Object-Oriented Programming and Abstraction

Jerry Cain
CS 106AX
November 6, 2023

slides leveraged from those constructed by Eric Roberts
The Principles of OOP

• Object-oriented programming (often abbreviated to OOP) was invented in Norway in the 1960s but was not adopted widely for more than a decade.

• Object-oriented programming is defined by two principles, both of which I mentioned on Friday during my discussion of classes and objects in Python:
  – *Encapsulation*—The principle that data values and the methods that manipulate them should be integrated into a single coherent structure called an object.
  – *Inheritance*—The idea that objects and the classes that those objects represent form hierarchies that allow new classes to share behavior with classes at higher levels in the hierarchy.

• Today’s lecture focuses on the use of encapsulation to define both the values and operations on rational numbers.
Rational Numbers

- Section 9.3 illustrates the idea of encapsulation by defining a class called `Rational` to represent *rational numbers*, which are simply the quotient of two integers.

- Rational numbers can be useful in cases in which you need exact calculation with fractions. Even if you use a `double`, the floating-point number 0.1 represented internally is actually an approximation. The rational number 1 / 10 is exact.

- Rational numbers support the standard arithmetic operations:

  \[
  \frac{a}{b} + \frac{c}{d} = \frac{ad + bc}{bd} \\
  \frac{a}{b} - \frac{c}{d} = \frac{ad - bc}{bd} \\
  \frac{a}{b} \times \frac{c}{d} = \frac{ac}{bd} \\
  \frac{a}{b} \div \frac{c}{d} = \frac{ad}{bc}
  \]
Implementing the **Rational** Class

- The next three slides show the initial version of the **Rational** class along with some brief annotations.

- As you read through the code, the following features are worth special attention:
  - *The constructor checks that rational numbers obey certain rules.* These rules are described in more detail in the text but include reducing the fraction to lowest terms.
  - *For now, operations are specified using the receiver syntax.* When you apply an operator to two **Rational** values, one of the operands is the receiver and the other is passed as an argument, as in

```
r1.add(r2)
```
# File: rational.py

"""
This module defines a class for representing rational numbers.
"""

import math

class Rational:

    # Implementation note
    # ---------------------
    # The Rational class ensures that every number has a unique
    # internal representation by guaranteeing that the following
    # conditions hold:
    #   1. The denominator must be greater than 0.
    #   2. The number 0 is always represented as 0/1.
    #   3. The fraction is always reduced to lowest terms.
def __init__(self, num, den=1):
    """Creates a new Rational object from num and den."""
    if den == 0:
        raise ValueError("Illegal denominator value")
    if num == 0:
        den = 1
    elif den < 0:
        den = -den
        num = -num
    g = math.gcd(abs(num), den)
    self._num = num // g
    self._den = den // g

def __str__(self):
    """Returns the string representation of this object."""
    if self._den == 1:
        return str(self._num)
    else:
        return str(self._num) + "/" + str(self._den)
The **Rational Class**

```python
def add(self, r):
    """Creates a new Rational by adding r to self.""
    return Rational(self._num * r._den + self._den * r._num,
                    self._den * r._den)

def sub(self, r):
    """Creates a new Rational by subtracting r from self.""
    return Rational(self._num * r._den - self._den * r._num,
                    self._den * r._den)

def mul(self, r):
    """Creates a new Rational by multiplying self by r.""
    return Rational(self._num * r._num, self._den * r._den)

def div(self, r):
    """Creates a new Rational by dividing self by r.""
    return Rational(self._num * r._den, self._den * r._num)
```
Simulating Rational Calculation

• The next slide works through all the steps in the calculation of a simple program that adds three rational numbers.

\[ \frac{1}{2} + \frac{1}{3} + \frac{1}{6} \]

• With rational arithmetic, the computation is exact. If you use floating-point arithmetic, the result looks like this:

```python
>>> 1/2 + 1/3 + 1/6
0.9999999999999999
```
def RationalSum():

def __str__(self):
    if self._den == 1:
        return str(self._num)
    else:
        return str(self._num) + "/" + str(self._den)

1/2 + 1/3 + 1/6 = 1
Overloading the Arithmetic Operators

• The receiver syntax used in the `RationalSum` program makes the program hard to read, particularly for people unfamiliar with object-oriented programming.

• The program would be much clearer if you could replace

  \[
  a\text{.add}(b)\text{.add}(c)
  \]

  with the more familiar expression

  \[
  a + b + c
  \]

• Unlike most modern languages, Python allows you to do just that. Each operator is associated with a special method name specifying how that operator should be implemented by the defining class. This technique is called *operator overloading*. 
Redefining Addition

• As an example, you can define addition for the `Rational` class by providing a definition for the `__add__` method. If you make use of the fact that the `Rational` class has an `add` method, the definition of the `__add__` operator looks like this:

```python
def __add__(self, rhs):
    return self.add(rhs)
```

• Although this simple implementation works, it is better for clients if one can mix types in an expression.

• As long as the `Rational` operand appears to the left of the `+` operator, the `__add__` method can define mixed-type addition by checking the type of `rhs`. If the `Rational` operand can appear on the right, you need to define the method `__radd__` as well. The code for overloading `+` in both directions appears on the next slide.
Overloading Addition on Both Sides

```python
def __add__(self, rhs):
    if type(rhs) is int:
        return self.add(Rational(rhs))
    elif type(rhs) is Rational:
        return self.add(rhs)
    else:
        return NotImplemented

def __radd__(self, lhs):
    if type(lhs) is int:
        return Rational(lhs).add(self)
    elif type(lhs) is Rational:
        return lhs.add(self)
    else:
        return NotImplemented
```
## Operator Methods in Python

<table>
<thead>
<tr>
<th>Method</th>
<th>Reverse Method</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td><code>__add__</code></td>
<td><code>__radd__</code></td>
<td>Redefines the + operator</td>
</tr>
<tr>
<td><code>__sub__</code></td>
<td><code>__rsub__</code></td>
<td>Redefines the - operator</td>
</tr>
<tr>
<td><code>__mul__</code></td>
<td><code>__rmul__</code></td>
<td>Redefines the * operator</td>
</tr>
<tr>
<td><code>__truediv__</code></td>
<td><code>__rtruediv__</code></td>
<td>Redefines the / operator</td>
</tr>
<tr>
<td><code>__floordiv__</code></td>
<td><code>__rfloordiv__</code></td>
<td>Redefines the // operator</td>
</tr>
<tr>
<td><code>__mod__</code></td>
<td><code>__rmod__</code></td>
<td>Redefines the % operator</td>
</tr>
<tr>
<td><code>__pow__</code></td>
<td><code>__rpow__</code></td>
<td>Redefines the ** operator</td>
</tr>
<tr>
<td><code>__neg__</code></td>
<td>(not applicable)</td>
<td>Redefines the unary - operator</td>
</tr>
<tr>
<td><code>__eq__</code></td>
<td>(symmetric)</td>
<td>Redefines the == operator</td>
</tr>
<tr>
<td><code>__ne__</code></td>
<td>(symmetric)</td>
<td>Redefines the != operator</td>
</tr>
<tr>
<td><code>__lt__</code></td>
<td>(inferred from &gt;)</td>
<td>Redefines the &lt; operator</td>
</tr>
<tr>
<td><code>__gt__</code></td>
<td>(inferred from &lt;)</td>
<td>Redefines the &gt; operator</td>
</tr>
<tr>
<td><code>__le__</code></td>
<td>(inferred from &gt;=)</td>
<td>Redefines the &lt;= operator</td>
</tr>
<tr>
<td><code>__ge__</code></td>
<td>(inferred from &lt;=)</td>
<td>Redefines the &gt;= operator</td>
</tr>
</tbody>
</table>
Type Abstraction

• One of the most important advantages of the object-oriented paradigm is the idea of type abstraction, in which the goal is to think about types in terms of their high-level behavior rather than their low-level implementation.

• In computer science, types that are defined by their behavior are called abstract data types or ADTs.

• Python includes several built-in abstract types, and you have already seen a few implementations of abstract types, such as the Rational we just discussed.

• We’ll spend the rest of lecture discussing strategies on how to define your own abstract data types.
Remembering Pig Latin

• One of the largest examples we covered while teaching JavaScript strings was a program that translated text from English to Pig Latin. We revisited that same program when we discussed Python’s support for strings.

• Both Pig Latin translators decomposed the problem into two functions: a `toPigLatin` function that divides the input into words and a `wordToPigLatin` function that translates a single word to its Pig Latin equivalent. The first phase of this operation is completely independent of Pig Latin domain.

• It would be useful to have a package that divides input strings into individual units that have integrity as a unit, as words do in English. Since the same idea applies in contexts beyond human languages, computer scientists use the term `token` to define these units. A library that returns individual tokens from an input source is called a `token scanner`. 
Designing a Token Scanner

- Section 12.2 in the Python reader describes a general library class called `TokenScanner`, which is implemented for several programming languages just as our graphics package is.

- The text also implements a small piece of that library that exports the following methods:

<table>
<thead>
<tr>
<th>Method</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td><code>scanner.setInput(str)</code></td>
<td>Sets the input for this scanner to the specified string or input stream.</td>
</tr>
<tr>
<td><code>scanner.hasMoreTokens()</code></td>
<td>Returns <code>true</code> if more tokens exist, and <code>false</code> at the end of the token stream.</td>
</tr>
<tr>
<td><code>scanner.nextToken()</code></td>
<td>Returns the next token from the token stream, and &quot; &quot; at the end.</td>
</tr>
<tr>
<td><code>scanner.ignoreWhitespace()</code></td>
<td>Tells the scanner to ignore whitespace characters.</td>
</tr>
</tbody>
</table>

- These methods are the primary `TokenScanner` methods you need for your next assignment.
A Simple TokenScanner Class

# File: tokenscanner.py

"""
This file implements a simple token scanner class.
"""

# A token scanner is an abstract data type that divides
# a string into tokens, which are strings of consecutive
# characters that form logical units. This simplified
# version recognizes two token types:
#
# 1. A string of consecutive letters and digits
# 2. A single character string

."
A Simple TokenScanner Class

class TokenScanner:

    def __init__(self, source=""): 
        self._source = source 
        self._nch = len(source) 
        self._cp = 0 
        self._ignoreWhitespaceFlag = False

    def setInput(self, source): 
        self._source = source 
        self._nch = len(source) 
        self._cp = 0
def nextToken(self):
    if self._ignoreWhitespaceFlag:
        self._skipWhitespace()
    if self._cp == self._nch:
        return ""
    token = self._source[self._cp]
    self._cp += 1
    if token.isalnum():
        while (self._cp < self._nch and
               self._source[self._cp].isalnum()):
            token += self._source[self._cp]
            self._cp += 1
    return token

def hasMoreTokens(self):
    if self._ignoreWhitespaceFlag:
        self._skipWhitespace()
    return self._cp < self._nch
def ignoreWhitespace(self):
    self._ignoreWhitespaceFlag = True

# Private methods

def _skipWhitespace(self):
    while (self._cp < self._nch and
           self._source[self._cp].isspace()):
        self._cp += 1
Using **TokenScanner** in PigLatin

```python
# File: PigLatin.py

from tokenscanner import PigLatin

def toPigLatin(line):
    result = ""
    scanner = TokenScanner(line)
    while scanner.hasMoreTokens():
        token = scanner.nextToken()
        if token.isalpha():
            token = wordToPigLatin(token)
        result += token
    return result
```
The End