1. An alternative to functional-decomposition design is to use object-oriented design (OOD). For the following program, what objects would be useful and what methods (operations on the objects) should each support?

"Write a program to roll two 6-sided dice 1,000 times to determine the percentage of each outcome (i.e., sum both dice). Report the outcome(s) with the highest percentage." (You only need consider the program’s OOD)

2. Consider the Die and AdvancedDie classes from the Python Summary handout.
   a) What data attributes of AdvancedDie 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?
   f) If die1 and die2 are AdvancedDie objects, then the statement "if die1 == die2:"
      invokes the __eq__ method of AdvancedDie with die1 "passed" as self and die2 passed as rhs_Die.
      def __eq__(self, rhs_Die):
        """Overrides default __eq__ operator to allow for deep comparison of dice"""
        return self._currentRoll == rhs_Die._currentRoll

      What would the code be for AdvancedDie __le__ method to allow for the "if die1 <= die2:"
      statement?
      def __le__(self, rhs_Die):
        return self._currentRoll <= rhs_Die._currentRoll

   g) Good software engineering practice is to include precondition and postcondition comments on each
      method/function where the:
      • precondition - indicates what must be true for the method to work correctly. Typically, the precondition describes
        the valid values of the parameters. If the precondition is not satisfied, the method does not need to work correctly!
      • postcondition - describes the expected state after the method has executed.

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

      What precondition and postcondition comments should we add?
      PRECONDITION: sides is an integer
      POSTCONDITION: initial roll between 1 and # sides

   h) If a method/function has a precondition that is not met when invoked (e.g., die1 = AdvancedDie("six")), why
      should the method raise an error?
      Raise error immediately where the error occurs.
Classes: A class definition is like a blueprint (recipe) 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 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
```

File: simple_die.py
Description: This module defines a six-sided Die class.

```python
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)
```

Consider the following script to test the Die class and its associated output:

```python
# testDie.py - script to test Die class
from simple_die import Die
die1 = Die()
die2 = Die()
print('die1 = ', die1)  # calls __str__
print('die2 = ', die2)
print()
print('getRoll():', die1.getRoll())
print('getRoll():', die2.getRoll())
dl1 = die1.getRoll()  # prints (by calling __str__ )
dl2 = die2.getRoll()
print('str(die1): ', str(die1))
print('str(die2): ', str(die2))
print('die1 + die2: ', die1.getRoll() + die2.getRoll())

>>>
die1 = 2
die2 = 5
dl1 = 2
dl2 = 3
str(die1): 2
str(die2): 3
die1 + die2: 8
```
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._currentRoll`
- all data types are objects, so they can be used as inherited base classes
- 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.)
- 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 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:
  ```python
  diceTotal = die1 + die2
  ```

The three most important features of Object-Oriented Programming (OOP) to simplify programs and make them maintainable are:
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 a of roll 8)
2. **inheritance** - allows one class (the subclass) to pickup data attributes and methods of other class(es) (the parents)
   - helps code reuse since the subclass can extend its parent class(es) by adding addition data attributes 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)

Consider using inheritance to extend the Die class to a generalized AdvancedDie class that can have any number of sides. The interface for the AdvancedDie class are:

<table>
<thead>
<tr>
<th>Method</th>
<th>Example Usage</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td><strong>init</strong></td>
<td><code>myDie = AdvancedDie(8)</code></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>getRoll</td>
<td><code>myDie.getRoll()</code></td>
<td>Returns the current roll of the die (inherited from Die class)</td>
</tr>
<tr>
<td>getSides</td>
<td><code>myDie.getSides()</code></td>
<td>Returns the number of sides on the die (did not exist in Die class)</td>
</tr>
<tr>
<td>roll</td>
<td><code>myDie.roll()</code></td>
<td>Recolls the die randomly (By overriding the <code>roll</code> method of Die, an AdvancedDie can generate a value based on its # of sides)</td>
</tr>
<tr>
<td><strong>eq</strong></td>
<td>if <code>myDie == otherDie:</code></td>
<td>Allows <code>==</code> operations to work correctly for AdvancedDie objects.</td>
</tr>
<tr>
<td><strong>lt</strong></td>
<td>if <code>myDie &lt; otherDie:</code></td>
<td>Allows <code>&lt;</code> operations to work correctly for AdvancedDie objects.</td>
</tr>
<tr>
<td><strong>gt</strong></td>
<td>if <code>myDie &gt; otherDie:</code></td>
<td>Allows <code>&gt;</code> operations to work correctly for AdvancedDie objects.</td>
</tr>
<tr>
<td><strong>add</strong></td>
<td><code>sum = myDie + otherDie</code></td>
<td>Allows the direct addition of AdvancedDie objects, and returns the integer sum of their current roll values.</td>
</tr>
<tr>
<td><strong>str</strong></td>
<td>Directly as: <code>myDie.__str__()</code></td>
<td>Returns a string representation for the AdvancedDie. By overriding the <code>__str__</code> method of the Die class, so the “print” statement will work correctly with an AdvancedDie.</td>
</tr>
</tbody>
</table>

or indirectly as: `print myDie`
Consider the following script and associated output:

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

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

die1 = Number of Sides=100 Roll=32
die2 = Number of Sides=100 Roll=76
die3 = Number of Sides=6 Roll=5

die1.getRoll() = 32
die1.getSides() = 100

die1.getRoll() = 70
die2.getRoll() = 76
die1 == die2: False
die1 < die2: True
die1 > die2: False
die1 != die2: True

die1.getRoll() = 5

str(die1): Number of Sides=100 Roll=70
die1 + die2: 146

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:
```

Notice that the `testAdvancedDie.py` script needed to import `AdvancedDie`, but not the `Die` class.

The initial `AdvancedDie` objects can be visualized as:

![Global Memory Diagram]

```
Initial AdvancedDie objects

- _numSides: 100 _currentRoll: 32
- _numSides: 100 _currentRoll: 76
- _numSides: 6 _currentRoll: 5
```
The AdvancedDie class that inherits from the Die superclass.

```python
# File: advanced_die.py
Description: Provides a AdvancedDie class that allows for any number of sides
Inherits from the parent class Die in module die_simple

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, sides = 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."""
        Die.__init__(self)  # call Die parent class constructor
        self.numSides = sides
        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 __eq__(self, rhs_Die):
        """Overides default 'eq' operator to allow for deep comparison of Dice"
        return self.currentRoll == rhs_Die.currentRoll

    def __lt__(self, rhs_Die):
        """Overides default 'lt' operator to allow for deep comparison of Dice"
        return self.currentRoll < rhs_Die.currentRoll

    def __gt__(self, rhs_Die):
        """Overides default 'gt' operator to allow for deep comparison of Dice"
        return self.currentRoll > rhs_Die.currentRoll

    def __str__(self):
        """Returns the string representation of the AdvancedDie.""
        return 'Number of Sides='+str(self.numSides)+', Roll='+str(self.currentRoll)

    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 better AdvancedDie class would check and enforce preconditions by raising appropriate exceptions (e.g., Exception, TypeError, IndexError, KeyError, ValueError, etc.). For example the __init__:

```python
class AdvancedDie(Die):  
    def __init__(self, sides = 6):
        """Constructor for any-sided Die that takes the number of sides as a parameter; if no parameter is given then default is 6-sided.
        Precondition: sides is an integer >= 1
        Postcondition: current roll of the die between 1 and sides
        Raises: TypeError if sides in not an integer and ValueError if sides < 1"
        # Check preconditions
        if not isinstance(sides, int):
            raise TypeError("Die sides must be an integer!")
        if sides < 1:
            raise ValueError("Die sides must be a positive integer!")

        Die.__init__(self)  # call Die parent class constructor
        self.numSides = sides
        self.currentRoll = randint(1, self.numSides)
```
3. General “Algorithmic-Complexity Analysis” terminology:

**problem** - question we seek an answer for, e.g., "What is the sum of all the items in a list/array?"

**parameters** - variables with unspecified values

**problem instance** - assignment of values to parameters, i.e., the specific input to the problem

```
myList: [5 1 2 3 4 5 6]
```

(numer of elements) n: 7

**sum:**

**algorithm** - step-by-step procedure for producing a solution

**basic operation** - fundamental operation in the algorithm (i.e., operation done the most). Generally, we want to derive a function for the number of times that the basic operation is performed related to the **problem size**.

**problem size** - input size. For algorithms involving lists/arrays, the problem size is the number of elements (“n”).

**Big-oh notation (O(·))** - As the size of a problem grows (i.e., more data), how will our program’s run-time grow.

Consider the following `sumList` function.

```
def sumList(myList):
    """Returns the sum of all items in myList"""
    total = 0
    for item in myList:
        total = total + item
    return total
```

a) What is the basic operation of `sumList` (i.e., operation done the most)?

b) What is the problem size of `sumList`?

c) If n is 10000 and `sumList` takes 10 seconds, how long would you expect `sumList` to take for n of 20000?

d) What is the big-oh notation for `sumList`?

4. Consider the following `someLoops` function.

```
def someLoops(n):
    total = 0
    for i in range(n):
        for j in range(n):
            total = total + i + j
    return total
```

```
<table>
<thead>
<tr>
<th>Execution flow</th>
<th>i = 0</th>
<th>i = 1</th>
<th>i = 2</th>
<th>i = n-1</th>
</tr>
</thead>
<tbody>
<tr>
<td>j = 0 to n-1</td>
<td>loops n times</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>j = 0 to n-1</td>
<td>loops n times</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>j = 0 to n-1</td>
<td>loops n times</td>
<td></td>
<td></td>
<td></td>
</tr>
</tbody>
</table>
```

a) What is the basic operation of `someLoops` (i.e., operation done the most)?

b) How many times will the basic operation execute as a function of n?

c) What is the big-oh notation for `someLoops`?

d) If we input n of 10000 and `someLoops` takes 10 seconds, how long would you expect `someLoops` to take for n of 20000?