

## 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 of 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***

**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*  
(reduces the duplication of existing code).

# Introduction

The child may:

- **introduce** one or more behaviors beyond those that are inherited (**augmenting** the parent class)
  - **specialize** one or more of the inherited behaviors from the parent (**provide an alternative definition** for the inherited method, i.e. **override** the original definition)
- these techniques are not necessarily used in isolation.
- a single class can serve as parent for many different child classes.
- single child class can inherit from multiple parent classes (**multiple inheritance**)

## 9.1 Augmentation

Recall our fraction class:

see program `fraction_class_final.py`

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

So, a mixed number will have one extra attribute: quotient or whole part (this is called **augmentation**)

## 9.1 Augmentation

Recall our fraction class:

see program `fraction_class_final.py`

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

So, a mixed number will have one extra attribute: quotient or whole part (this is called **augmentation**)

Here is a sketch of the class:

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

# 9.1 Augmentation

Here is a sketch of the class (more thorough):

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

What things should we take care of here?

```
    def __str__(self):  
        ...
```

How should the display method work?

## 9.1 Augmentation

Here is a sketch of the class (more thorough):

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

take care of the situation when a user gives improper fraction as a fractional part for the whole number, e.g.

```
    def __str__(self):  
        ...
```

$$1\frac{5}{4}$$

If denominator is 0, **display 'undefined'**;

If numerator is 0 and whole part is 0, **display 0**;

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

Otherwise we **display everything**.

# 9.1 Augmentation

```

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

        if num < denom: # fractional part is a proper fraction
            Fraction.__init__(self, num, denom)
        else: # fractional part is improper fraction
            self._q += num // denom
            r = num % denom
            Fraction.__init__(self, r, denom)

    def __str__(self):
        if self._d == 0: # if denominator is 0
            return 'Undefined'
        elif self._n == 0: # if the numerator is 0
            if self._q == 0: # and the whole part(quotient) is 0
                return str(0)
            else: # and the whole part (quotient) is not 0
                return str(self._q)
        else: # 2 3/4
            return str(self._q) + ' ' + str(self._n) + '/' + str(self._d)

```

*parent class* (points to Fraction)

*overriding the original constructor* (points to MixedNumber.\_\_init\_\_)

*Using the constructor of the parent class* (points to Fraction.\_\_init\_\_ calls)

## 9.1 Augmentation

We need two new behaviors/methods:

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

Where should these behaviors go?

## 9.1 Augmentation

We need two new behaviors/methods:

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

Where should these behaviors go?

1) conversion from mixed numbers to improper fractions can be a behavior of Mixed Number Class

## 9.1 Augmentation

We need two new behaviors/methods:

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

Where should these behaviors go?

1) conversion from mixed numbers to improper fractions can be a behavior of Mixed Number Class

2) conversion from fractions to mixed numbers can be a separate method/behavior

- for this we need to implement two more methods in the Fraction class: `getNum()` and `getDen()` **WHY?**

## ***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**

Multiplication

Power

Negative Reciprocal

Subtraction

Division - **will do**

Negation

## 9.2 Specialization: Addition Method

```
def __add__(self, other): # Addition
```

Idea: we'll be converting mixed numbers to improper fractions and adding them as fractions

There are many cases here:

Mixed Number + Mixed Number

Mixed Number + Fraction

Mixed Number + Decimal Number

Mixed Number + Integer

Mixed Number + Complex Number

...

We'll implement only first, second and fourth cases, in the rest of the cases we'll raise a **TypeError** message

## 9.2 Specialization: Addition Method

```
def __add__(self, other): # Addition
    a=self.mixed2fraction()
    print(self, ' is ', a)
    if isinstance(other, MixedNumber): # mixed + mixed
        b=other.mixed2fraction()
        print(other, ' is ', b)
        c=a+b
    elif isinstance(other, Fraction): # mixed + fract
        c=a+other
    elif isinstance(other, int): # mixed + integer
        b=Fraction(other, 1)
        c=a+b
    else: # otherwise
        print('Tried to perform the following
              operation:', self, ' + ', other)
        raise TypeError('These values cannot be added')
    return fraction2mixed(c)
```

See program [mixed\\_numbers\\_class2.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, it 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*

It looks like the decision to make class `MixedNumber` a child class of `Fraction` is not quite right. We could use **has-a** relation instead.

So another alternative: have an instance variable of `Fraction` class as an attribute of `MixedNumber` class, could be better.

## 9.3 When Should Inheritance (Not) Be Used

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):
        ''' displays mixed numbers '''

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

## ***9.3 When Should Inheritance (Not) Be Used***

See program [mixed\\_numbers\\_class\\_alternative.py](#)

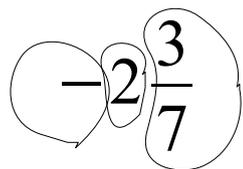
This is much better and more appropriate.

# Homework assignment

- Pages 327 - 328 / 9.1, 9.2, 9.3
- Write the functions/methods/behaviors for subtraction, multiplication and negation of mixed numbers

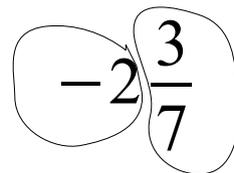
take care of the negative mixed numbers (update all the behaviors to accommodate negative mixed numbers)

**Hint:** sign can be stored as a separate attribute or “attached” to the whole part, i.e.



A diagram of the mixed number  $-2\frac{3}{7}$ . The minus sign is circled, and the whole number 2 is also circled. A larger circle encompasses both the minus sign and the 2, indicating that the sign is attached to the whole part.

or



A diagram of the mixed number  $-2\frac{3}{7}$ . The minus sign is circled, and the fraction  $\frac{3}{7}$  is also circled. A larger circle encompasses both the minus sign and the fraction, indicating that the sign is attached to the fraction part.