3 Days Training on Python3

Day 2 : Module 7

Muhammad Saufy Rohmad

Module 7 (90 minutes)

Objectives

1. Class Inheritance
2. Why Bother with Object Orientation?
3. Operator Overloading

1. Class Inheritance

- Inheritance is a core feature of Object-Oriented Programming
- It allows one class to inherit data or behaviour from another class and is one of the key ways in which reuse is enabled within classes.
- Inheritance allows features defined in one class to be inherited and reused in the definition of another class.
- In an object-oriented system we can achieve the reuse of data or behaviour via inheritance.
- That is one class (in this case the Employee class) can inherit features from another class (in this case Person).

1.1 Class Inheritance – Parent Class

```
class Person:
    def __init__(self, name, age):
        self.name = name
        self.age = age
    def birthday(self):
        print('Happy birthday you were', self.age)
        self.age += 1
        print('You are now', self.age)
```

1.2 Class Inheritance – Child Class

```
class Employee(Person):
    def __init__(self, name, age, id):
        super().__init__(name, age)
        self.id = id
    def calculate_pay(self, hours_worked):
        rate_of_pay = 7.50
        if self.age >= 21:
        rate_of_pay += 2.50
        return hours_worked * rate_of_pay
```

1.2 Class Inheritance – another Child Class

```
class SalesPerson(Employee):
    def __init__(self, name, age, id, region, sales):
        super().__init__(name, age, id)
        self.region = region
        self.sales = sales
    def bonus(self):
        return self.sales * 0.5
```

1.2 Class Inheritance - Object

```
print('Person')
p = Person('John', 54)
print(p)
print('-' * 25)
print('Employee')
e = Employee('Denise', 51, 7468)
e.birthday()
print('e.calculate_pay(40):', e.calculate_pay(40))
print('-' * 25)
print('SalesPerson')
s = SalesPerson('Phoebe', 21, 4712, 'UK', 30000.0)
s.birthday()
print('s.calculate_pay(40):', s.calculate_pay(40))
print('s.bonus():', s.bonus())
```

1.3 Terminology Around Inheritance

- The following terminology is commonly used with inheritance in most object oriented languages including Python:
  - Class
  - Subclass
  - Superclass
  - Single or multiple inheritance

1.4 Overriding Methods

- Overriding occurs when a method is defined in a class (for example, Person) and also in one of its subclasses (for example, Employee).
- It means that instances of Person and Employee both respond to requests for this method to be run but each has their own implementation of the method.

1.4 Overriding Methods(2)

```
class Person:
    def __init__(self, name, age):
        self.name = name
        self.age = age
    def __str__(self):
        return self.name + ' is ' + str(self.age)


class Employee(Person):
    def __init__(self, name, age, id):
        super().__init__(name, age)
        self.id = id
    def __str__(self):
        return self.name + ' is ' + str(self.age) + ' - i
        str(self.id) + ')'
```

1.4 Overriding Methods(3)

- Run with

```
p = Person('John', 54)
print(p)
e = Employee('Denise', 51, 1234)
print(e)
```

- Generate output

```
John is 54
Denise is 51 - id(1234)
```

1.5 Multiple Inheritance

- Python supports the idea of multiple inheritance; that is a class can inherit from one or more other classes (many object-oriented languages limit inheritance to a single class such as Java and C#).
- At first sight multiple inheritance in Python might appear to be particularly useful;after all it allows you to mix together multiple concepts into a single class very easily and quickly.
- This is certainly true and it can be a very flexible feature if used with care.
- However, the word care is used here and should be noted

2. Why Bother with Object Orientation?

- Classes in an object-oriented language provide a number of features that are not present in procedural languages
  - Classes provide for inheritance.
  - Inheritance provides for reuse.
  - Inheritance provides for extension of a data type.
  - Inheritance allows for polymorphism.
  - Inheritance is a unique feature of object orientation.

3. Operator Overloading

- Operator overloading allows user defined classes to appear to have a natural way of using operators such as +, −, <, > or == as well as logical operators such as & (and) and | (or).
- This leads to more readable code as it is possible to write code such as:

```
q1 = Quantity(5)
q2 = Quantity(10)
q3 = q1 + q2
```

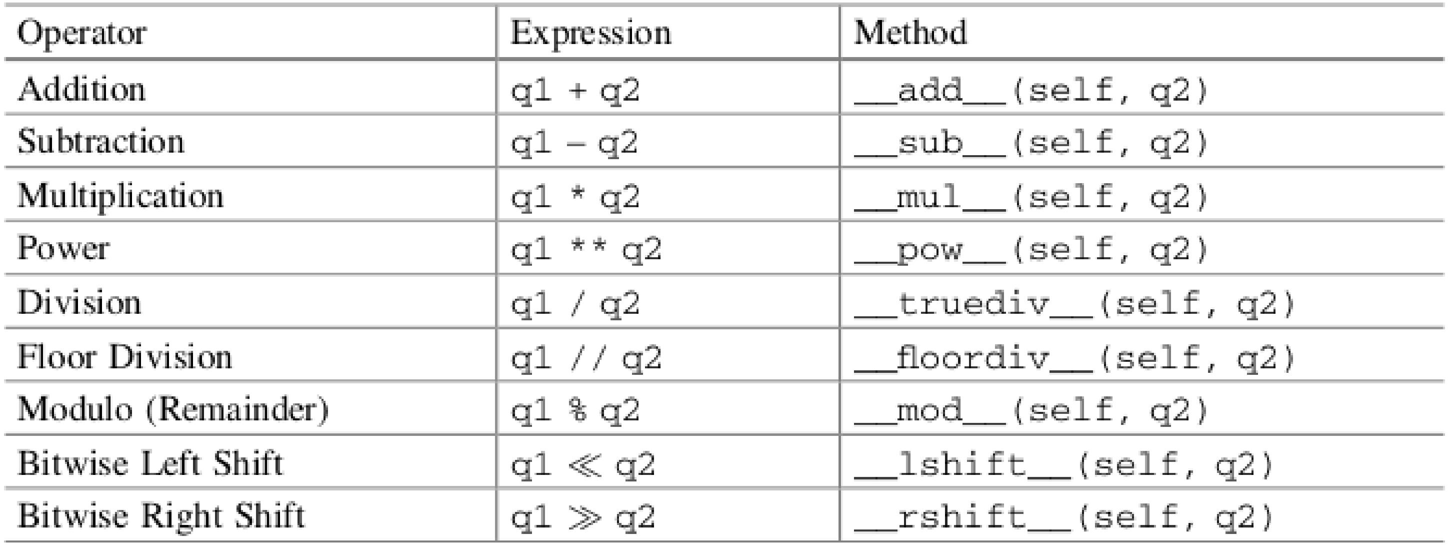
- The alternative would be to create methods such as add and write code such as

```
q1 = Quantity(5)
q2 = Quantity(10)
q3 = q1.add(q2)
```

- Which semantically might mean the same thing but feel less natural to most people.

3. Operator Overloading(2)

- There are nine different numerical operators that can be implemented by special methods; these operators are listed in the following table:



Operator	Expression	Method
Addition	q1 + q2	__add__(self, q2)
Subtraction	q1 – q2	__sub__(self, q2)
Multiplication	q1 * q2	__mul__(self, q2)
Power	q1 ** q2	__pow__(self, q2)
Division	q1 / q2	__truediv__(self, q2)
Floor Division	q1 // q2	__floordiv__(self, q2)
Modulo (Remainder)	q1 % q2	__mod__(self, q2)
Bitwise Left Shift	q1 ≪ q2	__lshift__(self, q2)
Bitwise Right Shift	q1 ≫ q2	__rshift__(self, q2)

3. Operator Overloading(3)

```
class Quantity:
    def __init__(self, value=0):
        self.value = value
    def __add__(self, other):
        new_value = self.value + other.value
        return Quantity(new_value)
    def __sub__(self, other):
        new_value = self.value - other.value
        return Quantity(new_value)
    def __mul__(self, other):
        new_value = self.value * other.value
        return Quantity(new_value)
    def __pow__(self, other):
        new_value = self.value ** other.value
        return Quantity(new_value)
    def __truediv__(self, other):
        new_value = self.value / other.value
        return Quantity(new_value)
    def __floordiv__(self, other):
        new_value = self.value // other.value
        return Quantity(new_value)
    def __mod__(self, other):
        new_value = self.value % other.value
        return Quantity(new_value)
    def __str__(self):
        return 'Quantity[' + str(self.value) + ']'
```

3. Operator Overloading(4)

- This means that we can now extend our simple application that uses the Quantity class to include some of these additional numerical operators:

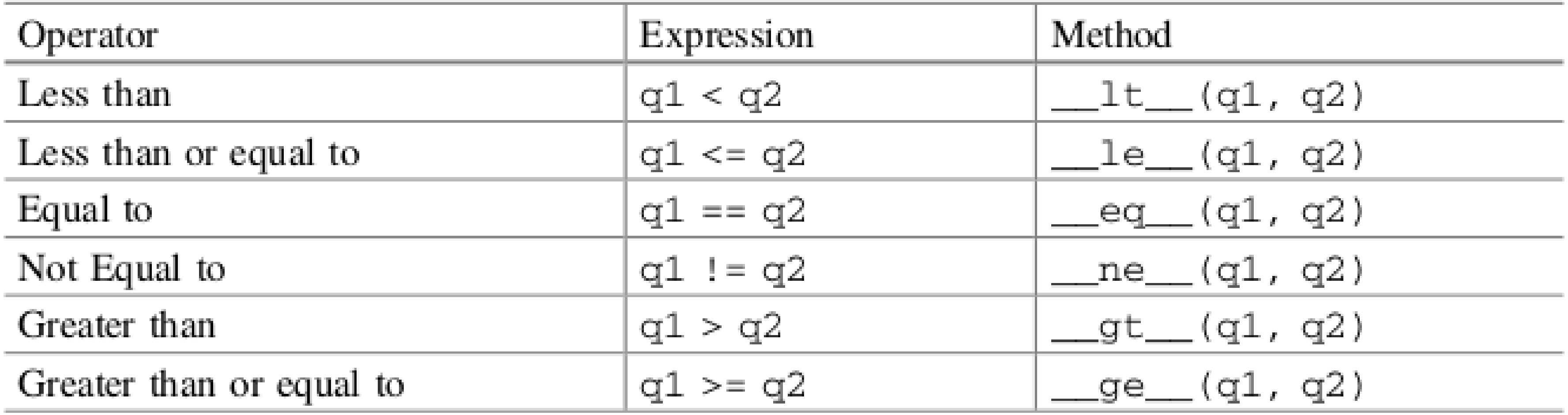
```
q1 = Quantity(5)
q2 = Quantity(10)
print('q1 =', q1, ', q2 =', q2)
q3 = q1 + q2
print('q3 =', q3)
print('q2 - q1 =', q2 - q1)
print('q1 * q2 =', q1 * q2)
print('q1 / q2 =', q1 / q2)
```

- The output :

```
q1 = Quantity[5] , q2=Quantity[10]
q3 = Quantity[15]
q2 -q1 = Quantity[5]
q1*q2 = Quantity[50]
q1/q2 = Quantity[0.5]
```

3. Operator Overloading(5)

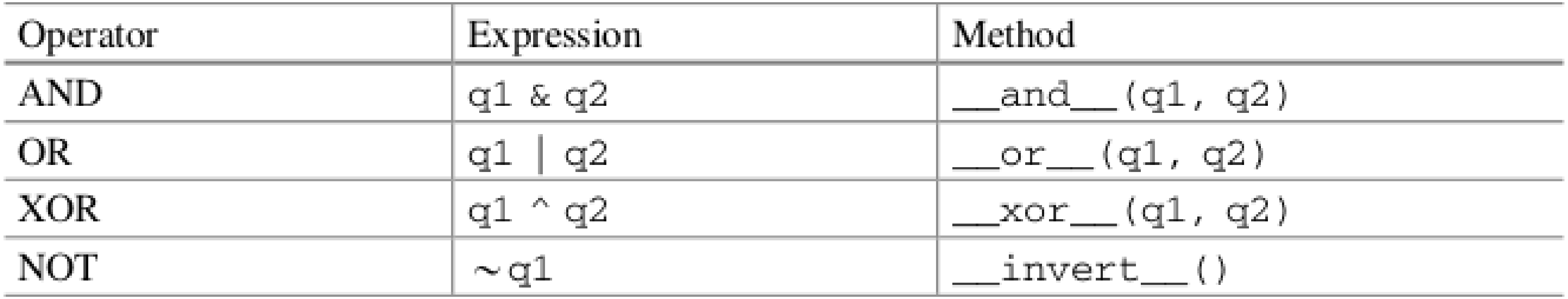
- Comparison Operator



Operator	Expression	Method
Less than	`q1 < q2`	`__lt__(q1, q2)`
Less than or equal to	`q1 <= q2`	`__le__(q1, q2)`
Equal to	`q1 == q2`	`__eq__(q1, q2)`
Not Equal to	`q1 != q2`	`__ne__(q1, q2)`
Greater than	`q1 > q2`	`__gt__(q1, q2)`
Greater than or equal to	`q1 >= q2`	`__ge__(q1, q2)`

3. Operator Overloading(6)

- Logical Operator



Operator	Expression	Method
AND	`q1 & q2`	`__and__(q1, q2)`
OR	`q1 | q2`	`__or__(q1, q2)`
XOR	`q1 ^ q2`	`__xor__(q1, q2)`
NOT	`~q1`	`__invert__()`