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.

<table>
<thead>
<tr>
<th>Program</th>
<th>Expected Output</th>
</tr>
</thead>
<tbody>
<tr>
<td><code>print('cat',5,'dog')</code></td>
<td><code>cat 5 dog</code></td>
</tr>
<tr>
<td><code>print()</code></td>
<td></td>
</tr>
<tr>
<td><code>print('cat',5,end='')</code></td>
<td><code>cat 5 horse</code></td>
</tr>
<tr>
<td><code>print('cow')</code></td>
<td><code>cow</code></td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>Program</th>
<th>Expected Output</th>
</tr>
</thead>
<tbody>
<tr>
<td><code>print ('cat',5,'dog',end='#',sep='23')</code></td>
<td><code>cat 23 5 dog #</code></td>
</tr>
<tr>
<td><code>print ('cat',5,'dog'(sep='23'),'horse')</code></td>
<td><code>error</code></td>
</tr>
<tr>
<td><code>print ('cat',5,'dog',sep='1*x')</code></td>
<td><code>cat 5 5 dog</code></td>
</tr>
</tbody>
</table>

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

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

```python
c = ['cat', 'dog']
d = c
c += ['fish']
print('c is', c)
print('d is', d).
```

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

```
+++ Simple IPO program to sum a list of numbers. +++
def main():
    label, values = get_input()
    total = sum(values)
    display_results(label, total)

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

def display_results(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)
```

### 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>r.a. label ?</td>
<td>r.a. label ?</td>
<td>r.a. label ?</td>
<td>&quot;money&quot;</td>
</tr>
<tr>
<td>numberOfValues ?</td>
<td>numberOfValues ?</td>
<td>numberOfValues 4</td>
<td>[10, 20, 30, 50]</td>
</tr>
<tr>
<td>values ?</td>
<td>values ?</td>
<td>values 4</td>
<td></td>
</tr>
<tr>
<td>total ?</td>
<td>total ?</td>
<td>total ?</td>
<td>&quot;money&quot;</td>
</tr>
</tbody>
</table>

After calling and returning returning to (****) from sum

<table>
<thead>
<tr>
<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>r.a. label (**)</td>
<td>r.a. label (**)</td>
<td>r.a. label (**)</td>
</tr>
<tr>
<td>total 110</td>
<td>&quot;money&quot;</td>
<td>(10, 20, 30, 50)</td>
</tr>
<tr>
<td>values (10, 20, 30, 50)</td>
<td>values (10, 20, 30, 50)</td>
<td>values (10, 20, 30, 50)</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.

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

Data Structure choices:

1. Dictionary (key: value pairs)
   - keys would be outcomes
   - values would be counts

2. Outcome Counts List - Initially all 0's
   - outcome = randint(1, 6) + randint(1, 6)
   - outcome counts [outcome] += 1

Data Structures (CS 1520) Lecture 1 Name: