CS 245: Principles of Programming Languages

Lab#7: OOP in Python

In this lab, we will learn how to use the OOP features of Python. Just like in Java, Python objects are defined using a class:

```
class <name>:
    <constructor(s)>
    <print method>
    <Accessors>
    <operations>
```

The class definition

To define a class for fraction, we do the following:

```
# Define a fraction class
# A fraction is represented as numerator/denominator
# The fraction is always kept normalized i.e.
# The fraction 4/6 will be represented as 2/3

class fraction:
```

Constructor(s): In Python, constructors are defined using the function **`__init(self, <args>)__`**. **`self`** is always the first argument. It refers to the current object (similar to **`this`** in Java). Thus, the constructor for **`fraction`** will be defined as:

```
  # Constructor
  def __init__(self, n, d):
    self.n = n        # numerator
    self.d = d        # denominator
    self.normalize()  # normalized
```

`self.n` and **`self.d`** are the two fields of **`fraction`** class. **`self.normalize()`** is used to normalize the fraction (i.e. convert a fraction like 4/6 to 2/3). It is defined as a method of the fraction class:

```
  def normalize(self):
    """ Normalizes the fraction """
    g = gcd(self.n, self.d)
    self.n = self.n // g
    self.d = self.d // g
    return self
```

The **`gcd()`** function is defined as a utility function outside the **`fraction`** class. We've already written this before, here it is:

```
# Utility function, not in the class
def gcd(a, b):
  """ Returns the greatest common divisor of a and b """
  if b == 0:
    return a
  else:
    return gcd(b, a % b)
```

Print Method: In Python, the print method is defined using the **`__str__(self)`** function. It is similar to the Java **`toString()`** method:

```
  def __str__(self):
    # print method to print a fraction as "n/d"
    return f"{self.n}/{self.d}"
```

Accessors: Accessor methods provide access to the fields of the fraction. Here we have two:

```
  # Accessors, in case they are needed
  def n(self):
    return self.n

  def d(self):
    return self.d
```

Operations: For this **`fraction`** class, we will define three operations: **`evaluate()`**, **`add()`**, and **`multiply()`**:

```
  def evaluate(self):
    """ Returns the float value of the fraction """
    return (float(self.n))/self.d

  def add(self, other):
    """ Adds another fraction to this fraction """
    if self.d == other.d:
      selfn = self.n + other.n
    else:
      self.n = self.n * other.d + other.n * self.d
      self.d = self.d * other.d
    self.normalize()

  def multiply(self, other):
    """ Multiplies another fraction to this fraction """
    self.n = self.n * other.n
    self.d = self.d * other.d
    self.normalize()
```

Note: `add()` and **`multiply()`** are implemented so the **`fraction`** object is *mutable*. What would it take to make it *immutable*?

Putting it all together: Here then is the complete definition of a **`fraction`** class, as presented above. You can save it in a file: **`fraction.py`**

```
# File: fraction.py
Define a fraction class
# A fraction is represented as numerator/denominator
# The fraction is always kept normalized i.e.
# The fraction 4/6 will be represented as 2/3
class fraction:
  # Constructor
  def __init__(self, n, d):
    self.n = n        # numerator
    self.d = d        # denominator
    self.normalize()  # normalized

  def __str__(self):
    # print method to print a fraction as "n/d"
    return f"{self.n}/{self.d}"

  # Accessors, in case they are needed
  def n(self):
    return self.n

  def d(self):
    return self.d

  def evaluate(self):
    """ Returns the float value of the fraction """
    return (float(self.n))/self.d

  def add(self, other):
    """ Adds another fraction to this fraction """
    if self.d == other.d:
      selfn = self.n + other.n
    else:
      self.n = self.n * other.d + other.n * self.d
      self.d = self.d * other.d
    self.normalize()

  def multiply(self, other):
    """ Multiplies another fraction to this fraction """
    self.n = self.n * other.n
    self.d = self.d * other.d
    self.normalize()
```

```
  def normalize(self):
    """ Normalizes the fraction """
    g = gcd(self.n, self.d)
    self.n = self.n // g
    self.d = self.d // g
    return self

# Utility function, not in the class
def gcd(a, b):
  """ Returns the greatest common divisor of a and b """
  if b == 0:
    return a
  else:
    return gcd(b, a % b)
```

Using the fraction class: A client Python program can be created (file **`use.py`**). The file **`fraction.py`** is now a module. One can import from a module in a couple of different ways:

```
from <module> import *
```

The above imports everything (denoted by *****) defined in **`<module>`**. Thus, we could define **`use.py`** as:

```
# Fil: use.py
from fraction import *

f1 = fraction(6, 9)
print("f1 = ", f1)
f2 = fraction(7, 11)
print("f2 = ", f2)
f3 = fraction(4, 6)
print("f3 = ", f3)

f2.add(f1)
print("f2 = f2 + f2 = ", f2)

f3.multiply(f1)
print("f3 = f3 * f1 = ", f3)
```

Save and run the file, **`use.py`** and you should see the following output:

```
f1 =  2/3
f2 =  7/11
f3 =  2/3
f2 = f2 + f2 =  43/33
f3 = f3 * f1 =  4/9
```

Alternately, and this is a better approach (why?), you can do the import as shown below:

```
import fraction

f1 = fraction.fraction(6, 9)
print("f1 = ", f1)
f2 = fraction.fraction(7, 11)
print("f2 = ", f2)
f3 = fraction.fraction(4, 6)
print("f3 = ", f3)

f2.fraction.add(f1)
print("f2 = f2 + f2 = ", f2)

f3.fraction.multiply(f1)
print("f3 = f3 * f1 = ", f3)

Try it.
```

Python Abstract Data Types? Unlike Java, where one can hide away the details in a class by declaring them private, there is no such facility in Python. All classes in Python are public! This implies that one can directly access any component of a class from any client. While this is possible, we encourage following good programming practice and try not to do this.

Testing individual Modules: As you learned in the last lab, you can use the **`__name__`** variable to detect if a module is being executed as the main program. This is useful since you can then write testing code in the module itself. Just add the following at the bottom of the **`fraction.py`** file:

```
if __name__ == '__main__':
    f1 = fraction(6, 9)
    print("f1 = ", f1)
    f2 = fraction(7, 11)
    print("f2 = ", f2)
    f3 = fraction(4, 6)
    print("f3 = ", f3)

    f2.add(f1)
    print("f2 = f2 + f2 = ", f2)

    f3.multiply(f1)
    print("f3 = f3 * f1 = ", f3)
```

Go ahead and try it.