Lecture 19:
Subclasses & Inheritance
(Chapter 18)
CS 1110
Introduction to Computing Using Python

http://www.cs.cornell.edu/courses/cs1110/2018sp

[E. Andersen, A. Bracy, D. Gries, L. Lee, S. Marschner, C. Van Loan, W. White]
Announcements

• More recursion examples on the Lectures page
• A3 is being graded this week
• A4 coming soon!
• Prelim 2
  ▪ Tuesday, April 24th, 7:30-9:00pm
  ▪ Please go to the same room you went for Prelim 1
  ▪ Conflicts assignment on CMS, due 11:59pm Thurs.
• Lab 11 is out (there is no Lab 10)
Goal: Make a drawing app

Rectangles, Stars, Circles, and Triangles have a lot in common, but they are also different in very fundamental ways....
Sharing Work

**Problem:** Redundant code.

(Any time you copy-and-paste code, you are likely doing something wrong.)

**Solution:** Create a *parent* class with shared code

- Then, create *subclasses* of the *parent* class
Defining a Subclass

class Shape():
    """A shape located at x,y """
    def __init__(self, x, y): ...
    def draw(self): ...

class Circle(Shape):
    """An instance is a circle."""
    def __init__(self, x, y, radius): ...
    def draw(self): ...

class Rectangle(Shape):
    """An instance is a rectangle."""
    def __init__(self, x, y, ht, len): ...
    def draw(self): ...
Extending Classes

```python
class <name>(<superclass>):
    """Class specification""
    class variables
    initializer (__init__)
    methods
```

- Class to extend (may need module name)
- So far, classes have implicitly 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
  - Default operators: 
    ```
    __init__, __str__, 
    __repr__, __eq__
    ```

Which of these need to be replaced?
```python
class Shape():
    '''Instance is shape @ x,y'''
    def __init__(self, x, y):
        self.x = x
        self.y = y

class Circle(Shape):
    '''Instance is a Circle @ x,y with radius'''
    def __init__(self, x, y, radius):
        self.radius = radius
        super().__init__(x, y)
```

- Want to use the original version of the method?
  - New method = original + more
  - Do not want to repeat code from the original version
- Call old method **explicitly**
Object Attributes can be Inherited

class Shape():
    """Instance is shape @ x,y""
    def __init__(self, x, y):
        self.x = x
        self.y = y

class Circle(Shape):
    """Instance is a Circle @ x,y with radius""
    def __init__(self, x, y, radius):
        self.radius = radius
        super().__init__(x, y)

cl = Circle(1, 2, 4.0)
class Shape():
    """Instance is shape @ x,y""
    def __init__(self, x, y):
    def __str__(self):
        return "Shape @ ("+str(self.x)+", "+str(self.y)+")"
    def draw(self):...

class Circle(Shape):
    """Instance is a Circle @ x,y with radius""
    def __init__(self, x, y, radius):
    def __str__(self):
        return "Circle: Radius=\""+str(self.radius)+\" \"+Shape.__str__(self)
    def draw(self):...
c1 = Circle(1,2,4.0)
print(str(c1))

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

• Each subclass automatically inherits methods of parent.

• New method definitions override those of parent.
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

Often called the **Bottom–Up Rule**

c1 = Circle(1, 2, 4.0)

r = c1.radius

c1.draw()
Q1: Name Resolution and Inheritance

```python
class A():
    def f(self):
        return self.g()
    def g(self):
        return 10

class B(A):
    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
  - B: 14
  - C: 5
  - D: ERROR
  - E: I don’t know
A1: Name Resolution and Inheritance

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

class B(A):
    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  CORRECT
B: 14
C: 5
D: ERROR
E: I don’t know
Q2: Name Resolution and Inheritance

```
class A():
    def f(self):
        return self.g()
    def g(self):
        return 10

class B(A):
    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 |
A2: Name Resolution and Inheritance

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

class B(A):
    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
Accessing the “Original” draw

```python
class Shape():
    def draw(self):
        turtle.penup()
        turtle.setx(self.x)
        turtle.sety(self.y)
        turtle.pendown()

class Circle(Shape):
    def draw(self):
        super().draw()
        turtle.circle(self.radius)
```

Note: we’ve imported the `turtle` module which allows us to move a pen on a 2D grid and draw shapes.

No matter the shape, we want to pick up the pen, move to the location of the shape, put the pen down. Only the shape subclasses know how to do the actual drawing, though.
class Shape(object):
    """Instance is shape @ x,y"""
    # Class Attribute
    NUM_SHAPES = 0
    ...

class Circle(Shape):
    """Instance is a Circle @ x,y with radius"""
    # Class Attribute
    NUM_CIRCLES = 0
    ...
Q3: Name Resolution and Inheritance

```
class A():
    x = 3  # Class Variable
    y = 5  # Class Variable

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

    def g(self):
        return 10

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

    def g(self):
        return 14

    def h(self):
        return 18
```

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

- What is value of `b.x`?
  
<table>
<thead>
<tr>
<th>Answer</th>
<th>Value</th>
</tr>
</thead>
<tbody>
<tr>
<td>A</td>
<td>4</td>
</tr>
<tr>
<td>B</td>
<td>3</td>
</tr>
<tr>
<td>C</td>
<td>42</td>
</tr>
<tr>
<td>D</td>
<td>ERROR</td>
</tr>
<tr>
<td>E</td>
<td>I don’t know</td>
</tr>
</tbody>
</table>


A3: Name Resolution and Inheritance

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

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

    def g(self):
        return 10

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

    def g(self):
        return 14

    def h(self):
        return 18
```

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

- What is value of `b.x`?

  ```
  A: 4
  B: 3  CORRECT
  C: 42
  D: ERROR
  E: I don’t know
  ```
Q4: Name Resolution and Inheritance

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

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

    def g(self):
        return 10

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

    def g(self):
        return 14

    def h(self):
        return 18

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

• What is value of \texttt{a.z}?

  \begin{itemize}
  \item A: 4
  \item B: 3
  \item C: 42
  \item D: ERROR
  \item E: I don’t know
  \end{itemize}
A4: Name Resolution and Inheritance

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

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

    def g(self):
        return 10

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

    def g(self):
        return 14

    def h(self):
        return 18
```

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

- What is value of `a.z`?
  
  A: 4
  B: 3
  C: 42
  D: ERROR  CORRECT
  E: I don’t know
Why override `__eq__`?

class Shape():
    """Instance is shape @ x,y"""
    def __init__(self,x,y):
    def __eq__(self, other):
        """If position is the same, then equal as far as Shape knows""
        return self.x == other.x and self.y == other.y

class Circle(Shape):
    """Instance is a Circle @ x,y with radius""
    def __init__(self,x,y,radius):
    def __eq__(self, other):
        """If radii are equal, let super do the rest""
        return self.radius == other.radius and super().__eq__(other)
eq vs. is

== compares equality
is compares identity

c1 = Circle(1, 1, 25)
c2 = Circle(1, 1, 25)
c3 = c2

c1 == c2 ?
c1 is c2 ?
c2 == c3 ?
c2 is c3 ?
eq vs. is

== compares equality
is compares identity

c1 = Circle(1, 1, 25)
c2 = Circle(1, 1, 25)
c3 = c2

c1 == c2 ? True
c1 is c2 ? False
c2 == c3 ? True
c2 is c3 ? True
The `isinstance` Function

`isinstance(<obj>,<class>)`

- True if `<obj>`’s class is same as or a subclass of `<class>`
- False otherwise

**Example:**

```python
c1 = Circle(1,2,4.0)
```

- `isinstance(c1,Circle)` is True
- `isinstance(c1,Shape)` is True
- `isinstance(c1,object)` is True
- `isinstance(c1,str)` is False

- Generally preferable to `type`
  - Works with base types too!
Q5: `isinstance` and Subclasses

```python
>>> shape1 = Rectangle(0,0,10,10)
>>> isinstance(shape1, Square)
```

???

A: True  
B: False  
C: Error  
D: I don’t know
A5: `isinstance` and Subclasses

```python
>>> shape1 = Rectangle(0,0,10,10)
>>> isinstance(shape1, Square)
???
```

<table>
<thead>
<tr>
<th>A: True</th>
<th>B: False</th>
<th>C: Error</th>
<th>D: I don’t know</th>
</tr>
</thead>
</table>

A: True  
B: False  **CORRECT**
C: Error
D: I don’t know