COMPSCI 105 S1 2017
Principles of Computer Science

Classes 3

Exercise

- 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

```
>>> s1 = Student(1, 'Angela', 'Chang')
>>> s1
>>> print(s1)
```

```
Student(1, Angela, Chang)
1: Angela Chang
```

Reminder – Fraction class

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

numerator $\longrightarrow$ 1/2 $\longleftarrow$ denominator

`Fraction.py`

Overloading Operators

- 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	f1 + f2	f1.__add__(f2)
Subtraction	f1 - f2	f1.__sub__(f2)
Equality	f1 == f2	f1.__eq__(f2)

__add__

- The __add__ method is called when the + operator is used
  - If we implement __add__ then we can use + to add the objects
    - f1 + f2 gets translated into f1.__add__(f2)

```
def __add__(self, other):
    new_num = self.num * other.den + self.den * other.num
    new_den = self.den * other.den
    return Fraction(new_num, new_den)
```

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

6/8

__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)

```
def __sub__(self, other):
    new_num = self.num * other.den - self.den * other.num
    new_den = self.den * other.den
    return Fraction(new_num, new_den)
```

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

2/8

__eq__

- 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
```

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

True

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

False

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)
```

```
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 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)
        - 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):
        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_n = n
            m = old_n
            n = old_m % old_n
        return n
```

Examples

- Without the GCD

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

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

- Using 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 funtions:
  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*)

Inplace arithmetic operators

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

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:

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

```
5/12
```

Exercise 3

- Implement the __lt__ method to compare two Fraction objects:

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

```
a<b
```

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-hand operand** using the "forward" name. If no suitable special method is found, it tries the right-hand operand, using the "reverse" name.
  - Version 1:

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

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

```
2/9
```

Invoke x.__mul__(y)

```
P = x * 2
```

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

Version 2

- Check the type of the right operand:

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

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

2/9

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

4/3

If the right operand is not a Fraction

```
P = 2 * x
```

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

Version 3

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

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

2/9

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

4/3

```
P = 2 * x
```

Invoke x.__rmul__(2)

4/3

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

In-Place Operators

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

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

Do the calculation in-place

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

Invoke x.__iadd__(y)

```
6422096
6422096
9/9
```

Exercise 4

- 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

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

p4 = p1 * p2
print(p4)
```

(8, 11)

43

3 * 5 + 4 * 7 = 15 + 28
= 43

Exercise 5

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

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

p5 = 2 * p2
print(p5)

p6 = p2 * 2
print(p6)
```

(10, 14)

(10, 14)

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.
- We can overwrite the default methods in a class definition.