Class-based model

- Have classes that describe the format of objects
- Create objects by stating the class of the object to be created.
- The created object is called an instance of the class

Class-based model

- In a class based model, the class is sometimes an object too (as is the case in Python)
- Q: what is the class of the class object?

Class-based model

- In a class based model, the class is sometimes an object too (as is the case in Python)
- Q: what is the class of the class object?
  - The “meta-class”? But then do we have a meta-meta-class?
  - many possibilities, but no clear answer
  - turns out to be a nasty problem!

What’s the alternative?

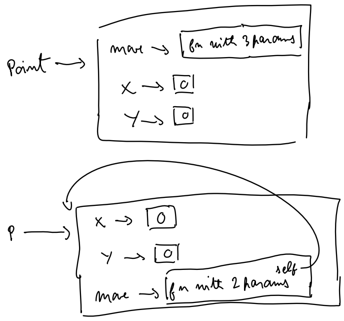
- Suppose we didn’t have classes
- How would one survive?

Prototype-based models

- Just have objects
  - Create a new object by cloning another one
  - Add/update fields later
- Benefits:
  - Simplifies the definition of the language
  - Avoids meta-class problem
- Drawbacks:
  - Don’t have classes for static typing
  - Some find the model harder to grock
- Python has hints of a prototype-based language. Go back to code

Methods

Methods



Structural, nominal subtyping

```
class Point:
  x = 0
  y = 0
  def move(self,dx,dy):
    self.x = self.x + dx
    self.y = self.y + dy

p = Point()
```

```
class Point2:
  x = 0
  y = 0
  def move(self,dx,dy):
    self.x = self.x + dx
    self.y = self.y + dy

q = Point2()
```

- p and q of the same type?
  - In Java, no: nominal subtyping (using names of classes to determine subtyping)
  - In Python, yes: structural subtyping (using fields/methods to determine subtyping)

Next: constructors

- Go back to code

Inheritance

- Key concept of OO languages

- Someone tell me what inheritance is?

Inheritance

- Key concept of OO languages

- Someone tell me what inheritance is?
- isa “concept”

- Examples?

Examples of inheritance

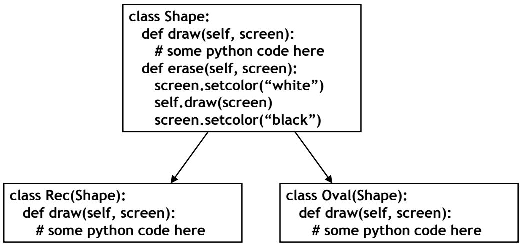
Overriding

- Super-class method can be overwritten in sub-class
- Polymorphism
  - external clients can write code that handles many different kinds of objects in the same way
  - don’t care about implementation details: as long as the object knows to draw itself, that’s good enough

Polymorphism, continued

- Super-class can have methods that are not overridden, but that work differently for different sub-classes
- For example: super-class method functionality changes because the super-class calls a method that gets overwritten in the sub-class

Simple example



```
class Shape:
  def draw(self, screen):
    # some python code here
  def erase(self, screen):
    screen.setcolor(“white”)
    self.draw(screen)
    screen.setcolor(“black”)
```

```
class Rec(Shape):
  def draw(self, screen):
    # some python code here
```

```
class Oval(Shape):
  def draw(self, screen):
    # some python code here
```

Stepping away from Python

- What are the fundamental issues with inheritance?

Stepping away from Python

- What are the fundamental issues with inheritance?
- Dispatch mechanism
  - most compilers use v-tables
  - more complicated with multi-methods
- Overloading vs. overriding
  - what’s the difference?
- How to decide on the inheritance graph?
  - not always obvious, see next example

Rectangle and Square

```
class Rectangle:
  length = 0
  width = 0
  def area(this):
    return  this.length *
            this.width
```

```
class Square:
  length = 0
  def area(this):
    return  this.length *
            this.length
```

- Which should be a sub-class of which?

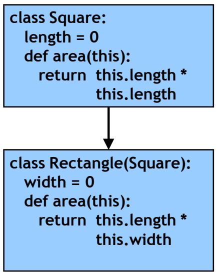
Rectangle and Square

```
class Rectangle:
   length = 0
   width = 0
   def area(this):
      return  this.length *
                  this.width
```

```
class Square:
   length = 0
   def area(this):
      return  this.length *
                  this.length
```

- Which should be a sub-class of which?
- Answer is not clear...

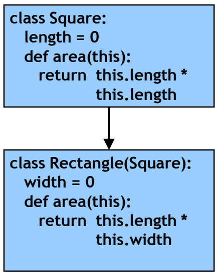
Option 1: Rectangle isa Square



```
class Square:
   length = 0
   def area(this):
      return  this.length *
                  this.length
```

```
class Rectangle(Square):
   width = 0
   def area(this):
      return  this.length *
                  this.width
```

Option 1: Rectangle isa Square



```
class Square:
   length = 0
   def area(this):
      return  this.length *
                  this.length
```

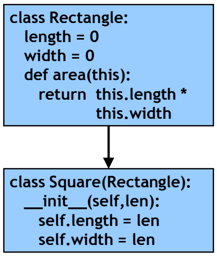
```
class Rectangle(Square):
   width = 0
   def area(this):
      return  this.length *
                  this.width
```

+ Store only what is needed (one field for square)

— Does not follow “isa” relationship from math (rectangle is not a square...)

— Have to override area method

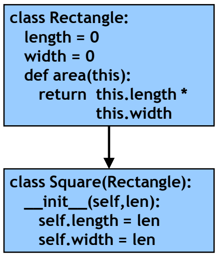
Option 2: Square isa Recangle



```
class Rectangle:
   length = 0
   width = 0
   def area(this):
      return  this.length *
                  this.width
```

```
class Square(Rectangle):
   __init__(self,len):
      self.length = len
      self.width = len
```

Option 2: Square isa Recangle



```
class Rectangle:
   length = 0
   width = 0
   def area(this):
      return  this.length *
                  this.width
```

```
class Square(Rectangle):
   __init__(self,len):
      self.length = len
      self.width = len
```

+ Follows isa relationship from math

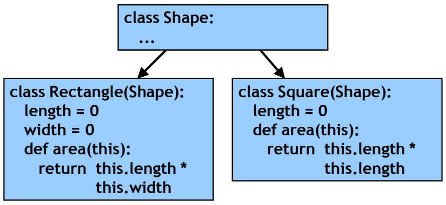
+ Don’t need to write two area methods

— Can’t enfore invariant that length=width

— Use two fields for Square (len and width)

But, does it matter? Performance is a tricky matter. Often better to implement first, then use profiler to find where bottlenecks are...

Option 3:

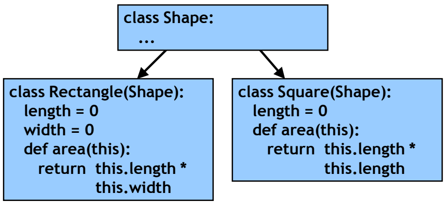


```
class Shape:
   ...
```

```
class Rectangle(Shape):
   length = 0
   width = 0
   def area(this):
      return  this.length *
                  this.width
```

```
class Square(Shape):
   length = 0
   def area(this):
      return  this.length *
                  this.length
```

Option 3:



```
class Shape:
  ...
```

```
class Rectangle(Shape):
  length = 0
  width = 0
  def area(this):
    return  this.length *
            this.width
```

```
class Square(Shape):
  length = 0
  def area(this):
    return  this.length *
            this.length
```

+ Store only what is needed (one field for square)

— Does not follow “isa” relationship from math (rectangle is not a square...)

— Have to write two area methods

Complex numbers

```
class Real:
  RealPart = 0
```

```
class Complex:
  RealPart = 0
  ComplexPart = 0
```

The same exact options present themselves here, with the same tradeoffs!

Summary of (single) inheritance

- Inheritance is a powerful mechanism
- From the programmer’s perspective, difficulty is in defining the inheritance diagram
- From a language implementer’s perspective, difficulty is in making dynamic dispatch work

Multiple inheritance

```
class ColorTextBox(ColorBox,TextPoint):
  def draw(self,screen,pos):
    ColorBox.draw(self,screen,pos)
    r=TextPoint.draw(self,screen,pos)
    return r
  def __str__(self):
    return ColorBox.__str__(self) + " text: " + str(self.text)
```

What are the issues?

- Inheritance tree becomes a DAG
- What’s the problem?

What are the issues?

- Issue 1: fields/methods with the same name inherited from two different places
- Issue 2: diamond problem, same exact field inherited by two different paths

What are the issues?

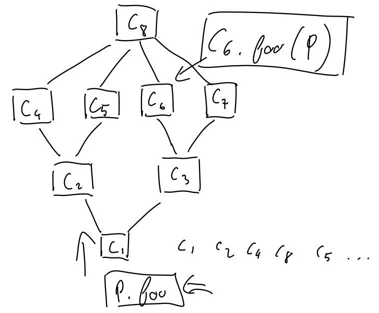
- Because of these issues, Java does not allow multiple inheritance
- Java does allow multiple inheritance of interfaces. How is that different from general multiple inheritance?

How Python solves these issues

- When you say: class $C(C_1, C_2, ...)$
- For any attribute not defined in C, Python first looks up in C_1, and parents of C_1
- If it doesn't find it there, it looks in C_2 and parents of C_2
- And so on...
- What kind of search is this?

How Python solves these issues

How Python solves these issues

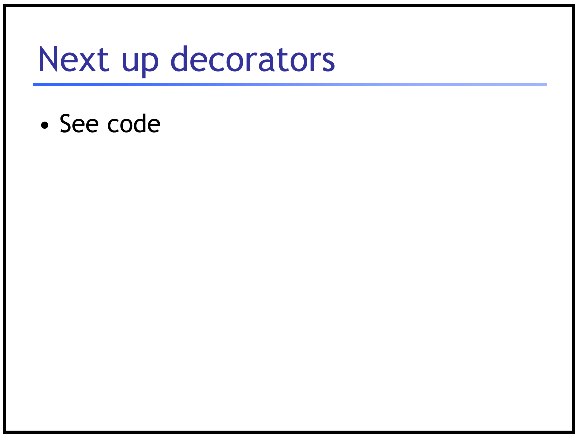


Does this solve the two issues?

- Issue 1: fields/methods with the same name inherited from two different places
  - Solved because we give leftmost parent priority
- Issue 2: diamond problem, same exact field inherited by two different paths
  - Solved because there is only one copy

Python's solutions

- For certain methods, may want one parent, whereas for other methods, may want another. Can always overwrite method and redirect to the right parent
- What about BFS?



Next up decorators

- See code