OO Example: Fractions

20 Nov 2009
CMPT140
Dr. Sean Ho
Trinity Western University

TRINITY WESTERN UNIVERSITY

OO Review: user-defined types

- A class is a user-defined container type
  - Attributes and methods
- Let's define a Fraction type
  - A fraction has an integer numerator and integer denominator
  - Attributes?
  - numer, denom
  - Methods?
  - add, sub, mul, etc.
- See oofraction.py in our example directory

Creating a bare Fraction class

- **class Fraction:**
- Constructor, with optional arguments:
  - We want to hide the numer and denom:
    - **def __init__(self, n=0, d=1):**
      - **self.__numer = n**
      - **self.__denom = d**
  - Any potential problems/constraints?
- String representation, for print:
    - **def __str__(self):**
      - **return "%d / %d" %**
        (self.__numer, self.__denom)

Using the Fraction class

- This is enough for us to create a Fraction object
  - a.k.a. “create a Fraction instance”
  - a.k.a. “instantiate the Fraction class”
  - **f1 = Fraction(2, 3)**
  - **print f1 # “2 / 3”**
- We can't do much with our Fraction object yet, so the next step is to implement some methods
- Multiple methods may want to check the constraint of denom ≠ 0: make a helper method

Helper: check constraints

- Constraint: denom should never be 0
- Don't want this to be publicly-accessible, so start name with '__' (double-underscore): hidden from view in Python
  - In C++/Java, can declare it 'private'
    - **def __check(self):**
- How to flag error? Use exceptions!
  - **if denom == 0:**
    - **raise ZeroDivisionError**
  - Up to whoever is using this Fraction to handle the error

Set/get (mutator/accessor)

- We have hidden the attributes __numer and __denom from direct access by other programs
- We can give them read or write access to those attributes, but only through our methods:
  - Get method (accessor): def get_N():
  - Set method (mutator): def set_N():
- This way we can do safety checking, e.g., check if denom is being set to 0
- Potentially: security/permissions, who is modifying this attribute, logging, etc.

Python customizations

- Now we can define the methods add, mul, etc.!
- Certain method names are special in Python:
  - __init__: Called by the constructor when we setup a new instance
  - __str__: Called by print
  - __mul__: Overloads the (*) operator
  - __add__: Overloads the (+) operator
  - __le__: Overloads the (<) operator
  - etc. (pretty much any operator can be overloaded!)
    - **http://docs.python.org/ref/specialnames.html**

e.g.: Multiplication method

- Multiplication (*) operator takes two operands:
  - self (the current Fraction object) and other (the other Fraction being multiplied):
  - **def __mul__(self, other):**
  - e.g., if f2 and f2 are Fractions, then doing f1 * f2 is equivalent to f1.__mul__(f2)
  - self refers to f1, other refers to f2
- Create a new Fraction object as the product:
  - **p = Fraction(self.get_N() * other.get_N(), self.get_D() * other.get_D())**
- Then simplify and return the product

Using customizations

- Now that we've written our multiplication method with the special name __mul__(), we can do:
  - f1 = Fraction(2, 3)
  - f2 = Fracion (1, 2)
  - print f1 # 2 / 3
  - print f2 # 1 / 2
  - print f1 * f2 # 2 / 6
- The other operators /, +, -, and even < can be defined similarly: operator overloading (extending definition of '*' to Fraction type)