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

- Which should be a sub-class of which?

Option 1: Rectangle isa Square

- 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

+ Follows isa relationship from math
+ Don’t need to write two area methods
  – Can’t enforce 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 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

```py
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(C₁, C₂, …)
  - For any attribute not defined in C, Python first looks up in C₁, and parents of C₁
  - If it doesn’t find it there, it looks in C₂ and parents of C₂
  - And so on...
  - What kind of search is this?
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?