1. If you download the recipe for a cake from the Internet, how many cakes do you have?

b) How many cakes can you make from this recipe?

2. In object-oriented languages like Python/C++/Java, etc., programmers can define classes (“recipes”) and can make (construct) as many objects (class instances) as needed. In object-oriented design (OOD), the programmer looks at the specifications for a new program to identify objects and their interactions.

- Objects are abstractions that are models of real-world things
- Object-based programming uses existing objects, and their methods to solve problems. The programmer knows:
  - How to instantiate a class to obtain an object (t = Turtle() or t = Turtle(400, 400))
  - Interface (set of methods) that can be used with objects of the class (t.up() or t.move(20, 15))
  - State of an object

A class definition is like a blueprint (recipe) for each of the objects of that class

- A class specifies a set of attributes and methods for the objects of that class
- The values of the attributes of a given object make up its state
- The behavior of an object depends on its current state and on the methods that manipulate this state
- The set of a class’s methods is called its interface

The general syntax of class definition is:

```python
class MyClass [ ( superClass1 [, superClass2 ]* ) ]:
    """Document comment which becomes the __.doc__ attribute for the class"
    def __init__(self, [param [, param]*]):
        """Document comment for constructor method with self be referencing to the object itself"
        #__init__body

        # defs of other class methods and assignments to class attributes

    # end class MyClass
```

Classes in Python have the following characteristics:

- all class attributes (data attributes and methods) are public by default, unless your identifier starts with a single underscores, e.g, self._numSides
- all data types are objects, so they can be used as inherited base classes
- most built-in operators (+, -, *, <, >, ==, etc.) can be redefined for a class. This makes programming with objects a lot more intuitive. For example suppose we have two Die objects: die1 & die2, and we want to add up their combined rolls. We could use accessor methods to do this:
  ```python
diceTotal = die1.getRoll() + die2.getRoll()
```
Here, the getValue method returns an integer (type int), so the ‘+’ operator being used above is the one for ints. But, it might be nice to “overload” the + operator by defining an __add__ method as part of the Die class, so the programmer could add dice directly as in:
  ```python
diceTotal = die1 + die2
```
- objects are passed by reference when used as parameters to functions
- all classes have a set of standard methods provided, but may not work properly (__str__, __doc__, etc.)
Consider the interface for a generalized Die class that can have any number of sides.

<table>
<thead>
<tr>
<th>Method</th>
<th>Example Usage</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td><strong>init</strong></td>
<td>myDie = Die(8)</td>
<td>Constructs a die with a specified number of sides and randomly rolls it (Default of 6 sides if no argument supplied)</td>
</tr>
<tr>
<td><strong>cmp</strong></td>
<td>if myDie == otherDie:</td>
<td>Allows the comparison operations (&gt;, &lt;, ==, etc.) to work correctly for Die objects. To implement this you need to return -1 if the left-hand-side die’s roll is less than the right-hand-side die’s roll, or return 0 if they are equal, or return +1 if the left-hand-side die’s roll is greater than the right-hand-side die’s roll. (see the Rational class for an example of <strong>cmp</strong>)</td>
</tr>
<tr>
<td><strong>add</strong></td>
<td>sum = myDie + otherDie</td>
<td>Allows the direct addition of Die objects, and returns the integer sum of there current values.</td>
</tr>
<tr>
<td><strong>str</strong></td>
<td>directly as:</td>
<td>Returns a string representation for the Die. By overriding the default <strong>str</strong> method the “print” statement will work correctly.</td>
</tr>
<tr>
<td></td>
<td>myDie.<strong>str</strong>()</td>
<td>Directly as: myDie.<strong>str</strong>()</td>
</tr>
<tr>
<td></td>
<td>str(myDie)</td>
<td>or indirectly as: print myDie</td>
</tr>
<tr>
<td>roll</td>
<td>myDie.roll()</td>
<td>Rolls the die randomly and return the value rolled</td>
</tr>
<tr>
<td>getRoll</td>
<td>myDie.getRoll()</td>
<td>Returns the current roll of the die</td>
</tr>
<tr>
<td>getSides</td>
<td>myDie.getSides()</td>
<td>Returns the number of sides on the die</td>
</tr>
<tr>
<td>show</td>
<td>myDie.show()</td>
<td>Displays the die’s value to standard output</td>
</tr>
</tbody>
</table>

Consider the following script and associated output:

```python
# testDie.py - script to test Die class
from die import Die
die1 = Die(100)
die2 = Die(100)
die3 = Die()

print 'die1 =', die1    #calls __str__
print 'die2 =', die2
print 'die3 =', die3

print 'die1.show() = ',
die1.show()
print 'die1.getRoll() = ', die1.getRoll()
print 'die1.roll() = ', die1.roll()
print 'die1.getRoll() = ', die1.getRoll()
print 'die2.getRoll() = ', die2.getRoll()
print 'die1 == die2: ', die1==die2
print 'die1 < die2: ', die1<die2
print 'die1 > die2: ', die1>die2
print 'die1 <= die2: ', die1<=die2
print 'die1 >= die2: ', die1>=die2
print 'die1 != die2: ', die1!=die2
print 'die1.__str__(): ', die1.__str__()

die1 = 59
die2 = 49
die3 = 1
die1.show() = 59
die1.getRoll() = 59
die1.roll() = 53
die1.getRoll() = 53
die2.getRoll() = 49
die1 == die2: False
die1 < die2: False
die1 > die2: True
die1 <= die2: False
die1 >= die2: True
die1 != die2: True
die1.__str__(): 53
```

Notice that the testDie script needed to import die. Having the Die class defined in its own file allows it to be developed separately, used easily, etc...

a) For the partial Die class, write the missing getSides and __add__ methods.
File: die.py
Description: Provides a simple Die class that allows for any number of sides

from random import randint

class Die:
    "Simple Die class that allows for any number of sides"
    def __init__(self, *args):
        "Constructor for any sided Die that takes an the number of sides as a parameter; if no parameter given then default is 6-sided."
        if len(args) == 0:
            self._numSides = 6
        elif len(args) == 1 and isinstance(args[0], int):
            self._numSides = args[0]
        else:
            print "Usage:  Die() or Die(numberOfSides)"
            return None

        self._currentRoll = randint(1, self._numSides)
    
    def __str__(self):
        "Overrides the standard method, converts a Die to a string"
        return str(self._currentRoll)

    def roll(self):
        "Causes a die to roll itself"
        self._currentRoll = randint(1, self._numSides)
        return self._currentRoll

    def getRoll(self):
        "Returns the value of current Die roll."
        return self._currentRoll

    def show(self):
        "Displays a Die by printing it"
        print self._currentRoll

    def __cmp__(self, rhs_Die):
        "Overrides the '__cmp__' operator for Dies, to allow for a deep comparison of two Dice"

        if self._currentRoll < rhs_Die._currentRoll:
            return -1
        elif self._currentRoll == rhs_Die._currentRoll:
            return 0
        else:
            return 1