1. **Python 3.x vs. 2.x Changes:**
   - The `print` statement has been replaced with a `print()` function, with keyword arguments to replace most of the special syntax of the old `print` statement. New function syntax:
     ```python
     print(value,...,sep=' ',end='
', file=sys.stdout)
     ```
   - **a) Predict the expected output of each of the following.**

     | Version 2.x  | Version 3.x                        | Expected Output |
     |--------------|-----------------------------------|-----------------|
     | print 'cat',5,'dog' | print('cat',5,'dog') |                 |
     | print         | print()                          |                 |
     | print 'cat',5,  | print('cat',5,end='' )          |                 |
     | print 'horse'  | print(' horse')                  |                 |
     | print 'cow'    | print('cow')                     |                 |

     | Version 3.x                        | Expected Output |
     |-----------------------------------|-----------------|
     | print ('cat',5,'dog',sep='23',end='#') |                 |
     | print ('cat',5,'dog',end='#',sep='23')  |                 |
     | print ('cat',5,'dog',sep='23','horse') |                 |
     | print ('cat',5,'dog',sep='>']*3)      |                 |

   - The `range()` now behaves like `xrange()` of version 2.x. The `xrange()` function no longer exists in version 3.
   - `raw_input()` was renamed to `input()`. That is, the new `input()` function reads a line from sys.stdin and returns it as a string with the trailing newline stripped. It raises `EOFError` if the input is terminated prematurely. To get the old behavior of `input()`, use `eval(input())`.
     - **Example**, use a for loop to generate a sequence of values one at a time for each iteration of the loop:
       ```python
       n = eval(input("Enter # of iterations? "))
       for count in range(n):
           print(count, end=" ")
       print("\nDone")
       ```
     - Enter # of iterations? 6
       0 1 2 3 4 5
       Done

   - Removed <> as an alternate “not equal” operator, so use `!=` instead.
   - There is only one built-in integral type, named `int`. It behaves like the old `long` type.
   - An expression like `1/2` returns a float. Use `1//2` to get the truncating “integer division” behavior of version 2.
   - Dictionary methods `dict.keys()`, `dict.items()` and `dict.values()` return iterable “views” instead of lists. For example, this no longer works: `keyList = d.keys(); keyList.sort()`. Use `keyList = sorted(d)` instead.
     (Also, the `dict.iterkeys()`, `dict.iteritems()` and `dict.itervalues()` methods are no longer supported.)

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)
   print()
   ```

   ```python
   c = 'cat'
   d = c
   c += 'fish'
   print('c is', c)
   print('d is', d)
   ```

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

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

```
main
  initialization
    - welcome user
    - initialize variables
    - prompt for input
  calculate
    - compute answer
  display results
```

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

b) An alternative design methodology is to use object-oriented design. For the above dice problem, what objects would be useful and what methods (operations on the objects) should each perform?