

00: Creating Classes

Fraction.py Example

4 Nov 2010

CMPT140

Dr. Sean Ho

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 two parameters `n` and `d`:
 - Hide the instance attributes `numer`, `denom`:

```
def __init__(self, n, d):  
    self.__numer = n  
    self.__denom = d
```

- Any potential problems/constraints?
- Default values for `n` and `d`?
- 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”
- ```
from oofraction import Fraction
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  $\neq$  0**: make a **helper method**

# Helper func.: check constraints

- Constraint: `denom` should never be 0
- Don't want this method to be publicly accessible, so start name with `'__'` (double-underscore): `hidden` from view in Python

- ◆ (In C++/Java, 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 permit **read** or **write** access to those attributes, but **only** through our methods:
  - **Get** method (**accessor**): `def getN():`
  - **Set** method (**mutator**): `def setN():`
- This way we can do **safety** checking, e.g., check if `denom` is being set to `0`
- Other potential uses: **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
  - ◆ `__truediv__`: Overloads the (`/`) operator (`//` is '`floordiv`')
  - ◆ `__le__`: Overloads the (`<`) operator
  - ◆ etc. (pretty much any operator can be **overloaded!**)
    - **See Python ref §3.3**

# 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 `f1` 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.getN() * other.getN(),
 self.getD() * other.getD())
```

- Then reduce the fraction 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 = Fraction(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)

# Making a testbed

- Include a **testbed** in our module that shows off all the features of our new class **Fraction**:

```
def runtests():
```

- Make a **list** of all the test cases, as **strings**:

```
tests = ["Fraction(2,3) + Fraction(1,3)", ...
```

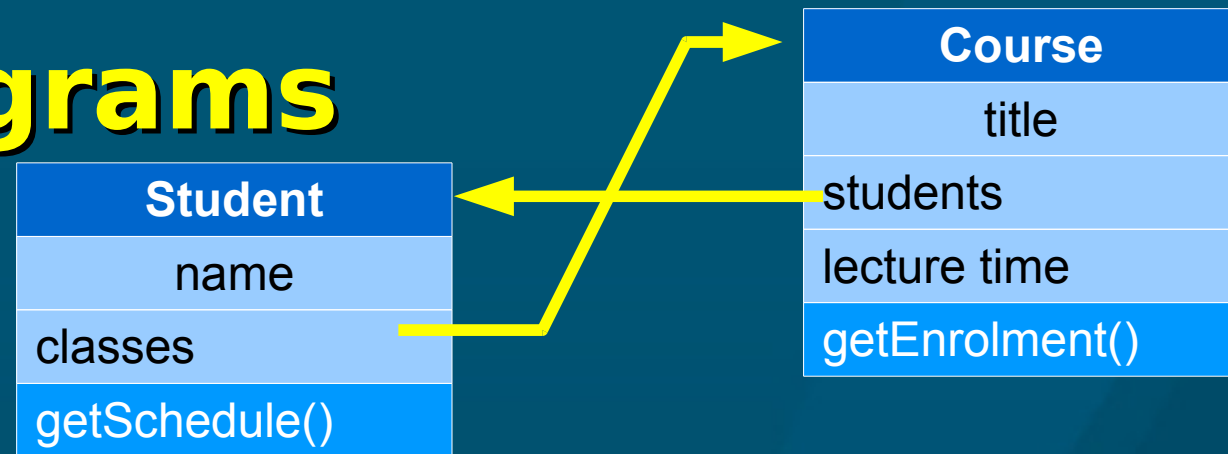
- **Print** each test and use **eval()** to **execute** it:

```
for test in tests:
```

```
 print(test, '=', eval(test))
```

- May also specify expected **result** for each test:
  - Automated **unit testing** to check for errors

# OO class diagrams



- An OO program is a collection of **classes**
  - Create objects: **instances** of the classes
  - Objects pass **messages** to each other
- A **class diagram** shows the classes and their relationships to each other:
  - **Name** of class
  - **Attributes** (public and private)
  - **Methods** (public and private)