

COMPSCI 105 S1 2017 Principles of Computer Science

Classes 3



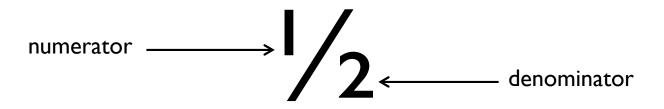
Exercise

- Create a Student class:
 - The Student class should have three attributes: id, last_name, and first_name.
 - Create a constructor to initialize the values
 - Implement the __repr__ method and __str__ method



Write a class to represent fractions in Python

- create a fraction
- ▶ add
- subtract
- multiply
- divide
- text representation





- > Python operators work for built-in classes.
 - But same operator behaves differently with different types.
 - E.g. the + operator:
 - perform arithmetic addition on two numbers,
 - merge two lists
 - concatenate two strings.
 - Allow same operator to have different meaning according to the context is called operator overloading

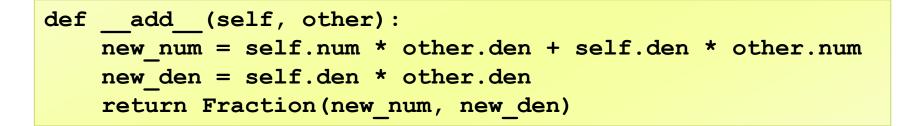
Operator	Expression	Internally
Addition	fl + f2	fladd(f2)
Subtraction	fl - f2	f1sub(f2)
Equality	fl == f2	fleq(f2)



add_

The __add__ method is called when the + operator is used

If we implement __add__ then we can use + to add the objects
fI + f2 gets translated into fI.__add__(f2)



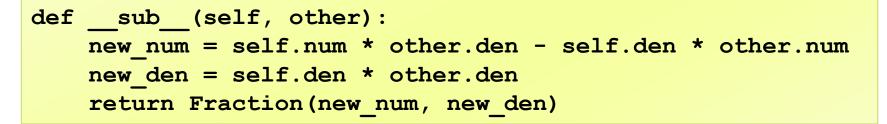


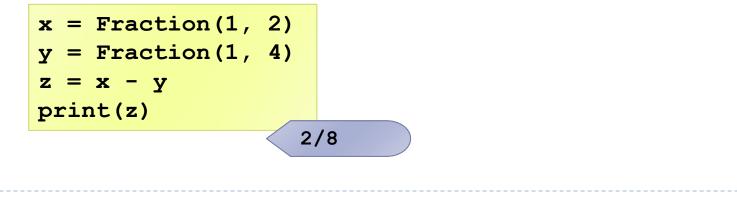
<u>sub</u>

The __sub__ method is called when the - operator is used

If we implement _____sub___ then we can use - to do subtraction

fl - f2 gets translated into fl.__sub__(f2)



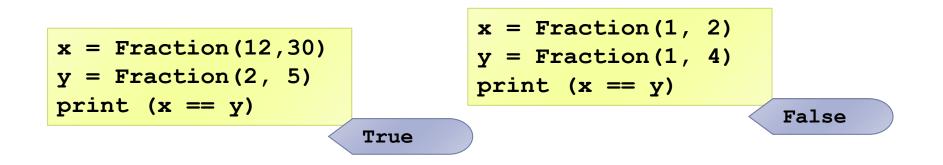




The ____eq__ method checks equality of the objects

- Default behaviour is to compare the references
- We want to compare the contents

def __eq__(self, other):
 return self.num * other.den == other.num * self.den





What is the output of the following code?

```
x = Fraction(2, 3)
y = Fraction(1, 3)
z = y + y
print(x)
print(z)
print(x == z)
```

x = Fraction(2, 3)print (x == 2)

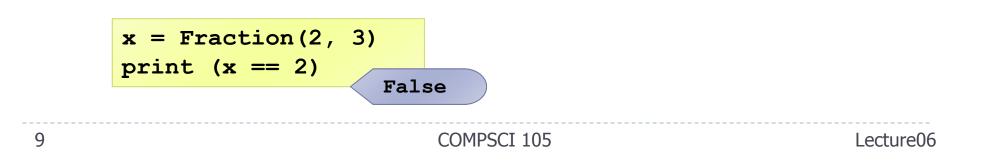
AttributeError: 'int' object has no attribute 'den'



Check the type of the other operand

- If the type is not a Fraction, then not equal?
- What other decisions could we make for equality?

```
def __eq_ (self, other):
    if not isinstance(other, Fraction):
        return False
        return self.num * other.den == other.num * self.den
```





- Fractions:
 - 12/30
 - > 2/5
- The first fraction can be simplified to 2/5
- The Common Factors of 12 and 30 were 1, 2, 3 and 6,
- The Greatest Common Factor is 6.
 - So the largest number we can divide both 12 and 30 evenly by is 6
- And so 12/30 can be simplified to 2/5



Use Euclid's Algorithm

- Given two numbers, n and m, find the number k, such that k is the largest number that evenly divides both n and m.
 - Example: Find the GCD of 270 and 192,
 - □ gcd(270, 192): m=270, n=192 (m≠0, n ≠0)
 - □ Use long division to find that 270/192 = 1 with a remainder of 78. We can write this as: gcd(270,192) = gcd(192,78)
 - □ gcd(192, 78) : m=192, n=78 (m≠0, n ≠0)
 - □ 192/78 = 2 with a remainder of 36 with a remainder of 78. We can write this as: gcd(192,78) = gcd(78,36)
 - □ gcd(78, 36) : m=78, n=36 (m≠0, n ≠0)
 - \Box 78/36 = 2 with a remainder of 6
 - \Box gcd(78,36) = gcd(36,6)
 - \Box gcd(36, 6) : m=36, n=6 (m \neq 0, n \neq 0)
 - \Box 36/6 = 6 with a remainder of 0
 - \Box gcd(36,6) = gcd(6,0) = 6

```
def gcd(m, n):
  while m % n != 0:
        old_m = m
        old_n = n
        m = old_n
        n = old_m % old_n
        return n
```

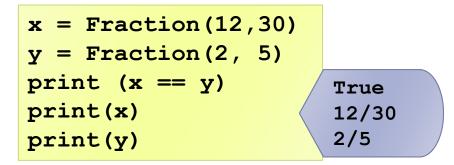


- We can improve the constructor so that it always represents a fraction using the "lowest terms" form.
 - What other things might we want to add to a Fraction?

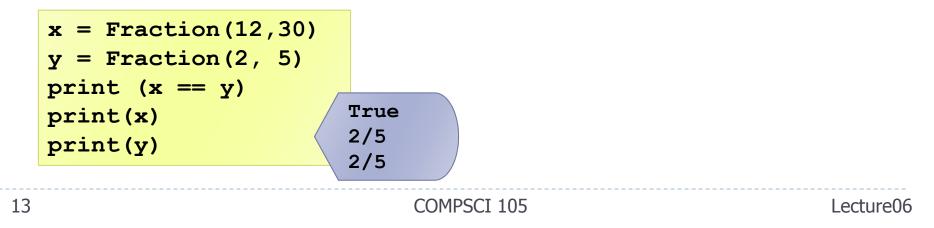
```
class Fraction:
    def __init__(self, top, bottom):
        common = Fraction.gcd(top, bottom) #get largest common term
        self.num = top // common #numerator
        self.den = bottom // common #denominator
    def gcd(m, n):
        while m % n != 0:
            old_m = m
            old_m = n
            m = old_n n
            n = old_n % old_n
        return n
```



Without the GCD



Using the GCD:





- Many standard operators and functions: https://docs.python.org/3.4/library/operator.html
- Common Arithmetic operators
 - object.__add__(self, other)
 - object.__sub__(self, other)
 - object.__mul__(self, other)
 - object.__truediv__(self, other)
- Common Relational operators
 - object.__lt__(self, other)
 - object.__le__(self, other)
 - object.__eq__(self, other)
 - object.__ne__(self, other)
 - object.__gt__(self, other)
 - object.__ge__(self, other)

Inplace arithmetic operators

- object.__iadd__(self, other)
- object.__isub__(self, other)
- object.__imul__(self, other)
- object.__itruediv__(self, other)

Reversed versions:

- object.__radd__(self, other)
- object.__rsub__(self, other)
- object.__rmul__(self, other)
- object.__rdiv__(self, other)

• ...



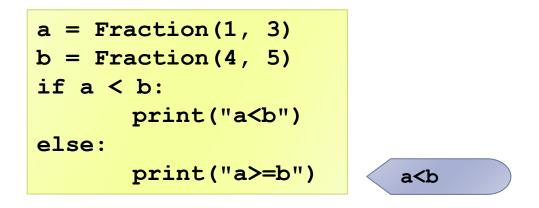
Implement the _____truediv___ of the Fraction class:

```
a = Fraction(1, 3)
b = Fraction(4, 5)
d = a / b
print (d)
```

5/12



Implement the __lt__ method to compare two Fraction objects:



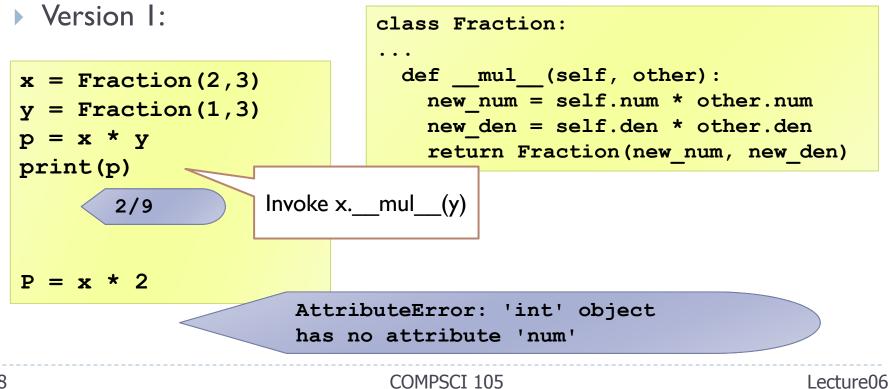
Forward, Reverse and In-Place

- Every arithmetic operator is transformed into a method call. By defining the <u>numeric special methods</u>, your class will work with the built-in arithmetic operators.
 - First, there are as many as *three* variant methods required to implement each operation.
 - For example, * is implemented by __mul__, __rmul__ and __imul__
 - There are forward and reverse special methods so that you can assure that your operator is properly commutative.
 - You don't need to implement all three versions.
 - The reverse name is used for special situations that involve objects of multiple classes.



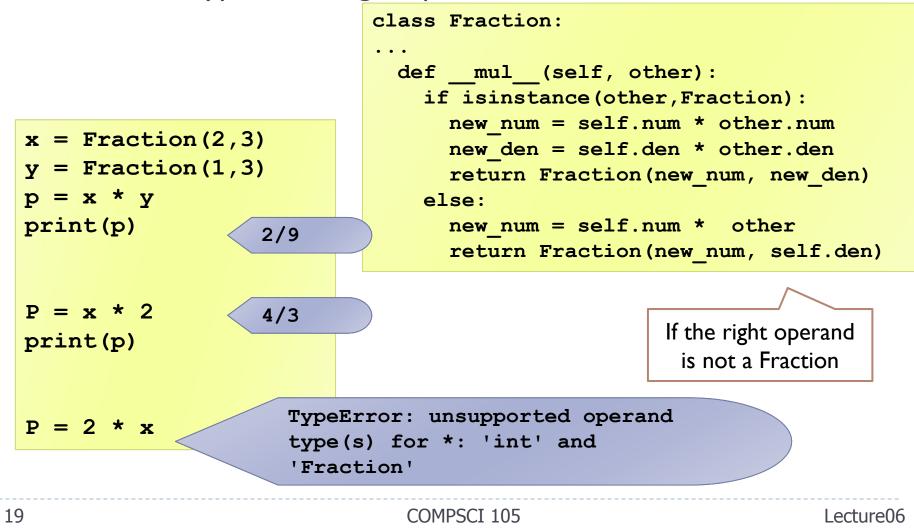
Locating an appropriate method for an operator

First, it tries a class based on the left-hand operand using the "forward" name. If no suitable special method is found, it tries the right-hand operand, using the "reverse" name.



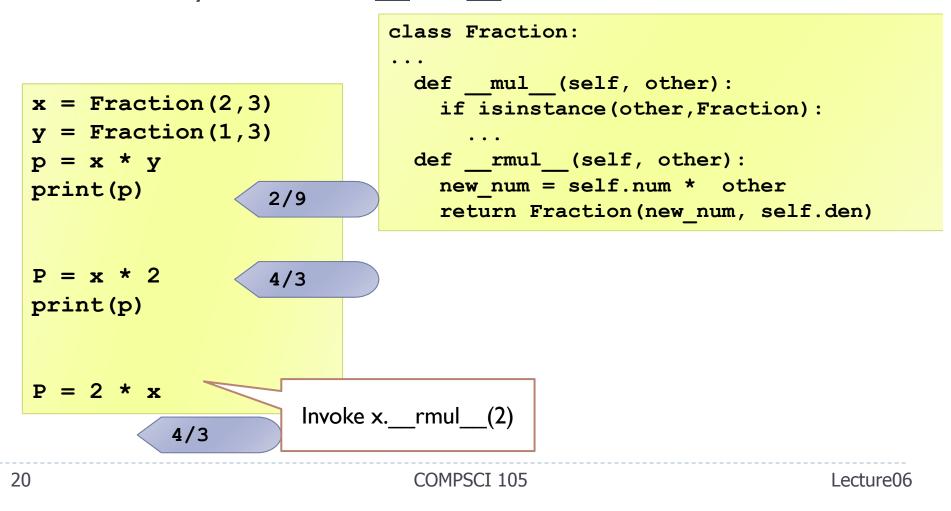


• Check the type of the right operand:



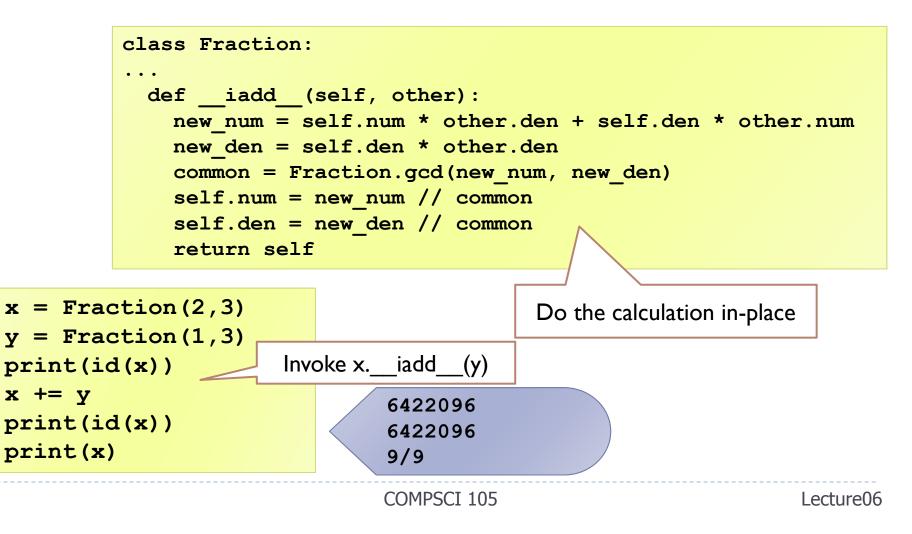


If the left operand of * is a primitive type and the right operand is a Fraction, Python invokes ___rmul___





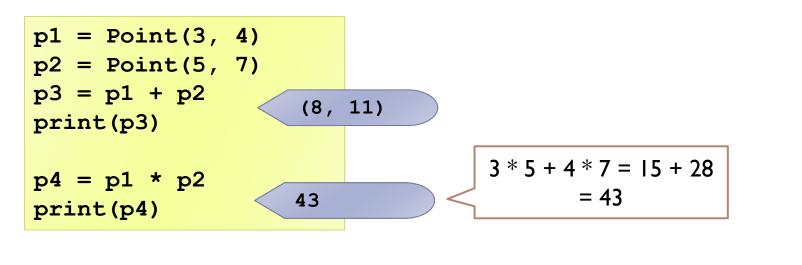
+=, -=, *=, /= etc





Overload the following operators in the Point class:

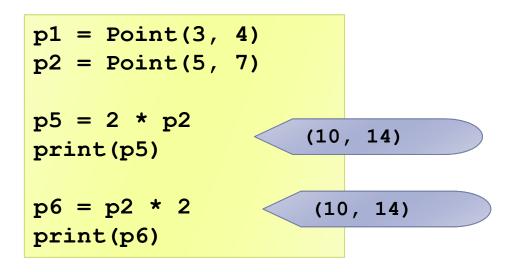
- +: return a new Point that contains the sum of the x coordinates and the sum of the y coordinates.
- *: computes the dot product of the two points, defined according to the rules of linear algebra





If the left operand of * is a primitive type and the right operand is a Point, Python invokes __rmul__, which performs scalar multiplication:

COMPSCI 105





- A class is a template, a blueprint and a data type for objects.
- A class defines the data fields of objects, and provides an initializer for initializing objects and other methods for manipulating the data.
- The initializer always named __init__. The first parameter in each method including the initializer in the class refers to the object that calls the methods, i.e., self.
- Data fields in classes should be hidden to prevent data tampering and to make class easy to maintain.
- We can overwrite the default methods in a class definition.