Data Structures in Python
Chapter 1

1. Introduction - Review Python
2. Objects and References
3. Object-Oriented Programming
4. OOP - Fraction Example
5. **OOP - Classes**
6. Exceptions 1, 2
7. JSON

Prof. Youngsup Kim, idebtor@gmail.com, CSEE Dept., Grace School Rm204 Handong Global University

하나님은 모든 사람이 구원을 받으며 진리를 아는데 이르기를 원하시느니라 (딤전2:4)

내 아들들을 먼 곳에서 이끌며 내 딸들을 땅 끝에서 오게 하며 내 이름으로 불려지는 모든 자 곧 내가 내 영광을 위하여 창조한 자를 오게 하라 그를 내가 지었고 그를 내가 만들었노라 (사43:6-7)

너는 청년의 때에 너의 창조주를 기억하라 곧 곤고한 날이 이르기 전에, 나는 아무 낙이 없다고 할 해들이 가깝기 전에 (전12:1)

그런즉 너희가 먹든지 마시든지 무엇을 하든지 다 하나님의 영광을 위하여 하라 (고전10:31)

Agenda

- Topics:
  - Overloading Operators
  - __add__, __sub__, __eq__
  - GCD
  - __lt__
  - __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

Exercise

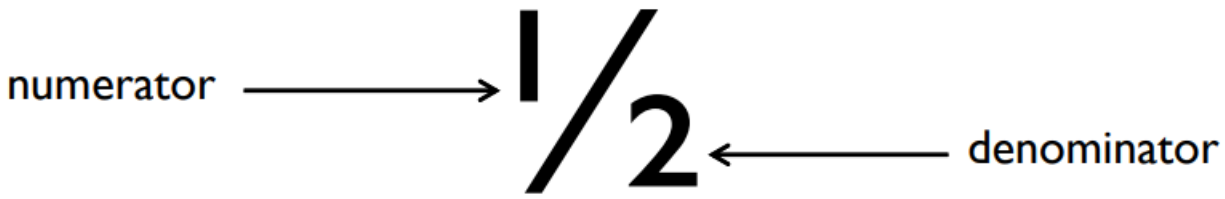
- Create a **Student** class:
  - The Student class should have three attributes: id, last_name, and first_name.
  - Create a constructor to initialize the values
  - Implement the __repr__ method and __str__ method
- Sample Run:

```
s1 = Student(12, 'David', 'Song')
print(s1)
print(s1.__repr__())
s1
```

```
12: David Song
Student(12, David, Song)
Student(12, David, Song)
```

Reminder: Fraction class

- Write a class to represent fractions in Python
  - create a fraction
  - add
  - subtract
  - multiply
  - divide
  - text representation



Overloading Operators

- Python operators work for built-in classes.
  - But same operator behaves differently with different types.
  - E.g. the + operator:
    - performs arithmetic addition on two numbers,
    - merges two lists,
    - concatenates two strings.
  - Allow same operator to have different meaning according to the context is called **operator overloading(연산자 오버딩).**

Operator	Expression	Internally
Addition	f1 + f2	f1.__add__(f2)
Subtraction	f1 - f2	f1.__sub__(f2)
Equality	f1 == f2	f1.__eq__(f2)

__sub__

- The __sub__ method is called when the - operator is used.
  - If we implement __sub__ then we can use - to do subtraction.
    - f1 - f2 gets translated into f1.__sub__(f2)
  - Sample Run:

```
x = Fraction(1, 2)
y = Fraction(1, 4)
z = x - y
print(z)
```

```
2/8
```

```
= self - other
= 1/2 - 1/4
= (1 * 4 - 1 * 2) / (2 * 4)
= 2/8
```

  - Code:

```
def __sub__(self, other):
```

__eq__

- The __eq__ method checks equality of the objects.
  - Default behavior is to compare the references.
  - We want to compare the contents.
  - Sample Run:

```
x = Fraction(12, 30)
y = Fraction(2, 5)
print(x == y)
```

```
True
```

```
x = Fraction(4, 1)
y = Fraction(1, 4)
print(x == y)
```

```
False
```

  - Code:

```
def __eq__(self, other):

```

__eq__

- The __eq__ method checks equality of the objects.
  - Default behavior is to compare the references.
  - We want to compare the contents.
  - Sample Run:

```
x = Fraction(12, 30)
y = Fraction(2, 5)
print(x == y)
```

True

```
x = Fraction(4, 1)
y = Fraction(1, 4)
print(x == y)
```

False

```
= (self == other)
= (12/30 == 2/5)
= (12 * 5 == 2 * 30)
= (60 == 60)
```

  - Code:

```
def __eq__(self, other):
```

Exercise 1

- 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)
```

Exercise 1

- 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)
```

```
2/3
1/3
True
```

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

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

Improving __eq__

- 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
```

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

```
False
```

Improving your code

- 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**.

Greatest Common Divisor

- 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.
        We can write this as: gcd(192,78) = gcd(78,36)
    - gcd(78, 36) : m=78, n=36 (m≠0, n≠0)
      - 78/36 = 2 with a remainder of 6.
    - gcd(78,36) = gcd(36,6)
      - gcd(36, 6) : m=36, n=6 (m≠0, n≠0)
    - 36/6 = 6 with a remainder of 0
      - 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
```

Improve the constructor

- 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):
        gcd = Fraction.gcd(top, bottom) # get gcd
        self.num = top    // gcd
        self.den = bottom // gcd

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

Sample Run:

- **Without** the GCD

```
x = Fraction(12,30)
y = Fraction(2, 5)
print (x == y)
print(x)
print(y)
```

```
True
12/30
2/5
```

- **With** the GCD

```
x = Fraction(12,30)
y = Fraction(2, 5)
print (x == y)
print(x)
print(y)
```

```
True
2/5
2/5
```

Other standard Python operators

- Many standard operators and functions:
  - https://docs.python.org/3.9/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)`

- **In-place** 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)`
  - `...`

Exercise 2

- Implement the `__truediv__` of the Fraction class:

- Sample Run:

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

```
5/12
```

```
= (self / other)
= (1/3 / 4/5)
= (1 * 5 / 3 * 4)
= (5 / 12)
```

- Code

```
def __truediv__(self, other):
```

Exercise 2 solution

- Implement the `__truediv__` of the Fraction class:

- Sample Run:

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

```
5/12
```

```
= (self / other)
= (1/3 / 4/5)
= (1 * 5 / 3 * 4)
= (5 / 12)
```

- Code:

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

Exercise 3

- Implement the `__lt__` method to compare two Fraction objects:

- Sample Run:

```
a = Fraction(1, 3)
b = Fraction(4, 5)
if a < b:
    print("a < b")
else:
    print("a >= b")
```

```
a < b
```

- Code

```
= (self < other)
= (1/3 / 4/5)
= 5, 12
= 5 < 12
```

```
def __lt__(self, other):
```

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)
print(p1 * p2)
```

```
Point(8, 11)
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)

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

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

```
Circle(5)
Circle(6)

Circle(4)
Circle(6)

Circle(5)
Circle(6)
```

Summary

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