Announcements for Today

Reading

- Today: Chapter 18
- Online reading for Thursday
- **Prelim, Nov 12th 7:30-9:00**
  - Material up to Thursday
  - Review posted on Thursday
  - Recursion + Loops + Classes
- **S/U Students are exempt**
- **Conflict with Prelim time?**
  - Prelim 2 Conflict on CMS
  - Submit by Thursday

Assignments

- A4 graded by end of week
  - Survey is still open
- A5 was posted Friday
  - Shorter written assignment
  - Due Thursday at Midnight
- A6 also posted Friday
  - Due a **week after** prelim
  - Designed to take two weeks
  - Finish first part before exam

An Application

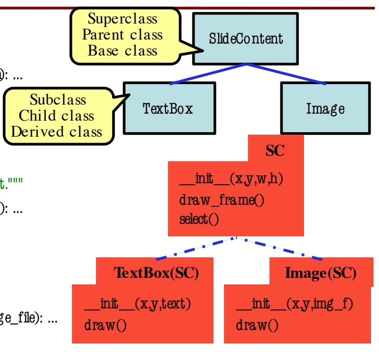
- **Goal**: Presentation program (e.g. PowerPoint)
- **Problem**: There are many types of content
  - **Examples**: text box, rectangle, image, etc.
  - Have to write code to display each one
- **Solution**: Use object oriented features
  - Define class for every type of content
  - Make sure each has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

```
for x in slide[i].contents:
    x.draw(window)
```

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
```

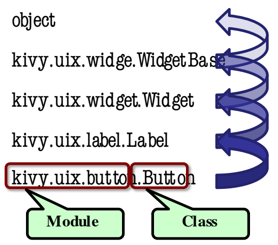
Class type to extend (may need module name)

- Every class must extend *something*
- Previous classes all extended object

object and the Subclass Hierarcy

- 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__

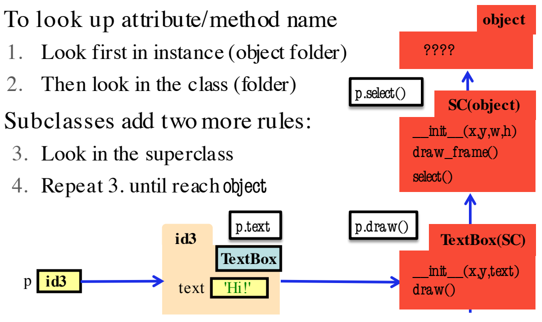
Kivy Example



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

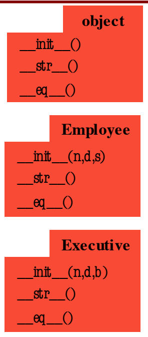
Name Resolution Revisited

- 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



A Simpler Example

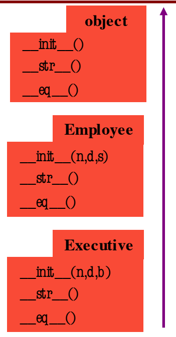
```
class Employee(object):
    """Instance is salaried worker
    INSTANCE ATTRIBUTES:
        name  [string]:  full name
        start  [int ≥ -1, -1 if unknown]:
                first year hired
        salary [float]: yearly wage"""

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



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

  all "methods"



Accessing the "Previous" Method

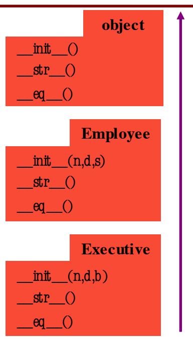
- What if you want to use the original version method?
  - New method = original+more
  - Do not want to repeat code from the original version
- Call old method **explicitly**
  - Use method as a function
  - Pass object as first argument
- **Example**:
  Employee.__str__(self)
- **Cannot do with properties**

```
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) )
```

Primary Application: Initializers

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

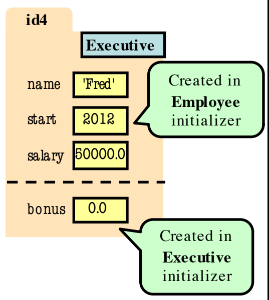
```
class Executive(Employee):
    ...
    def __init__(self,n,d,b=0.0):
        Employee.__init__(self,n,d)
        self._bonus = b
```



Instance Attributes are (Often) Inherited

```
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):
        Employee.__init__(self,n,d)
        self._bonus = b
```

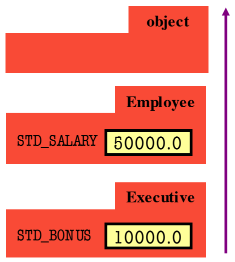

id4
Executive
name
'Fred'
start
2012
salary
50000.0
bonus
0.0
Created in Employee initializer
Created in Executive initializer

Also Works With Class Attributes

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
```


object
Employee
STD_SALARY
50000.0
Executive
STD_BONUS
10000.0