

Lecture 16

Chapter 9 Inheritance

9.1 Augmentation

9.2 Specialization

9.3 When Should Inheritance (Not) Be Used

Introduction

A core principle of object-oriented programming:

instances of a given class **support the same behaviors** and

are represented by a similar set of attributes.

So during the design a software we try to identify these underlying commonalities (which allows the greater re-use of code and minimizes the duplication of programming efforts).

Introduction

The concept of **inheritance** is a technique that allows us to define a new (child) class based upon an existing (parent) class.

The child class inherits all of the members of its parent class (thereby reducing the duplication of existing code).

Introduction

The child may:

- **introduce** one or more behaviors beyond those that are inherited, thereby **augmenting** the parent class
- **specialize** one or more of the inherited behaviors from the parent. It is accomplished by **providing an alternative definition** for the inherited method, thereby **overriding** the original definition

These techniques are not necessarily used in isolation.

A single class can serve as parent for many different child classes.

A single child class can inherit from multiple parent classes (**multiple inheritance**)

9.1 Augmentation

Recall our fraction class: see the handout.

Let's add a **mixed number class**, which will be a child of fraction class.

$q \frac{n}{d}$, where q is whole part and $\frac{n}{d}$ is fractional part

Then let's add two new behaviors:

- conversion from mixed numbers to improper fractions, and
- conversion from fractions to mixed numbers.

Where should these behaviors go?

9.1 Augmentation

A sketch of our new class:

```
class MixedNumber(Fraction):
    def __init__(self, quotient=0, num=0, denom=1):
        self._q=quotient
        Fraction.__init__(self, num, denom)
```

parent class

overriding the original constructor

Using the constructor of the parent class (and showing the difference between the `__init__` of the parent class and the `__init__` of the child class)

```
def __str__(self):
    # explain how to display the mixed numbers
```

```
# conversion of a mixed number to improper fraction
def mixed2fraction(self):
    return Fraction(self._q*self._d+self._n, self._d)
```

9.1 Augmentation

A sketch of our new class:

```
class MixedNumber(Fraction):
    def __init__(self, quotient=0, num=0, denom=1):
        self._q=quotient
        Fraction.__init__(self, num, denom)
```

! take care of the situation when a user gives improper fraction as a fractional part for the whole number, e.g. $1\frac{5}{4}$

```
    def __str__(self):
        # explain how to display the mixed numbers
```

If denominator is 0 we should display 'undefined';

If numerator is 0 and whole part is 0, we should display 0;

If numerator is 0, but the whole part is not, we should display whole part only;

Otherwise we display everything.

```
    # conversion of a mixed number to improper fraction
    def mixed2fraction(self):
        return Fraction(self._q*self._d+self._n, self._d)
```

No additional comments

9.1 Augmentation

See the program [mixed_numbers_class.py](#)

9.2 Specialization

Recall the overriding of the constructor of the parent Fraction class:

```
class MixedNumber(Fraction):  
    def __init__(self, quotient=0, num=0, denom=1):  
        self._q=quotient  
        Fraction.__init__(self, num, denom)
```

What else do we need to override (specialize)?
(which methods?)

9.2 Specialization

Recall the overriding of the constructor of the parent Fraction class:

```
class MixedNumber(Fraction):  
    def __init__(self, quotient=0, num=0, denom=1):  
        self._q=quotient  
        Fraction.__init__(self, num, denom)
```

What else do we need to override (specialize)?
(which methods?)

- Addition - will do

- Subtraction

- Multiplication

- Division - will do

- Negation and

- Negative Reciprocal

See program [mixed_numbers_class_updated.py](#)

9.3 When Should Inheritance (Not) Be Used

Let's summarize all that we did:

We built a class `MixedNumber` that has parent class `Fraction`.

Almost all behaviors of the parent class had to be overridden, and one: `n_invert(self)`, negative reciprocal, is not needed at all.

Almost everywhere we immediately converted mixed numbers to `Fraction`'s instances, performed operations, and then converted the result back to mixed number.

9.3 When Should Inheritance (Not) Be Used

The relationship between a parent and child class when using inheritance is often termed an **is-a relationship**, in that every mixed number “is” a fraction.

When one class is implemented using an instance variable of another, this is termed a **has-a relationship**.

In general, there is not always a clear-cut rule for when to use *inheritance* and when to use *has-a relationship*. The decision comes down to the number of potentially inherited behaviors that are undesirable versus the number of desirable ones that would need to be explicitly regenerated if using a has-a relationship.

9.3 When Should Inheritance (Not) Be Used

So it looks like the decision to make class `MixedNumber` a child class of `Fraction` is not quite right.

It looks better if we have an instance variable of `Fraction` class as an attribute of `MixedNumber` class.

Here is a sketch:

```
class MixedNumber():
    def __init__(self, w=0, num=0, denom=1):
        self._w=w
        self._f=Fraction(num, denom)

    def __str__(self):
        # explain how to display the mixed numbers

# conversion of a mixed number to improper fraction
def mixed2fraction(self):
    return Fraction(self._w, 1) + self._f
```

9.3 When Should Inheritance (Not) Be Used

See program [mixed_numbers_class_alternative.py](#)

This is much better and more appropriate.

Homework assignment № 10

Modify the program we discussed in class (the latest version):

- Write the functions for subtraction, multiplication and negation of mixed numbers
- take care of the negative mixed numbers (update all the methods/functions to accommodate negative mixed numbers)
- **Hint:** sign can be stored as a separate attribute or “attached” to the whole part, i.e.

$$\left(-2 \right) \left(\frac{3}{7} \right)$$

or

$$-2 \left(\frac{3}{7} \right)$$