1. The print function has optional *keyword arguments* which can be listed last that modify its behavior. The print function syntax: `print(value,..., sep=' ', end='
', file=sys.stdout)`

   a) Predict the expected output of each of the following.

   **Program**
   ```python
   print('cat', 5, 'dog')
   print()
   print('cat', 5, end='!')
   print(', horse')
   print('cow')
   ```

   **Expected Output**
   ```
   cat 5 u dog
   </br>
   cat 5 u horse
   cow
   ```

   **Program**
   ```
   print ('cat', 5, 'dog', end='\n', sep='23')
   print ('cat', 5, 'dog', sep='23' ('horse')
   print ('cat', 5, 'dog', sep='>')*3)
   ```

   **Expected Output**
   ```
   cat23523dog# 75 to be last
   error becaused named parameter need
   ```

2. Review of assignment statements. Predict the output of the following programs.

   ```python
   a = 123
   b = a
   a += 1
   print ('a is', a)  
   print ('b is', b)
   ```

   ```
   c = ['cat', 'dog']
   d = c.
   c.append('cow')
   print('c is', c)
   print('d is', d)
   ```

   Most simple programs have a similar functional-decomposition design pattern (IPO - Input, Process, Output):

   ```
   main
   ```

   ```
   input, initialize variables
   ```

   ```
   main
   ```

   ```
   input params
   ```

   ```
   results params.
   ```

   ```
   Output
   ```

   ```
   - display results
   ```

   ```
   Process
   ```

   ```
   - compute answer
   ```

   ```
   subtask 1
   ```

   ```
   subtask 2
   ```

   ```
   """ Simple IPO program to sum a list of numbers. """
   def main():
       label, values = getInput()
       total = sum(values)
       displayResults(label, total)

   def getInput():
       """ Get label and list of values to sum. """
       label = input("What are we summing? ")
       numberOfValues = int(input("How many values are there? "))
       values = []
       for i in range(numberOfValues):
           values.append(eval(input("Enter the next number: ")))
       return label, values

   def displayResults(label, total):
       """ Display sum of values. """
       print("The sum of", label, "values is", total)

   main()  # starts the main function running

   What are we summing? money
   How many values are there? 4
   Enter the next number: 10
   Enter the next number: 20
   Enter the next number: 30
   Enter the next number: 50
   The sum of money values is 110
When a function is called, a call-frame is pushed onto the run-time stack part of memory. A call-frame contains information about the function: (1) return address -- where function was called from, (2) formal parameters, and (3) local variables -- temporary values created inside the function.

```python
def main():
    label, values = getInput()
    total = sum(values)
    displayResults(label, total)

def getInput():
    """Get label and list of values to sum."""
    label = input("What are we summing?")
    numberOfValues = int(input("How many values are there? "
                              values = []
    for i in range(numberOfValues):
        values.append(eval(input("Enter the next number: ")))
    return label, values

def displayResults(label, total):
    """Display sum of values."""
    print("The sum of", label, "values is", total)
```

```
start
here
* main()  # starts the main function running
```

### Sequence of snapshots of the run-time stack during execution of program

<table>
<thead>
<tr>
<th>At start of main</th>
<th>At start of getInput</th>
<th>At end of getInput</th>
<th>After returning to (***) from getInput</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td>r.s.</td>
<td>r.a.</td>
<td>r.a.</td>
</tr>
<tr>
<td></td>
<td>label</td>
<td>label</td>
<td>(**)</td>
</tr>
<tr>
<td></td>
<td>numberOfValues</td>
<td>numberOfValues</td>
<td>(**)</td>
</tr>
<tr>
<td></td>
<td>values</td>
<td>values</td>
<td>&quot;money&quot;</td>
</tr>
<tr>
<td></td>
<td>i</td>
<td>i</td>
<td>(**)</td>
</tr>
<tr>
<td></td>
<td>r.a.</td>
<td>r.a.</td>
<td>(**)</td>
</tr>
<tr>
<td></td>
<td>label</td>
<td>label</td>
<td>&quot;money&quot;</td>
</tr>
<tr>
<td></td>
<td>values</td>
<td>values</td>
<td>[10, 20, 30, 50]</td>
</tr>
<tr>
<td></td>
<td>total</td>
<td>total</td>
<td></td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>After calling and returning</th>
<th>At start of displayResults</th>
<th>After returning to (*****) from displayResults</th>
<th>After returning to (*) from main</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td>r.a.</td>
<td>r.a.</td>
<td>r.a.</td>
</tr>
<tr>
<td></td>
<td>label</td>
<td>label</td>
<td>(**)</td>
</tr>
<tr>
<td></td>
<td>total 110</td>
<td>total 110</td>
<td>(**)</td>
</tr>
<tr>
<td></td>
<td>&quot;money&quot; [10, 20, 30, 50]</td>
<td>&quot;money&quot; [10, 20, 30, 50]</td>
<td>(**)</td>
</tr>
</tbody>
</table>
3. Design 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) How would you solve this problem "by hand" without a computer?

```
2 3 4 5 8    12
1
```

b) What built-in Python module/function can be used to simulate a 6-sided die?

```python
from random import randint
random.randint(1, 6)
```

c) What built-in data structure(s) could be used to tally the 1,000 roll outcomes?

```
dictionary
(1: 0, 2: 1, 3: 3, 4: 1, 5: 1, 6: 8, 7: 12)
```

d) Customize the functional-decomposition diagram for the dice problem by briefly describing what each function does and what parameters are passed.

(see label for my solution)
Objective: To practice writing Python code.

To start the lab: Download and unzip the file lab1.zip from http://www.cs.uni.edu/~fienup/cs1520s20/labs/lab1.zip

Part A: In the folder lab1, open the diceOutcomes.py program in IDLE. (Right-click on diceOutcomes.py | Edit with IDLE) It contains a partial program we started to discuss in class to solve 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."

I decided to functional-decomposition this problem as:

```
main - provides an outline of program by calling top-level functions
welcomeAndInputRolls - Displays welcome message for the user. Gets and returns the number of dice rolls 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.
findOutcomes - 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 program rolls two 6-sided dice many times to determine the outcome(s) with the highest percentage.

How many times would you like to roll the pair of dice? 10

outcomeCounts: [0, 0, 1, 0, 2, 1, 0, 0, 3, 0, 0, 3, 0, 0]
mostFrequentRolls: [7, 10] and highestCount: 3
The highest percentage is 30.0 for outcome(s): 7 10

Your task for lab 1 is to complete the code for the `rollAndTallyOutcomes` and `findOutcomes` functions.

After you have working code, raise your hand and demonstrate your code.

If you complete all parts of the lab, nothing needs to be turned in for this lab. If you do not get done today, then show me the completed lab in next week's lab period. When done, remember save your program to a USB drive, email to yourself, Google drive, etc.