Lecture 9

Subclasses & Inheritance
A Interesting Challenge

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

• But Python has many “built-in” classes
  ▪ **Examples**: string, list, time, date (in datetime)
  ▪ **Kivy 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?
Program design

• **Goal**: A 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

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): ...

Abbreviate as SC to right

10/23/12 Subclasses & Inheritance
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*
- 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 implementations of operators (e.g. `__str__`)

---

**Kivy Example**

- `object`
- `kivy.event.EventDispatcher`
- `kivy.uix.widget.Widget`
- `kivy.uix.label.Label`
- `kivy.uix.button.Button`
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 implementations of operators (e.g. __str__)
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. up the hierarchy

10/23/12 Subclasses & Inheritance
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. up the hierarchy
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. up the hierarchy

Often Called the **Bottom-Up Rule**
class Employee(object):
    
    """An Employee with a salary
    Attributes:
    name [string]: Employee name
    start [int, -1...]: Starting year
    -1 if undefined
    salary [float]: yearly wage"

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

object
    __init__()
    __str__()
    __eq__()

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

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

• Which \_\_str\_\_ does \_\_str\_\_ use?
  ▪ Start at bottom class folder
  ▪ Find first method name
  ▪ Use that definition

• New method definitions \textbf{override} those of parent

• Also applies to
  ▪ Constructor
  ▪ Operators \{ all "methods"
  ▪ 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 definition of the overridden method?
  ▪ New method just *extends*
  ▪ Do not want to repeat code from the old version

• Call old method **explicitly**
  ▪ Use method as a function
  ▪ Pass object as first argument

• **Example:**
  Employee.__str__(self)

• Doesn’t work on properties
Accessing the “Previous” Method

• What if you want definition of the overridden method?
  ▪ New method just *extends*
  ▪ Do not want to repeat code from the old version

• Call old method *explicitly*
  ▪ Use method as a function
  ▪ Pass object as first argument

• **Example:**
  Employee.__str__(self)

• Doesn’t work on 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)
        self._bonus = b