Reminder

On Friday, January 20, class will be in OI G162 (basement of OISE).

Outline

Managing Attributes

Types within Types... Composition!

Generalize Classes with Inheritance

Managing Attributes **num** and **denom**

Suppose that client code written by billions of developers uses Rational, but some of them complain that that class doesn't protect them from silly mistakes like supplying non-integers for the numerator or denominator, or even zero for the denominator...

After you have already shipped class Rational, you can write methods **_get_num**, **_set_num**, **_get_denom**, and **_set_denom**, and then use **property** to have Python use these functions whenever it sees **num** or **denom**

Shapes with extras

We decide to devise the following class

> *Squares have four vertices (corners) have a perimeter, an area, can move themselves by adding an offset point to each corner, and can draw themselves.*

Use Composition

Squares need drawing capabilities, so make sure each Square has a Turtle. Furthermore, the vertices of Squares are Points, and if we include those we'll get the ability to add an offset point and calculate distance... All without writing code to duplicate the capabilities of Turtle or Point.

More Square-like classes

What if we decided to devise a RightAngleTriangle class with similar characteristics to Square? There is an implementation of RightAngleTriangle, but it has a problem:

There's a lot of duplicate code. What do you suggest?

We could try:

1. Cut-paste-modify Square $\longrightarrow$ RightAngleTriangle?
2. Include a Square in the new class to get at its attributes and services??

We really need a general Shape with the features that are common to both Square and RightAngleTriangle, and perhaps other shapes that come along

Abstract Class Shape

Most of the features of Square are identical to RightAngleTriangle. Indeed we (shamefully...) cut-and-pasted a lot...

The differences are the class names (Square, RightAngleTriangle) and the code to calculate the area.

Put the common features into Shape, with unimplemented _set_area as a place-holder...

Declare Square and RightAngleTriangle as subclasses of Shape, inheriting the identical features by declaring:

class Square(Shape): ...

Inherit, Override, or Extend?

Subclasses use three approaches to recycling the code from their superclass, using the same name

1. Methods and attributes that are used as-is from the superclass are **inherited** — examples?
2. Methods and attributes that replace what's in the superclass **overriden** — example?
3. Methods and attributes that add to what is in the superclass are **extended** — example?

Write general code

Client code Written to use Shape will now work with subclasses of Shape — even those written in the future.

The client code can rely on these subclasses having methods such as **move_by** and **draw**