파이썬으로 배우는 데이터 구조

한동대학교
전산전자공학부
김영섭 교수

학습 목표

제자리(In-Place) 연산이 가능하게 하는
메소드를 정의할 수 있다

Data Structures in Python
Chapter 1 - 2

- Object-Oriented Programming
- OOP in Python
- OOP - Fraction Example
- OOP - Classes
- **OOP - In-Place Operators**
- Exceptions
- Exception Clauses

Agenda

- Classes
  - Overloading Operators
  - __add__, __sub__, __eq__
  - GCD
  - __lt__
- **In-Place Operations**
  - **__mul__, __rmul__, __imul__**

- References:
  - Problem Solving with Algorithms and Data Structures using Python
    - Chapter 1.13 Object-Oriented Programming in Python
    - Chapter 2.2 A Proper Class

Forward, Reverse and In-Place

- Every arithmetic operator is transformed into a method call.
  By defining **the numeric special methods**, 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.

mul vs. rmul

- Locating an appropriate method for an operator
  - First, it tries a class based on the left operand using the "forward" name. If no suitable special method is found, it tries the right-hand operand, using the "reverse" name.
- **Sample Run and Version 1:**

```
x = Fraction(2,3)
y = Fraction(1,3)
p = x * y
print(p)

p = x * 2
print(p)
```

Invoke x.__mul__(y)

```
2/9
```

```
AttributeError:
'int' object  has
no attribute 'num'
```

```
class Fraction:
...
    def __mul__(self, other):
        num = self.num * other.num
        den = self.den * other.den
        return Fraction(num, den)
```

mul vs. rmul

- Locating an appropriate method for an operator
  - First, it tries a class based on the left operand using the "forward" name. If no suitable special method is found, it tries the right-hand operand, using the "reverse" name.
- **Sample Run and Version 2:**

```
x = Fraction(2,3)
y = Fraction(1,3)
p = x * y
print(p)
```

Invoke x.__mul__(y)

2/9

```
p = x * 2
print(p)
```

Invoke x.__mul__(y)

4/3

Version 2 checks the type of the right operand:

```
class Fraction:
...
    def __mul__(self, other):
        if isinstance(other, Fraction):
            num = self.num * other.num
            den = self.den * other.den
            return Fraction(num, den)
        else:
            num = self.num * other
            return Fraction(num, self.den)
```

If the right operand is not a Fraction

mul vs. rmul

- Locating an appropriate method for an operator
  - First, it tries a class based on the left operand using the "forward" name. If no suitable special method is found, it tries the right-hand operand, using the "reverse" name.
- **Sample Run and Version 2:**

```
x = Fraction(2,3)
y = Fraction(1,3)
p = x * y
print(p)

p = x * 2
print(p)

p = 2 * x
print(p)
```

Invoke x.__mul__(y)

2/9

Invoke x.__mul__(y)

4/3

```
TypeError: unsupported
operand type(s) for *:
'int' and 'Fraction'
```

Version 2 checks the type of the right operand:

```
class Fraction:
...
    def __mul__(self, other):
        if isinstance(other, Fraction):
            num = self.num * other.num
            den = self.den * other.den
            return Fraction(num, den)
        else:
            num = self.num * other
            return Fraction(num, self.den)
```

If the right operand is not a Fraction

mul vs. rmul

- Locating an appropriate method for an operator
  - First, it tries a class based on the left operand using the "forward" name. If no suitable special method is found, it tries the right-hand operand, using the "reverse" name.
- **Sample Run and Version 3:**

```
x = Fraction(2,3)
y = Fraction(1,3)
p = x * y
print(p)

p = x * 2
print(p)

p = 2 * x
print(p)
```

Invoke x.__mul__(y)

```
2/9
```

Invoke x.__mul__(y)

```
4/3
```

```
TypeError: unsupported
operand type(s) for *:
'int' and 'Fraction'
```

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

```
class Fraction:
...
    def __mul__(self, other):
        if isinstance(other, Fraction):
            num = self.num * other.num
            den = self.den * other.den
            return Fraction(num, den)
        else:
            num = self.num * other
            return Fraction(num, self.den)
```

mul vs. rmul

- Locating an appropriate method for an operator
  - First, it tries a class based on the left operand using the "forward" name. If no suitable special method is found, it tries the right-hand operand, using the "reverse" name.
- **Sample Run and Version 3:**

```
x = Fraction(2,3)
y = Fraction(1,3)
p = x * y
print(p)

p = x * 2
print(p)

p = 2 * x
print(p)
```

Invoke x.__mul__(y)

2/9

Invoke x.__mul__(y)

4/3

Invoke x.__rmul__(2)

4/3

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

```
class Fraction:
...
    def __mul__(self, other):
        if isinstance(other, Fraction):
            num = self.num * other.num
            den = self.den * other.den
            return Fraction(num, den)
        else:
            num = self.num * other
            return Fraction(num, self.den)

    def __rmul__(self, other):
        num = self.num *  other
        return Fraction(num, self.den)
```

In-Place Operators

- +=, -=, *=, /= etc
- Sample Run:

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

```
6422096

6422096
1/1
```

Invoke x.__iadd__(y)

- Code:

```
class Fraction:
...
    def __iadd__(self, other):
        num = self.num * other.den + self.den * other.num
        den = self.den * other.den
        gcd = Fraction.gcd(num, den)
        self.num = num // gcd
        self.den = den // gcd
        return self
```

Do the calculation in-place

Exercise 4

- Overload the following operators in the **`Point`** class:
  - `+`: returns a new **`Point`** that contains the sum of x's and the sum of y's, respectively.
  - `*`: computes the **dot product** of the two points, defined according to the rules of linear algebra.

- Sample Run:

```
p1 = Point(3, 4)
p2 = Point(5, 7)
p3 = p1 + p2
print(p3)
```

```
Point(8, 11)
```

```
print(p1 * p2)
```

```
43
```

= 3*5 + 4*7 = 15 + 28

Exercise 5

- If the left operand of `*` or `+` is a primitive type and the right operand is a Point, Python invokes **`__rmul__`** and **`__radd__`**.
- Let them perform scalar multiplication and addition, respectively in your code.
- Sample Run:

```
p1 = Point(3, 4)
p2 = Point(5, 7)
p5 = 2 * p1
print(p5)          Point(6, 8)

p6 = p2 * 2
print(p6)          Point(10, 14)

print(2 + p1)      Point(5, 6)
print(p1 + 2)      Point(7, 9)
```

Exercise 6

- Overload the following operators in the `Circle` class:
  - `+`: returns a new `Circle` that contains the sum of two radii.
  - `*`: computes a new `Circle` that contains the multiplication of two radii.
  - If the left operand of `*` or `+` is a primitive type and the right operand is a `Circle`, Python invokes `__rmul__` and `__radd__`. Let them perform scalar multiplication and addition, respectively in your code.
- Sample Run:

```
c1 = Circle(2)
c2 = Circle(3)
print(c1 + c2)
print(c1 * c2)
```

```
Circle(5)
Circle(6)
```

```
print(c1 * 2)
print(2 * c2)
```

```
Circle(4)
Circle(6)
```

```
print(3 + c1)
print(c2 + 3)
```

```
Circle(5)
Circle(6)
```

Summary

- We can **override(재정의) the default methods** in a class definition.

학습 정리

1) __rmul__(), __radd__() 같은 특별한 메소드는 피연산자의 타입이 서로 맞지 않을 때 순서를 바꾸어 계산할 수 있도록 돕는다

2) 제자리(In-Place) 연산자는 객체의 reference가 바뀌지 않으므로 self를 반환해야 한다

파이썬으로 배우는 데이터 구조

수고했습니다
곧 다음 시간에
다시 뵙겠습니
다