Lecture 22: Subclasses & Inheritance (Chapter 18)

CS 1110

Introduction to Computing Using Python

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

• No new lab exercises this week. Lab sections cancelled but there’ll be extra office hours. *Good opportunity to go over A4 if you have any questions.* (Hours are listed in the office hr calendar):
  ▪ Tues 1:15-2:30pm (Jonathan C.)
  ▪ Wedn 10:10-11am (Priya M.)

• Prelim 2: we expect feedback to be available on Monday

• Assignment 5: expected release tonight (Tues)
Topics

• Why define subclasses?
  ▪ Understand the resulting hierarchy
  ▪ Design considerations

• How to define a subclass
  ▪ Initializer
  ▪ New methods
  ▪ Write modified versions of inherited methods
  ▪ Access parent’s version using super()
Goal: Make a drawing app

Rectangles, Stars, Circles, and Triangles have a lot in common, but they are also different in very fundamental ways....
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
- A subclass deals with specific details different from 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

class <name>(<superclass>):

"""Class specification"""

<class variables>

<initializer>

<methods>

Class to extend
(may need module name:
<modulename>.<superclass>)

So far, classes have implicitly extended object
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__`, `__eq__`

Which of these need to be replaced?
**init**: write new one, access parent’s

```python
class Shape:
    
    """A shape @ location x,y """
    def __init__(self, x, y):
        self.x = x
        self.y = y

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

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

class Shape:

    """A shape @ location x,y """
    def __init__(self, x, y):
        self.x = x
        self.y = y

class Circle(Shape):

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

c1 = Circle(1, 2, 4.0)
Can override methods; can access parent’s version

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) + " " + super().__str__()
Understanding Method Overriding

c1 = Circle(1,2,4.0)
print(str(c1))

• Which \texttt{\_\_str\_\_} do we use?
  ▪ Start at bottom class folder
  ▪ Find first method with name
  ▪ Use that definition
• Each subclass automatically \textit{inherits} methods of parent.
• New method definitions \textit{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 \texttt{object}

Often called the \textbf{Bottom–Up Rule}

\begin{verbatim}
c1 = Circle(1,2,4.0)
r = c1.radius
c1.draw()
\end{verbatim}
Q1: 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  
  B: 14  
  C: 5  
  D: ERROR  
  E: I don’t know
Q2: Name Resolution and Inheritance

**class A:**

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

def g(self):
    return 10
```

**class B(A):**

```python
def g(self):
    return 14

def h(self):
    return 18
```

- Execute the following:
  ```python
  >>> a = A()
  >>> b = B()
  ```
- What is value of `b.f()`?

  - A: 10
  - B: 14
  - C: 5
  - D: ERROR
  - E: *I don’t know*
Demo using Turtle Graphics

A turtle holds a pen and can draw as it walks! Follows simples commands:

- `setx`, `sety` – set start coordinate
- `pendown`, `penup` – control whether to draw when moving
- `forward`
- `turn`

Part of the turtle module in Python ([docs.python.org/3.7/library/turtle.html](docs.python.org/3.7/library/turtle.html))

- *You don’t need to know it*
- Just a demo to explain design choices of `draw()` in our classes `Shape`, `Circle`, `Rectangle`, `Square`
Who draws what?

```python
class Shape:
    """Moves pen to correct location""
    def draw(self):
        turtle.penup()
        turtle.setx(self.x)
        turtle.sety(self.y)
        turtle.pendown()

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

Note: need to import 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. But only the shape subclasses know how to do the actual drawing.

See `shapes_v3.py`, `draw_shapes.py`