draw()
  p.select()

TextBox(SC)
  __init__(x,y,text)
  draw()

SC(object)
  __init__(x,y,w,h)
  draw_frame()
  select()

object
```
Name Resolution Revisited

• To look up attribute/method name
  1. Look first in instance (object folder)
  2. Then look in the class (folder)
• Subclasses add:
  3. Look in the superclass
  4. Repeat 3. up the hierarchy

Often Called the **Bottom-Up Rule**.
Subclass *inherits* methods of parent.
A Simpler Example

class Employee(object):
    """Instance is salaried worker
    INSTANCE ATTRIBUTES:
    name:  full name [string]
    start:  first year hired
        [int ≥ -1, -1 if unknown]
    salary: yearly wage [float]"

class Executive(Employee):
    """An Employee with a bonus
    INSTANCE ATTRIBUTES:
    bonus: annual bonus [float]"

10/10/14 Subclasses & Inheritance
class Employee(object):
    """Instance is salaried worker
    INSTANCE ATTRIBUTES:
    name: full name [string]
    start: first year hired
    [int $\geq$ -1, -1 if unknown]
    salary: yearly wage [float]"

class Executive(Employee):
    """An Employee with a bonus
    INSTANCE ATTRIBUTES:
    bonus: annual bonus [float]"

object
__init__()
__str__()
__eq__()

Employee
__init__(n,d,s)
__str__()
__eq__()

Executive
__init__(n,d,b)
__str__()
__eq__()
Method Overriding

• Which `__str__` do we use?
  - Start at bottom class folder
  - Find first method with name
  - Use that definition

• New method definitions **override** those of parent

• Also applies to
  - Initializers
  - Operators
  - Properties

```
object
__init__()
__str__()
__eq__()

Employee
__init__(n,d,s)
__str__()
__eq__()

Executive
__init__(n,d,b)
__str__()
__eq__()
```
Accessing the “Previous” Method

- What if you want to use the original version method?
  - New method = `original + more`
  - Do not want to repeat code from the original version
- Call old method `explicitly`
  - Use method as a function
  - Pass object as first argument
- **Example:**
  ```
  Employee.__str__(self)
  ```
- **Cannot do with properties**
Accessing the “Previous” Method

- What if you want to use the original version method?
  - New method = original + more
  - Do not want to repeat code from the original version

- Call old method explicitly
  - Use method as a function
  - Pass object as first argument

- Example:
  Employee.__str__(self)

- Cannot do with properties

```python
class Employee(object):
    '''An Employee with a salary'''
    ...
    def __str__(self):
        return (self.name +
                ', year ' + str(self.start) +
                ', salary ' + str(self.salary))

class Executive(Employee):
    '''An Employee with a bonus.''
    ...
    def __str__(self):
        return (Employee.__str__(self) +
                ', bonus ' + str(self.bonus))
```
class Employee(object):
    ...
    def __init__(self, n, d, s=50000.0):
        self._name = n
        self._start = d
        self._salary = s

class Executive(Employee):
    ...
    def __init__(self, n, d, b=0.0):
        Executive.__init__(self, n, d, b)
        self._bonus = b
Instance Attributes are (Often) Inherited

```python
class Employee(object):
    ...
    def __init__(self, n, d, s=50000.0):
        self._name = n
        self._start = d
        self._salary = s

class Executive(Employee):
    ...
    def __init__(self, n, d, b=0.0):
        Executive.__init__(self, n, d)
        self._bonus = b
```
Class Attribute: Assigned outside of any method definition

class Employee(object):
    """Instance is salaried worker"""
    # Class Attribute
    STD_SALARY = 50000.0

class Executive(Employee):
    """An Employee with a bonus."""
    # Class Attribute
    STD_BONUS = 10000.0
Name Resolution and Inheritance

class A(object):
    x = 3  # Class Attribute
    y = 5  # Class Attribute
    def f(self):
        return self.g()
    def g(self):
        return 10

class B(A):
    y = 4  # Class Attribute
    z = 42  # Class Attribute
    def g(self):
        return 14
    def h(self):
        return 18

• Execute the following:
  >>> a = A()
  >>> b = B()

• What is value of a.f()?

A: 10
B: 14
C: 5
D: ERROR
E: I don’t know
Name Resolution and Inheritance

```python
class A(object):
    x = 3  # Class Attribute
    y = 5  # Class Attribute

def f(self):
    return self.g()

def g(self):
    return 10

class B(A):
    y = 4  # Class Attribute
    z = 42  # Class Attribute

def g(self):
    return 14

def h(self):
    return 18
```

- Execute the following:
  ```python
  >>> a = A()
  >>> b = B()
  ```
- What is value of `a.f()`?
  
  | A: 10  | CORRECT |
  | B: 14  |
  | C: 5   |
  | D: ERROR |
  | E: I don’t know |

10/10/14  
Subclasses & Inheritance
Name Resolution and Inheritance

class A(object):
    x = 3  # Class Attribute
    y = 5  # Class Attribute

def f(self):
    return self.g()

def g(self):
    return 10

class B(A):
    y = 4  # Class Attribute
    z = 42  # Class Attribute

def g(self):
    return 14

def h(self):
    return 18

- Execute the following:
  >>> a = A()
  >>> b = B()

- What is value of b.f()?

A: 10  
B: 14  
C: 5  
D: ERROR
E: I don’t know
Name Resolution and Inheritance

class A(object):
    x = 3  # Class Attribute
    y = 5  # Class Attribute

def f(self):
    return self.g()
def g(self):
    return 10

class B(A):
    y = 4  # Class Attribute
    z = 42  # Class Attribute

def g(self):
    return 14
def h(self):
    return 18

• Execute the following:
  >>> a = A()
  >>> b = B()

• What is value of b.f()?

A: 10
B: 14  CORRECT
C: 5
D: ERROR
E: I don’t know
Name Resolution and Inheritance

```python
class A(object):
    x = 3  # Class Attribute
    y = 5  # Class Attribute

def f(self):
    return self.g()

def g(self):
    return 10

class B(A):
    y = 4  # Class Attribute
    z = 42 # Class Attribute

    def g(self):
        return 14

    def h(self):
        return 18
```

- Execute the following:
  ```
  >>> a = A()
  >>> b = B()
  ```

- What is value of `b.x`?

  A: 4  
  B: 3  
  C: 42  
  D: ERROR  
  E: I don’t know
Name Resolution and Inheritance

class A(object):
    x = 3  # Class Attribute
    y = 5  # Class Attribute

def f(self):
    return self.g()

def g(self):
    return 10

class B(A):
    y = 4  # Class Attribute
    z = 42  # Class Attribute

def g(self):
    return 14

def h(self):
    return 18

• Execute the following:
    >>> a = A()
    >>> b = B()

• What is value of b.x?

A: 4
B: 3  CORRECT
C: 42
D: ERROR
E: I don’t know
Name Resolution and Inheritance

```python
class A(object):
    x = 3  # Class Attribute
    y = 5  # Class Attribute

def f(self):
    return self.g()

def g(self):
    return 10

class B(A):
    y = 4  # Class Attribute
    z = 42  # Class Attribute

    def g(self):
        return 14

def h(self):
    return 18
```

- Execute the following:
  ```
  >>> a = A()
  >>> b = B()
  ```

- What is value of `a.z`?

  A: 4  
  B: 3  
  C: 42  
  D: ERROR  
  E: I don’t know
Name Resolution and Inheritance

```python
class A(object):
    x = 3  # Class Attribute
    y = 5  # Class Attribute

def f(self):
    return self.g()

def g(self):
    return 10

class B(A):
    y = 4  # Class Attribute
    z = 42  # Class Attribute

def g(self):
    return 14

def h(self):
    return 18
```

- Execute the following:
  ```
  >>> a = A()
  >>> b = B()
  ```

- What is value of `a.z`?

  A: 4  
  B: 3  
  C: 42 
  D: ERROR  
  E: I don’t know  

CORRECT
Properties and Inheritance

• Properties: all or nothing
  ▪ Typically inherited
  ▪ Or fully overridden (both getter and setter)
• When override property, **completely** replace it
  ▪ Cannot use super()
• Very rarely overridden
  ▪ **Exception**: making a property read-only
  ▪ See employee2.py

```python
class Employee(object):
    ...
    @property
def salary(self):
        return self._salary
    @salary.setter
def salary(self, value):
        self._salary = value

class Executive(Employee):
    ...
    @property  # no setter; now read-only
def salary(self):
        return self._salary
```