Writing a Fraction Class

- So far we have worked with floating-point numbers but computers store binary values, so not all real numbers can be represented precisely
- In applications where the precision of real numbers is important, we can use *rational numbers* to store exact values
  - This helps to reduce or eliminate round-off errors that can occur when performing arithmetic operations
  - A rational number is a number that can be expressed as a ratio of two integers: 7/8
  - The top value is called the *numerator* and the bottom value, which cannot be zero, is called the *denominator*

Designing the Fraction Class

- We want to use our rational numbers as we would use integers and floating point values
- Thus, our `Fraction` class must perform the following operations:
    1. Create a rational number
    2. Access the numerator and denominator values, individually
    3. Determine if the rational number is negative or zero
    4. Perform normal mathematical operations on two rational numbers (addition, subtraction, multiplication, division, exponentiation)
    5. Logically compare two rational numbers
    6. Produce a string representation of the rational number
- The objects of the Fraction class will be **immutable** because none of the operations modify the objects' instance variables

Required Data Attributes

- Because a rational number consists of two integers, we need two instance variables to store those values:

```
self._numerator = 0
self._denominator = 1
```

- At no time should the rational number be converted to a floating-point value or we will lose the precision gained from working with rational numbers

Representing Values Equivalently

- Signed values
  - Negative and positive rational numbers each have two forms that can be used to specify the corresponding value
  - Positive values can be indicated as 1/2 or –1/–2, and negative values as –2/5 or 2/–5
  - When performing an arithmetic operation or logically comparing two rational numbers, it will be much easier if we have a single way to represent a negative value
  - For simplicity, we choose to set only the numerator to a negative value when the rational number is negative, and both the numerator and denominator will be positive integers when the rational number is positive

Representing Values Equivalently

- Equivalent fractions
  - For example, 1/4 can be written as 1/4, 2/8, 16/64, or 123/492
  - It will be much easier to perform the operation if the number is stored in reduced form

The Constructor (1)

- Because `Fraction` objects are immutable, their values must be set when they are created. This requires parameter variables for both the numerator and denominator

```
def _ _init_ _(self, numerator, denominator) :
```

- The method must check for special cases:
  - Zero denominators
  - The number represents zero or a negative number

The Constructor

```
def _ _init_ _(self, numerator = 0, denominator = 1) :
    if denominator == 0 :
        raise ZeroDivisionError("Denominator cannot be zero.")
    if numerator == 0 :
        self._numerator = 0
        self._denominator = 1
    else :
        if (numerator < 0 and denominator >= 0 or
            numerator >= 0 and denominator < 0) :
           sign = -1
        else :
           sign = 1
```

The Constructor

```
a = abs(numerator)
b = abs(denominator)
while a % b != 0 :
    tempA = a
    tempB = b
    a = tempB
    b = tempA % tempB
self._numerator = abs(numerator)       # b * sign
self._denominator = abs(denominator)   #b
```

Testing the Constructor

```
frac1 = Fraction(1, 8) # Stored as 1/8
frac2 = Fraction(-2, -4) # Stored as 1/2
frac3 = Fraction(-2, 4) # Stored as -1/2
frac4 = Fraction(3, -7) # Stored as -3/7
frac5 = Fraction(0, 15) # Stored as 0/1
frac6 = Fraction(8, 0) # Error! exception is raised.
```

Comparing Fractions (1)

- In Python, we can define and implement methods that will be called automatically when a standard Python operator (+, *, ==, <) is applied to an instance of the class
- For example, to test whether two fractions are equal, we could implement a method:
  - `isequal()` and use it as follows:

```
if frac1.isequal(frac2) :
    print("The fractions are equal.")
```

Comparing Fractions (2)

- Of course, we would prefer to use the operator ==
- This is achieved by defining the special method:

  `_ _eq_ _():`

```
def _ _eq_ _(self, rhsValue) :
    return (self._numerator == rhsValue.numerator and
            self._denominator == rhsValue.denominator)
```

- Automatically calls this method when we compare two `Fraction` objects using the == operator:

```
if frac1 == frac2 : # Calls frac1._ _eq_ _(frac2)
    print("The fractions are equal.")
```

Special Methods

- Some special methods are called when an instance of the class is passed to a built-in function. For example, suppose you attempt to convert a `Fraction` object to a floating point number using the `float()` function:

```
x = float(frac1)
```

- Then the `_ _float_ _()` special method is called.
- Here is a definition of that method:

```
def _ _float_ _(self) :
    return self._numerator / self._denominator
```

Common Special Methods

Table 1 Common Special Methods

Expression	Method Name	Returns	Description
x + y	`__add__(self, y)`	object	Addition
x - y	`__sub__(self, y)`	object	Subtraction
x * y	`__mul__(self, y)`	object	Multiplication
x / y	`__truediv__(self, y)`	object	Real division
x // y	`__floordiv__(self, y)`	object	Floor division
x % y	`__mod__(self, y)`	object	Modulus
x ** y	`__pow__(self, y)`	object	Exponentiation
x == y	`__eq__(self, y)`	Boolean	Equal
x != y	`__ne__(self, y)`	Boolean	Not equal

Common Special Methods

Table 1 Common Special Methods

`x < y`	`__lt__(self, y)`	Boolean	Less than
`x <= y`	`__le__(self, y)`	Boolean	Less than or equal
`x > y`	`__gt__(self, y)`	Boolean	Greater than
`x >= y`	`__ge__(self, y)`	Boolean	Greater than or equal
`-x`	`__neg__(self)`	object	Unary minus
`abs(x)`	`__abs__(self)`	object	Absolute value
`float(x)`	`__float__(self)`	float	Convert to a floating-point value
`int(x)`	`__int__(self)`	integer	Convert to an integer value
`str(x)` `print(x)`	`__repr__(self)`	string	Convert to a readable string
`x = ClassName()`	`__init__(self)`	object	Constructor

Addition of Fractions

- All of the arithmetic operations that can be performed on a `Fraction` object should return the result in a new `Fraction` object
- For example, when the statement below is executed, `frac1` should be added to `frac2` and the result returned as a new `Fraction` object that is assigned to the `newFrac` variable

```
newFrac = frac1 + frac2
```

Fractional Addition

- From elementary arithmetic, you know that two fractions must have a common denominator in order to add them. If they do not have a common denominator, we can still add them using the formula:

$$\frac{a}{b} + \frac{c}{d} = \frac{d \cdot a + b \cdot c}{b \cdot d}$$

Defining the Method For Addition

```
def _ _add_ _(self, rhsValue) :
    num = (self._numerator * rhsValue._denominator +
       self._denominator * rhsValue._numerator)
    den = self._denominator * rhsValue._denominator
    return Fraction(num, den)
```

Logic: Less Than

- Note that $a \,/\, b < c \,/\, d$ when $d \cdot a < b \cdot c$. (Multiply both sides with $b \cdot d$.)
- Based on this observation, the less than operation is implemented by the `_ _lt_ _()` method as follows:

```
def _ _lt_ _(self, rhsValue) :
    return (self._numerator * rhsValue._denominator
            self._denominator * rhsValue._numerator)
```

Fraction.py

```python
class Fraction :
    ## Constructs a rational number initialized to zero or a user specified value.
    #  @param numerator the numerator of the fraction (default is 0)
    #  @param denominator the denominator of the fraction (cannot be 0)
    #
    def __init__(self, numerator = 0, denominator = 1) :
        # The denominator cannot be zero.
        if denominator == 0 :
            raise ZeroDivisi

        # If the rational nur
        if numerator == 0 :
            self._numerator
            self._denominato
```

```python
        # Otherwise, store the rational number in reduced form.
        else :
            # Determine the sign.
            if (numerator < 0 and denominator >= 0 or
                numerator >= 0 and denominator < 0) :
                sign = -1
            else :
                sign = 1

            # Reduce to smallest form.
            a = abs(numerator)
            b = abs(denominator)
            while a % b != 0 :
                tempA = a
                tempB = b
                a = tempB
                b = tempA % tempB

            self._numerator = abs(numerator) // b * sign
            self._denominator = abs(denominator) // b
```

Fraction.py

```
def __add__(self, rhsValue) :
   num = (self._numerator * rhsValue._denominator +
          self._denominator * rhsValue._numerator)
   den = self._denominator * rhsValue._denominator
   return Fraction(num, den)

## Subtracts a fraction from this fraction.
#  @param rhsValue the right-hand side fraction
#  @return a new Fraction object resulting from the subtraction
#
def __sub__(self, rhsValue) :
   num = (self._numerator * rhsValue._denominator -
          self._denominator * rhsValue._numerator)
   den = self._denominator * rhsValue._denominator
   return Fraction(num, den)
```

```
def __eq__(self, rhsValue) :
   return (self._numerator == rhsValue._numerator and
           self._denominator == rhsValue._denominator)
```

Fraction.py

```
  def __lt__(self, rhsValue) :
    return (self._numerator * rhsValue._denominator <
            self._denominator * rhsValue._numerator)

  ## Determines if this fraction is not equal to another fraction.
  #  @param rhsValue the right-hand side fraction
  #  @return True if the fractions are not equal
  #
  def __ne__(self, rhsValue) :
    return not self == rhsValue

  ## Determines if this fraction is less than or equal to another fraction.
  #  @param rhsValue the right-hand side fraction
  #  @return True if this fraction is less than or equal to the other
  #
  def __le__(self, rhsValue) :
    return not rhsValue < self

  ## Determines if this fraction is greater than another fraction.
  #  @param rhsValue the right-hand side fraction
  #  @return True if this fraction is greater than the other
  #
  def __gt__(self, rhsValue) :
    return rhsValue < self

  ## Determines if this fraction is greater than or equal to another fraction.
  #  @param rhsValue the right-hand side fraction
  #  @return True if this fraction is greater than or equal to the other
  #
  def __ge__(self, rhsValue) :
    return not self < rhsValue
```

Fraction.py

```
def __float__(self) :
   return self._numerator / self._denominator

## Gets a string representation of the fraction.
#  @return a string in the format #/#
#
def __repr__(self) :
   return str(self._numerator) + "/" + str(self._denominator)
```

Checking Type

- To ensure that variables are the correct type, Python provides the built-in `isinstance()` function that can be used to check the type of object referenced by a variable.
- For example, the constructor for the `Fraction` class requires two integers

```
class Fraction :
    def _ _init_ _(self, numerator, denominator) :
        if (not isinstance(numerator, int) or
            not isinstance(denominator, int)) :
           raise TypeError
              ("The numerator and denominator must be integers.")
```

Summary: Classes and Objects

- A class describes a set of objects with the same behavior
  - Every class has a public interface: a collection of methods through which the objects of the class can be manipulated
  - Encapsulation is the act of providing a public interface and hiding the implementation details
  - Encapsulation enables changes in the implementation without affecting users of a class

Summary: Variables and Methods

- An object's instance variables store the data required for executing its methods
- Each object of a class has its own set of instance variables
- An instance method can access the instance variables of the object on which it acts
- A private instance variable should only be accessed by the methods of its own class
- Class variables have a single copy of the variable shared among all of the instances of the class

Summary: Method Headers, Data

- Method Headers
  - You can use method headers and method comments to specify the public interface of a class
  - A mutator method changes the object on which it operates
  - An accessor method does not change the object on which it operates
- Data Representation
  - For each accessor method, an object must either store or compute the result
  - Commonly, there is more than one way of representing the data of an object, and you must make a choice
  - Be sure that your data representation supports method calls in any order

Summary: Constructors

- A constructor initializes the object's instance variables
- A constructor is invoked when an object is created
- The constructor is defined using the special method name: `_ _init_ _()`
- Default arguments can be used with a constructor to provide different ways of creating an object

Summary: Method Implementation

- The object on which a method is applied is automatically passed to the `self` parameter variable of the method
- In a method, you access instance variables through the `self` parameter variable

Summary: Testing Classes

- A unit test verifies that a class works correctly in isolation, outside a complete program
- To test a class, use an environment for interactive testing, or write a tester class to execute test instructions
- Determining the expected result in advance is an important part of testing

Summary: Object Tracing

- Object tracing is used to visualize object behavior
- Write the methods on the front of a card, and the instance variables on the back
- Update the values of the instance variables when a mutator method is called

Summary: Patterns for Classes

- An instance variable for the total is updated in methods that increase or decrease the total amount
- A counter that counts events is incremented in methods that correspond to the events
- An object can collect other objects in a list
- An object property can be accessed with a getter method and changed with a setter method
- If your object can have one of several states that affect the behavior, supply an instance variable for the current state

Summary: Patterns for Classes

- To model a moving object, you need to store and update its position

Summary: Object References

- An object reference specifies the location of an object
- Multiple object variables can contain references to the same object
- Use the `is` and `is not` operators to test whether two variables are aliases
- The `None` reference refers to no object

Summary: Defining Special Methods

- To use a standard operator with objects, define the corresponding special method
- Define the special `_ _repr_ _()` method to create a string representation of an object