Tasks of Graduation Party:

- Pick date

  ↕

- Location
- Invitations/Guest list
- Food <
- Decorations <

```
twoCat = 0
three Cat = 0


outcome=6

if outcome == 2:
    twoCat = twoCat +1
elif outcome == 3:
    three Cat += 1
        ⋮
```

Let's use a data structure to hold multiple counts. outcomeCounts list. in 1(a).

As our programs get bigger it becomes important to design them first, so let's consider Chapter 9 Building Bigger Programs. *Top-down design* (*hierarchical decomposition*) starts with developing a list of high-level *requirements* and high-level tasks. The high-level tasks are refined into smaller, but more specific subtasks. These subtasks might need to be further refined into even more detailed sub-subtasks, etc. The goal is to keep refining until all subtasks are easily implemented as a simple function with a "few" programming statements (I like to use the rule of thumb of ~25 lines containing a single loop).

You also need to decide how your data is represented (variables, lists, dictionaries) and passed between the subtasks. I find it helpful to think about how I would solve the same problem "by hand" and model that behavior.

1. Consider the problem:

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

a) Describe how you would solve this problem by hand -- including what values you would need to keep track of.

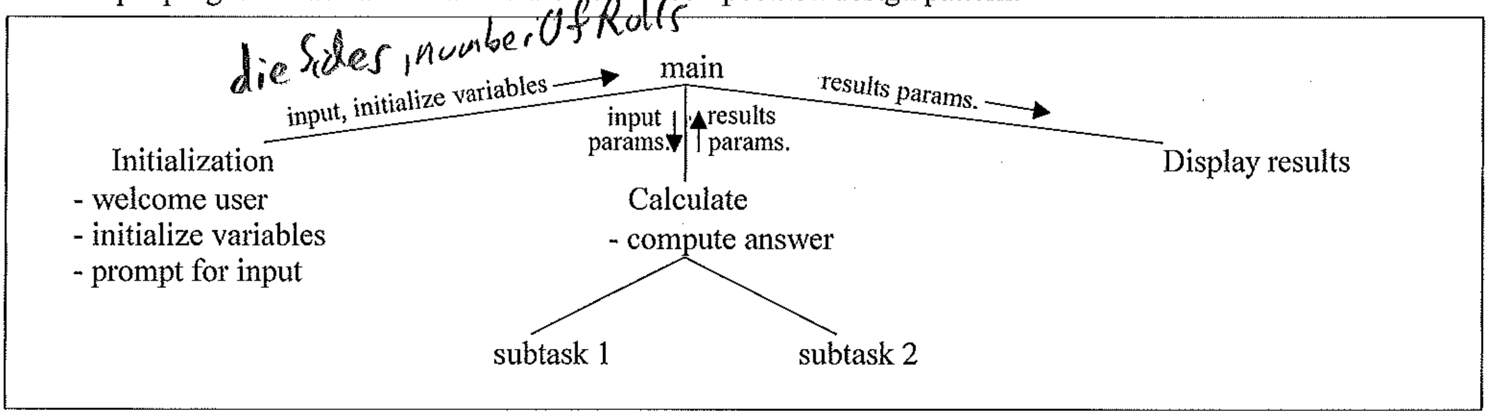
outcomeCounts: 1 2 3 4 5 6 7 8 9 10 11 12

~~1~~ 2

rolls

2

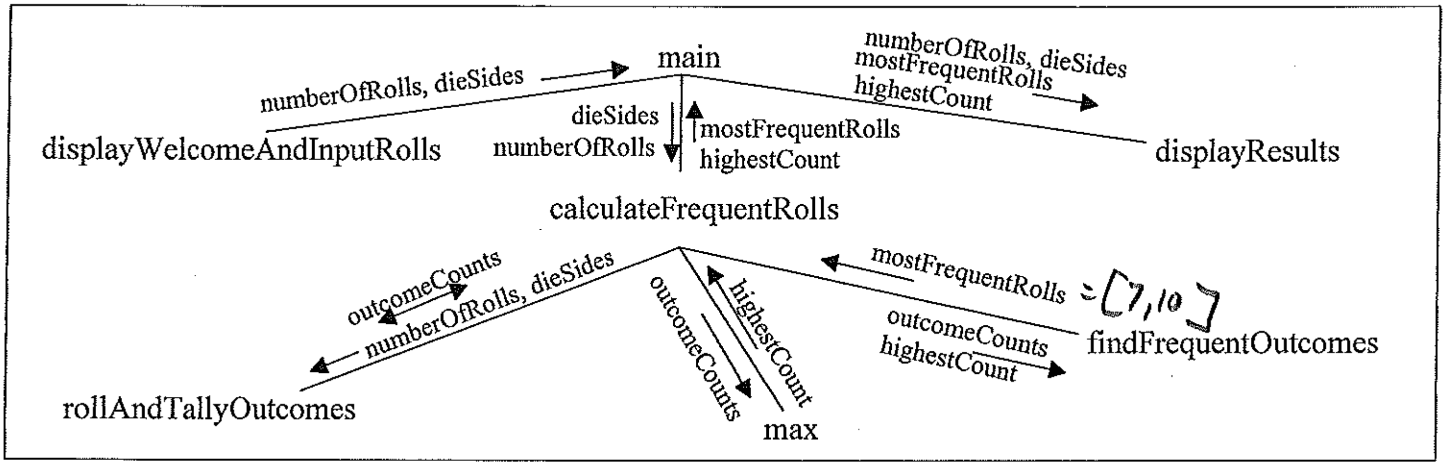
Most simple programs have a similar hierarchical-decomposition design pattern:



b) Customize the diagram for the dice problem by briefly describing what each function does and what parameters are passed.

see page 2

My hierarchical-decomposition of this problem is:



`main` - provides an outline of program by calling top-level functions
`displayWelcomeAndInputRolls` - Displays welcome message for the user. Gets and returns the number of dice rolls and number of sides on each die from the user.
`calculateFrequentRolls` - Rolls the dice the correct number of times, tallies the outcomes, and returns a list of outcomes with the highest count and highest count.
`rollAndTallyOutcomes` - Rolls the dice the correct number of times and tallies the outcomes. Returns a list of tallies with the index being the outcome.
`max` - built-in function to return the largest item in an iterable data structure like a list.
`findFrequentOutcomes` - Returns a list of outcomes with the highest count.
`displayResults` - Displays the outcome(s) with the highest percentage.

Consider running the program with only 10 dice rolls instead of 1,000. The program output with some extra debugging prints showing the two Python lists used: outcomeCounts and mostFrequentRolls.

```
This programs rolls two 6-sided dice 10 times to
determine the outcome(s) with the highest percentage.
                 0   1   2   3   4   5   6   7   8   9   10  11  12
outcomeCounts: [0, 0, 1, 0, 2, 1, 0, 3, 0, 0, 3, 0, 0]
mostFrequentRolls: [7, 10] and highestCount: 3
The highest percentage is 30.0% for outcome(s): 7 10
```

Implementation and testing can proceed from the *bottom-up* (i.e., simplest subtasks first) and integrated together.

c) Complete the `rollAndTallyOutcomes` function using `from random import randint`
The `randint` function call to generate a random value between 5 and 10 would be: `value = randint(5,10)`

```
                              10            6        [0,0,0, . . 0]
def rollAndTallyOutcomes(numberOfRolls, dieSides, outcomeCounts):
      for rolls in range(numberOfRolls):
          outcome = randint(1,dieSides) + randint(1,dieSides)
          outcomeCounts[outcome] += 1
```

d)

```
                                                    7      10
                       3               [            3      3      ]
def findFrequentOutcomes(highestCount, outcomeCounts):
    mostFrequentRolls = []
    for index in range(2, len(outcomeCounts)):
        if outcomeCounts[index] == highestCount:
            mostFrequentRolls.append(index)
    return mostFrequentRolls
```