18. Classes and Objects - Digging Deeper

18.1. Fractions

18.2. Objects are Mutable

18.3. Sameness

18.4. Arithmetic Methods

18.5. Glossary

18.6. Exercises

SavvyPython

View:

Textbook

Social

Search

User

Scratch Activecode

Help

This Chapter

18.1. Fractions

We have all had to work with fractions when we were younger. Or, perhaps you do a lot of cooking and have to manage measurements for ingredients. In any case, fractions are something that we are familiar with. In this chapter we will develop a class to represent a fraction including some of the most common methods that we would like fractions to be able to do.

A fraction is most commonly thought of as two integers, one over the other, with a line separating them. The number on the top is called the numerator and the number on the bottom is called the denominator. Sometimes people use a slash for the line and sometimes they use a straight line. The fact is that it really does not matter so long as you know which is the numerator and which is the denominator.

To design our class, we simply need to use the analysis above to realize that the state of a fraction object can be completely described by representing two integers. We can begin by implementing the Fraction class and the \_\_init\_\_ method which will allow the user to provide a numerator and a denominator for the fraction being created.

1 class Fraction:

2

​3 def \_\_init\_\_(self, top, bottom):

4

​5 self.num = top # the numerator is on top

6 self.den = bottom # the denominator is on the bottom

7

​8 def \_\_str\_\_(self):

9 return str(self.num) + "/" + str(self.den)

10

​11 def getNum(self):

12 return self.num

13

​14 def getDen(self):

15 return self.den

16

17 myfraction = Fraction(3, 4)

18 print(myfraction)

19

​20 print(myfraction.getNum())

21 print(myfraction.getDen())

22

Note that the \_\_str\_\_ method provides a “typical” looking fraction using a slash between the numerator and denominator. The figure below shows the state of myfraction. We have also added a few simple accessor methods, getNum and getDen, that can return the state values for the fraction.

SavvyPython

View:

Textbook

Social

Search

User

Scratch Activecode

Help

This Chapter

18.2. Objects are Mutable

We can change the state of an object by making an assignment to one of its instance variables. For example, we could change the numerator of the fraction by assigning a new value to self.num. Likewise, we could do the same thing for self.den.

One place where this type of modification makes sense is when we place a fraction in lowest terms. Lowest terms simply means that the numerator and denominator do not share any common factors. For example, 12/16 is a fraction but it is not in lowest terms since 2 can divide into both 12 and 16. We call 2 a common divisor. If we divide the numerator and the denominator by a common divisor, we get an equivalent fraction. If we divide by the greatest common divisor, we will get the lowest terms representation. In this case 4 would be the greatest common divisor and the lowest terms representation would be 3/4.

There is a very nice iterative method for computing the greatest common divisor of two integers. Try to run the function on a number of different examples.

1

def gcd(m, n):

2

while m % n != 0:

3

oldm = m

4

oldn = n

5

​

6

m = oldn

7

n = oldm % oldn

8

​

9

return n

10

​

11

print(gcd(12, 16))

12

​

Activity: 18.2.1 ActiveCode (fractions\_gcd)

Now that we have a function that can help us with finding the greatest common divisor, we can use that to implement a fraction method called simplify. We will ask the fraction “to put itself in lowest terms”.

The simplify method will pass the numerator and the denominator to the gcd function to find the greatest common divisor. It will then modify itself by dividing its num and its den by that value.

1 def gcd(m, n):

2 while m % n != 0:

3 oldm = m

4 oldn = n

5

​6 m = oldn

7 n = oldm % oldn

8

​9 return n

10

​11 class Fraction:

12

​13 def \_\_init\_\_(self, top, bottom):

14

​15 self.num = top # the numerator is on top

16 self.den = bottom # the denominator is on the bottom

17

​18 def \_\_str\_\_(self):

19 r eturn str(self.num) + "/" + str(self.den)

20

​21 def simplify(self):

22 common = gcd(self.num, self.den)

23

​24 self.num = self.num // common

There are two important things to note about this implementation. First, the gcd function is not a method of the class. It does not belong to Fraction. Instead it is a function that is used by Fraction to assist in a task that needs to be performed. This type of function is often called a helper function. Second, the simplify method does not return anything. Its job is to modify the object itself. This type of method is known as a mutator method because it mutates or changes the internal state of the object.

SavvyPython

View:

Textbook

Social

Search

User

Scratch Activecode

Help

This Chapter

18.3. Sameness

The meaning of the word same seems perfectly clear until you give it some thought and then you realize there is more to it than you expected.

For example, if you say, Chris and I have the same car, you mean that his car and yours are the same make and model, but that they are two different cars. If you say, Chris and I have the same mother, you mean that his mother and yours are the same person.

When you talk about objects, there is a similar ambiguity. For example, if two Fractions are the same, does that mean they represent the same rational number or that they are actually the same object?

We’ve already seen the is operator in the chapter on lists, where we talked about aliases. It allows us to find out if two references refer to the same object.

1

class Fraction:

2

​

3

def \_\_init\_\_(self, top, bottom):

4

​

5

self.num = top # the numerator is on top

6

self.den = bottom # the denominator is on the bottom

7

​

8

def \_\_str\_\_(self):

9

return str(self.num) + "/" + str(self.den)

10

​

11

​

12

myfraction = Fraction(3, 4)

13

yourfraction = Fraction(3, 4)

14

print(myfraction is yourfraction)

15

​

16

ourfraction = myfraction

17

print(myfraction is ourfraction)

18

​

Activity: 18.3.1 ActiveCode (fractions\_is)

Even though myfraction and yourfraction refer to the same rational number, they are not the same object.

../\_images/fractionpic2.png

If we assign myfraction to ourfraction, then the two variables are aliases of the same object.

../\_images/fractionpic3.png

This type of equality is called shallow equality because it compares only the references, not the contents of the objects. Using the == operator to check equality between two user defined objects will return the shallow equality result. In other words, the Fraction objects are equal (==) if they are the same object.

Of course, we could define equality to mean the fractions are the same in that they represent the same rational number. Recall from algebra that a/b = c/d is equivalent to a\*d = b\*c. Here is a boolean function that performs this check.

def sameRational(f1, f2):

return f1.getNum()\*f2.getDen() == f2.getNum() \* f1.getDen()

This type of equality is known as deep equality since it compares the values “deep” in the object, not just the reference to the object.

1 def sameRational(f1, f2):

2 return f1.getNum()\*f2.getDen() == f2.getNum() \* f1.getDen()

3

​4 class Fraction:

5

​6 def \_\_init\_\_(self, top, bottom):

7

​8 self.num = top # the numerator is on top

9 self.den = bottom # the denominator is on the bottom

10

​11 def \_\_str\_\_(self):

12 return str(self.num) + "/" + str(self.den)

13

​14 def getNum(self):

15 return self.num

16

​17 def getDen(self):

18 eturn self.den

19

​20

​21 myfraction = Fraction(3, 4)

22 yourfraction = Fraction(3, 4)

23 print(myfraction is yourfraction)

24 print(sameRational(myfraction, yourfraction))

Of course, if the two variables refer to the same object, they have both shallow and deep equality.

Beware of ==

“When I use a word,” Humpty Dumpty said, in a rather scornful tone, “it means just what I choose it to mean — neither more nor less.” Alice in Wonderland

Python has a powerful feature that allows a designer of a class to decide what an operation like == or < should mean. (We’ve just shown how we can control how our own objects are converted to strings, so we’ve already made a start!) We’ll cover more detail later. But sometimes the implementors will attach shallow equality semantics, and sometimes deep equality, as shown in this little experiment:

p = Point(4, 2)

s = Point(4, 2)

print("== on Points returns", p == s) # by default, == does a shallow equality test here

a = [2, 3]

b = [2, 3]

print("== on lists returns", a == b) # by default, == does a deep equality test on lists

This outputs:

== on Points returns False

== on lists returns True

So we conclude that even though the two lists (or tuples, etc.) are distinct objects with different memory addresses, in one case the == operator tests for deep equality, while in the case of points it makes a shallow test.

SavvyPython

View:

Textbook

Social

Search

User

Scratch Activecode

Help

This Chapter

18.4. Arithmetic Methods

We will conclude this chapter by adding a few more methods to our Fraction class. In particular, we will implement arithmetic. To begin, consider what it means to add two fractions together. Remember that you can only add fractions if they have the same denominator. The easiest way to find a common denominator is to multiply the two individual denominators together. Anything we do to the denominator needs to the done to the numerator. This gives us the following equation for fraction addition:

a/b + c/d = (ad + cb)/bd

Our add method will take a Fraction as a parameter. It will return a new Fraction representing the sum. We will use the equation shown above to compute the new numerator and the new denominator. Since this equation will not give us lowest terms, we will utilize a similar technique as was used in the simplify method to find the greatest common divisor and then divide each part of the new fraction.

def add(self,otherfraction):

newnum = self.num\*otherfraction.den + self.den\*otherfraction.num

newden = self.den \* otherfraction.den

common = gcd(newnum,newden)

return Fraction(newnum//common,newden//common)

You can try the addition method and then modify the fractions and retry.

1 def gcd(m, n):

2 while m % n != 0:

3 oldm = m

4 oldn = n

5

​6 m = oldn

7 n = oldm % oldn

8

​9 return n

10

​11 class Fraction:

12

​13 def \_\_init\_\_(self, top, bottom):

14

​15 self.num = top # the numerator is on top

16 self.den = bottom # the denominator is on the bottom

17

​18 def \_\_str\_\_(self):

19 return str(self.num) + "/" + str(self.den)

20

​21 def simplify(self):

22 common = gcd(self.num, self.den)

23

​24 self.num = self.num // common

One final modification to this method will be quite useful. Instead invoking the add method, we can use the addition operator “+”. This requires that we implement another special method, this time called \_\_add\_\_. The details of the method are the same.

def \_\_add\_\_(self, otherfraction):

newnum = self.num\*otherfraction.den + self.den\*otherfraction.num

newden = self.den \* otherfraction.den

common = gcd(newnum, newden)

return Fraction(newnum // common, newden // common)

However, now we can perform addition in the same manner that we are used to with other numeric data.

f1 = Fraction(1, 2)

f2 = Fraction(1, 4)

f3 = f1 + f2 # calls the \_\_add\_\_ method of f1

print(f3)

18.5. Glossary

deep copy

To copy the contents of an object as well as any embedded objects, and any objects embedded in them, and so on; implemented by the deepcopy function in the copy module.

deep equality

Equality of values, or two references that point to objects that have the same value.

shallow copy

To copy the contents of an object, including any references to embedded objects; implemented by the copy function in the copy module.

shallow equality

Equality of references, or two references that point to the same object.