CSC148 winter 2018

special methods, property, composition, inheritance
week 2

Danny Heap
heap@cs.toronto.edu
BA4270 (behind elevators)
http://www.cdf.toronto.edu/~csc148h/winter/
416-978-5899

January 14, 2018

Outline

special methods

types within types... composition!

generalize classes with inheritance

rational fractions

Similarly to last week, we want to design an implement a class for rational numbers. We follow a design recipe for classes.

design class Rational

Rational numbers are ratios of two integers n/d, where n is called the numerator and d is called the denominator. The denominator d is non-zero. Operations on rationals include addition, multiplication, and comparisons:
$>, \quad <, \quad \geq, \quad \leq, \quad =.$

...Create our own Rational class.

build class Rational

Define a class API:

1. choose a class name and write a brief description in the class docstring.
2. write some examples of client code that uses your class
3. decide what services your class should provide as public methods, for each method declare an API (examples, header, type contract, description)
4. decide which attributes your class should provide without calling a method, list them in the class docstring

continue building class Rational

Implement the class:

1. body of special methods `__init__`, `__eq__`, and `__str__`

2. body of other methods[1]

[1] use the CSC108 function design recipe

special, aka magic, methods

Python recognizes the names of special methods such as **_init_**, **_eq_**, **_add_**, and **_mul_** and has short-cuts (aliases) for them. This **syntactic sugar** doesn't change the **semantics** (meaning) of these methods, but may allow more manageable code.

For example, suppose you create a list of Rational, and then want to sort it, or check to see whether an equivalent element is in it... **_lt_** and friends...

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**

... or use **assert**

the Python Way (TM)

- make public attributes directly accessible (no accessors, aka getters/setters)
- use **assert**
- use **property** to delegate the management of public attributes behind the scenes

shapes with extras

I 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.

Here's an implementation of Square

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 may come along

abstract class Shape

most of the features of Square are identical to RightAngleTriangle. Indeed I (blush) 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**

Here is some client code that takes a list objects from subclasses of Shape, moves each object around, and then draws it.