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 boths dice). Report the outcome(s) with the highest percentage.” (You only need consider the program’s OOD)

2. Die: `roll`, `getRoll` Tally Sheet: `addLabel`, `increment`

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? `currentRoll` `numSides`
   b) What new data attributes are added as part of the subclass AdvancedDie? `numSides`
   c) Which Die class methods are used directly for an AdvancedDie object? `getRoll`
   d) Which Die class methods are redefined/overridden by the AdvancedDie object? `__str__` `__eq__` `__init__` `roll` `add` `getSides`
   e) Which methods are new to the AdvancedDie class and not in the Die class? `__eq__` `__init__` `__str__` `add` `getSides`
   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`.
      ```python
      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?
      ```python
      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:
   ```python
   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._init_(self) # call Die parent class constructor
           self._numSides = sides
           self._currentRoll = randint(1, self._numSides)
   ```
   What precondition and postcondition comments should we add?
   **Precondition:** sides parameter must positive integer
   **Postcondition:** Die with specified sides created with initial roll between 1 and # of 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?
      ```python
      if not isinstance(sides, int):
          raise TypeError("sides must be an integer")
      if sides <= 0:
          raise ValueError("sides must be positive")
      ```
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:  0  1  2  3  4  5  6  
         |  |  |  |  |  | 
         5 10 2 15 20 1 11

(number of elements) n: 7
```

- sum: 64

- 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.

```python
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)? \( \text{add} \)

b) What is the problem size of `sumList`? \( \text{len}(myList) \equiv \text{"n"} \)

c) If n is 10000 and `sumList` takes 10 seconds, how long would you expect `sumList` to take for n of 20000? \( 20 \text{ sec} \)

d) What is the big-oh notation for `sumList`? \( O(n) \equiv \text{"linear time"} \)

4. Consider the following `someLoops` function.

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

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 \( n \)? \( n \times n = n^2 \)

c) What is the big-oh notation for `someLoops`? \( O(n^2) \)

d) If we input n of 10000 and `someLoops` takes 10 seconds, how long would you expect `someLoops` to take for n of 20000? \( 40 \text{ sec} \)