Lecture 18:
Subclasses & Inheritance
(Chapter 18)

CS 1110

Introduction to Computing Using Python

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

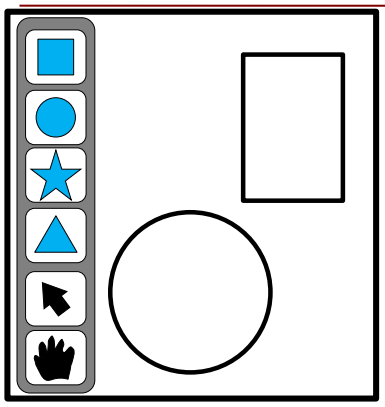
Announcements

- Prelim 2 is next week!

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….

See shapes_v0.py

Sharing Work

Problem: Redundant code.

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

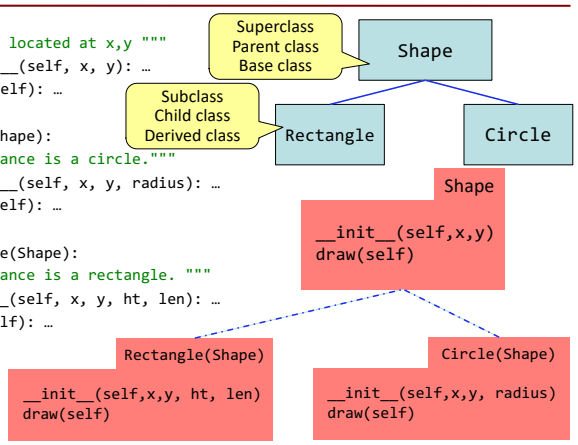
Solution: Create a *parent* class with shared code

- Then, create *subclasses* of the *parent* class
- A subclass deals with specific details different from the parent class

See shapes_v1.py

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 in stance 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>`)

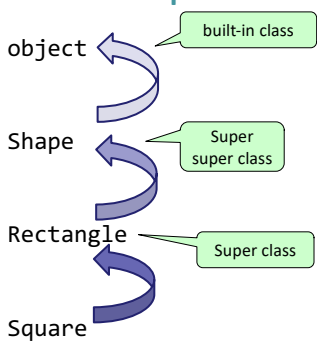
So far, classes have implicitly extended `object`

`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__`, `__eq__`

Which of these need to be replaced?

Example



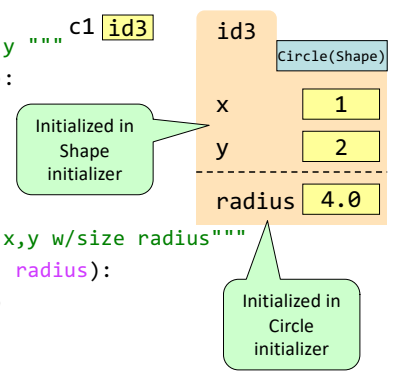
`__init__`: write new one, access parent's

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

class Circle(Shape):
  """Instance is Circle @ x,y w/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 Circle @ x,y w/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__()
  def draw(self):…
```

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

Shape
`__init__(self,x,y)`
`__str__(self)`

Circle
`__init__(self,x,y,radius)`
`__str__(self)`

Why override `__eq__` ? Compare equality

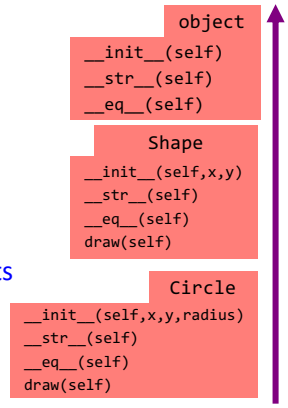
```
class Shape:
   """Instance is shape @ x,y"""
   def __init__(self,x,y):
   def __eq__(self, other):
   """If position is the same, then equal as far as Shape knows"""
      return self.x == other.x and self.y == other.y

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)
```

Want to compare equality of the values (data) of two instances, not the id of the two instances!

Understanding Method Overriding

```
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.

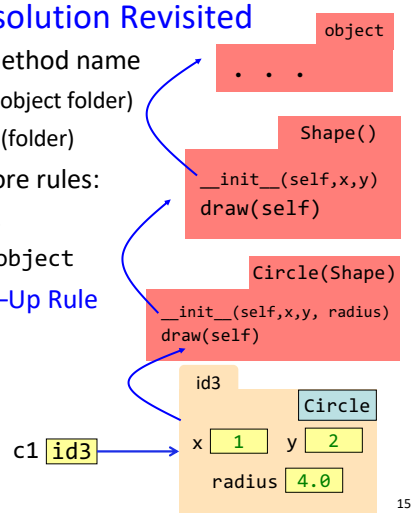


Name Resolution Revisited

- 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()
```



The following questions will be addressed in the lecture that follows.

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:

    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