Intro. to Computing

Name:_____

1) If you download a cake receipe from the Internet, how many cakes do you have?

2) A *class* definition is like a blueprint (receipe) for each of the objects of that class

- A class specifies a set of data attributes and methods for the objects of that class
- The values of the data 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 three most important features of Object-Oriented Programming (OOP) to simplify programs and make them maintainable:

1. *encapsulation* - restricts access to an object's data to access only by its methods
   - ⇨ helps to prevent indiscriminant changes that might cause an invalid object state (e.g., 6-side die with roll 8)
2. *inheritance* - allows one class (the *subclass*) to pickup data attributes and methods of other class(es) (the *parent's*)
   - ⇨ helps code reuse since the subclass can extend its parent class(es) by adding addition data and/or methods, or overriding (through polymorphism) a parent's methods
3. *polymorphism* - allows methods in several different classes to have the same names, but be tailored for each class
   - ⇨ helps reduce the need to learn new names for standard operations (or invent strange names to make them unique)

The general syntax of class definition is:

```
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 referencing the object """
            #__init__body


      # defs of other class methods and assignments to class attributes
```

end class MyClass

```
"""
File: simple_die.py

This module defines the Die class.
"""

from random import randint

class Die(object):
    """This class represents a six-sided die."""

    def __init__(self):
        """The initial face of the die."""
        self._currentRoll = randint(1, 6)

    def roll(self):
        """Resets the die's value to a random number
        between 1 and 6."""
        self._currentRoll = randint(1, 6)

    def getRoll(self):
        """Returns the face value of the die."""
        return self._currentRoll

    def __str__(self):
        """Returns the string representation of the die."""
        return str(self._currentRoll)
```

Classes in Python have the following characteristics:

- all class attributes (data attributes and methods) are *public* by default, unless the attribute 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:

  ```
  diceTotal = die1.getRoll() + die2.getRoll()
  ```

  Here, the `getRoll` 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:

  ```
  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.)

```
# testSimpleDie.py - script to test Die class

from simple_die import Die

die1 = Die()
die2 = Die()

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


print 'die1.getRoll() = ', die1.getRoll()
print 'die2.getRoll() = ', die2.getRoll()

die1.roll()
print 'After die1.roll(): die1.getRoll() = ', die1.getRoll()

help(Die)
```

```
>>>
die1 = 5
die2 = 6
die1.getRoll() =  5
die2.getRoll() =  6
After die1.roll(): die1.getRoll() =  3
Help on class Die in module simple_die:

class Die(__builtin__.object)
 |  This class represents a six-sided die.
 |
 |  Methods defined here:
 |
 |  __init__(self)
 |      The initial face of the die.
 |
 |  __str__(self)
 |      Returns the string representation
of the die.
```

a) Write code to create a list containing 100 Die objects.

b) Write code to sum the rolls on all 100 Die objects.

3) Consider the interface for a generalized AdvancedDie class that can have any number of sides.

Detail Descriptions of the AdvancedDie Class Methods		
Method	**Example Usage**	**Description**
`__init__`	`myDie = AdvancedDie(8)`	Constructs a die with a specified number of sides and randomly rolls it (Default of 6 sides if no argument supplied)
`__cmp__`	`if myDie == otherDie:`	Allows the comparison operations (>, <, ==, etc.) to work correctly for AdvancedDie objects.
`__add__`	`sum = myDie + otherDie`	Allows the direct addition of AdvancedDie objects, and returns the integer sum of there current values.

`__str__`	Directly as:
	`myDie.__str__()`
	`str(myDie)`
	or indirectly as:
	`print myDie`
		Returns a string representation for an `AdvancedDie`. Overrides the `Die __str__` method so the "print" statement will work correctly with an `AdvancedDie` object.

`roll`	`myDie.roll()`	Rolls the die randomly. Overrides the `Die roll` method.
`getRoll`	`myDie.getRoll()`	Returns the current roll of the die. Can use the `Die getRoll` method
`getSides`	`myDie.getSides()`	Returns the number of sides on the die

Consider the following script and associated output:

```
# testAdvancedDie.py - script to test AdvancedDie
class
from simple_die import Die
from advanced_die import AdvancedDie

die1 = AdvancedDie(100)
die2 = AdvancedDie(100)
die3 = Die()

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

print 'die1.getRoll() = ', die1.getRoll()
print 'die3.getRoll() = ', die3.getRoll()
die1.roll()
print 'After die1.roll(): 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__()
print 'die1.getSides() =', die1.getSides()

help(AdvancedDie)
```

```
die1 = Number of Sides=100 Roll=96
die2 = Number of Sides=100 Roll=7
die3 = 6
die1.getRoll() =  96
die3.getRoll() =  6
After die1.roll(): die1.getRoll() =  77
die2.getRoll() =  7
die1 == die2: False
die1 < die2: False
die1 > die2: True
die1 <= die2: False
die1 >= die2: True
die1 != die2: True
die1.__str__():  Number of Sides=100
Roll=77
die1.getSides() = 100
Help on class AdvancedDie in module advanced_die:

class AdvancedDie(simple_die.Die)
 | Advanced die class that allows for any number of sides
 |
 | Method resolution order:
 |     AdvancedDie
 |     simple_die.Die
 |     __builtin__.object
 |
 | Methods defined here:
 |
 | __add__(self, rhs_Die)
 |     Returns the sum of two dice rolls
```

The `testAdvancedDie.py` script only `import`ed the `Die` class for `die3`. To just create and use `AdvancedDie` objects we would NOT have needed to `import Die`.

```
"""
File:  advanced_die.py
Description: Provides a AdvancedDie class that allows for any number of sides
Inherits from the parent class Die in module simple_die
"""
from simple_die import Die
from random import randint

class AdvancedDie(Die):
    """Advanced die class that allows for any number of sides"""

    def __init__(self, numberOfSides = 6):
        """Constructor for any sided Die that takes an the number of sides
        as a parameter; if no parameter given then default is 6-sided."""
        # call Die parent class constructor
        Die.__init__(self)
        self._numSides = numberOfSides
        self._currentRoll = randint(1, self._numSides)

    def roll(self):
        """Causes a die to roll itself -- overrides Die class roll"""
        self._currentRoll = randint(1, self._numSides)

    def __cmp__(self, rhs_Die):
        """Overrides the '__cmp__' operator for Dies, to allow for
        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

    def __add__(self, rhs_Die):
        """Returns the sum of two dice rolls"""
        return self._currentRoll + rhs_Die.currentRoll

    def getSides(self):
        """Returns the number of sides on the die."""
        return self._numSides
```

a) What data attributes are inherited from the parent Die class?

b) What new data attributes are added as part of the subclass AdvancedDie?

c) Which Die class methods are used directly for an AdvancedDie object?

d) Which Die class methods are redefined/overridden by the AdvancedDie object?

e) Which methods are new to the AdvancedDie class and not in the Die class?