CS 112 – Project #3 **Due Date: Sunday October 6th, 11:59pm.** **Loops, Lists**

Extra Credit(+5%): Due Thursday, October 3rd

The purpose of this assignment is to practice using loops and basic lists effectively.

Background:

Loop statements allow us to run the same block of code repeatedly, with the chance to use different values for variables each time as the accumulated effects pile up. Lists and strings are sequences that can be inspected value-by-value (and modified at will, for lists). We will use loops to define functions that perform calculations over sequences and numbers.

- Project Basics document (part of assignment): http://cs.gmu.edu/~marks/112/projects/project_basics.pdf
- Project Three tester file: http://cs.gmu.edu/~marks/112/projects/tester3p.py
- no template provided – create your file from scratch and include the functions defined below.

Grading

```
Well-commented/submitted:     10
Code passes shared tests:     80
Code passes hidden tests:     10
TOTAL:                       100
```

What can and can't I use or do?

Many built-in functions in python would make our tasks so trivial that you wouldn't really be learning, just "phoning it in". The following restrictions are in place for the entire project, and individual functions may list further restrictions (or pointed reminders). The whole point of this assignment is to practice writing loops, and seeing lists in the most basic way possible. There are indeed many different approaches that can "hide the loop" inside a function call, conversion to a different type, and other ways, but we want to make sure you're getting the practice intended from this assignment. ***Learning to code has different goals than finishing programs.***

Restrictions:

- you can't `import` anything
- no built-in functions other than those in the "allowed" list.
- no usage of sets, dictionaries, file I/O, list comprehensions, or self-made classes. Just focus on lists/loops.

Allowed:

- variables, assignments, selection statements (if-elif-else), loops (of course!), indexing/slicing.
- basic operators: `+`, `-`, `*`, `/`, `//`, `%`, `==`, `!=`, `<`, `>`, `<=`, `>=`, `in`, `not in`, `and`, `or`, `not`.
- when you need to build up a list, you can start with the empty list `[]` and `.append()` values into it.
- only these built-in functions/methods can be used: `len()`, `range()`, `str()`, `int()`, `.append()`, `.extend()`, `.pop()`
- you can implement your own version of other functions you want, and call them anywhere in your project.

Hints

In my solution, I only used the following things:

- basic operators, assignment, branching/loops, indexing, three built-ins functions `int()`, `len()`, `range()`.
- when the answer is a list, I build it up from an initial `[]` by calling `.append()` repeatedly.
- you'll *need* to use at least one loop per function, if not more!

Functions

First, we have a couple of warm-ups.

- `def how_odd(n):` Given a positive integer `n`, calculate how many times you can integer-divide this number in half and get another odd number? For instance, 11 divided by 2 is 5.5, which rounds down to 5, which is also odd. Then we'd next consider 5 to see if we get any more integer halvings to odd numbers.
  - **Assume:** `n` is a positive integer.
  - ```
    how_odd(11) →  2   # 11 → 5 → 2. 11 and 5 are odd, 2 is not.
    how_odd(10) →  0   # 10 is even.
    how_odd(47) →  4   # 47 → 23 → 11 → 5 → 2.
    ```

- `def vibrate(n):` Given a positive integer `n`, we take steps until we reach the value 1. Each time, if n is odd, we reduce it to one third and truncate to an int value; if n is even, we multiply it by 4/3, truncate it to an int, and then add one to get the next number. Return ***how many steps it takes to get to 1.***
  - **Assume:** `n` is a positive integer.
  - ```
    vibrate( 1) →  0   # it's already at 1, so zero steps are taken.
    vibrate(10) →  6   # 10 → 14 → 19 → 6 → 9 → 3 → 1. Six steps are taken.
    vibrate(81) →  4   # 81 → 27 → 9 → 3 → 1. Four steps are taken.
    ```

The rest of the functions have a theme – we want to ship items from our online store and will calculate boxes, look out for combustible items, and other calculations.

- `def is_combustible(name, combustibles):` Given an item `name`, and a list `combustibles` of items that could catch fire, return `True` if our named item is combustible, and `False` if it is not.
  - **Assume:** `name` is a string, `combustibles` is a list of strings.
  - **Reminder**: you must not hard-code the combustible items; use the given list argument!
    ```
    is_combustible("battery", ["battery", "lighter", "power bank"] ) →  True
    is_combustible( "doll",   ["battery", "lighter", "power bank"] ) →  False
    is_combustible("battery", ["lighter", "power bank"] )            →  False
    ```

- `def biggest_combustible(names, sizes, combustibles):` Given a list `names`, a corresponding list `sizes`, and a list `combustibles`, return the ***name*** of the largest item that is combustible. If nothing is combustible, or there are no items, return `None`.
  - **Assume:** `names` and `sizes` are lists of the same length; `names` only holds strings, and `sizes` only holds non-negative ints. `combustibles` is a list of strings.
    ```
    biggest_combustible(["doll","AA","AAA"], [15,5,4], ["AA","AAA","D"]) → "AA"
    biggest_combustible(["doll","bike"], [5,120], ["AA","AAA","D"])      → None
    biggest_combustible([],[],["laptop"])                                → None
    ```

- `def any_oversized(sizes, maximum):` Given a list of ints `sizes`, and an int `maximum`, return `True` if any sizes are larger than the maximum allowed value, `False` if not.
  - **Assume:** `sizes` is a list of ints; maximum is a non-negative int.
    ```
    any_oversized([1,2,3,4,5,6,7], 5) → True
    any_oversized([50,100, 200], 101) → True
    any_oversized([50,10, 20], 100)   → False
    ```

- `def any_adjacent_combustibles(names, combustibles):` Given a list of strings `names`, and a list of strings `combustibles`, return `True` if there are any combustibles in the names list directly next to each other, `False` if not.
    - **Assume:** `names` is a list of strings; `combustibles` is a list of strings.
      ```
      any_adjacent_combustibles(['a','b','c'],['a','b','d']) → True
      any_adjacent_combustibles(['a','b','c'],['c','b'])     → True
      any_adjacent_combustibles(['a','b','c'],['a','c'])     → False
      ```

- `def get_combustibles(names, combustibles):` Given a list of `names`, and a list of `combustibles`, create and return a list of the names of combustible items. Preserve ordering. The list might be empty.
    - **Assume:** `names` is a list of strings, and `combustibles` is a list of strings.
      ```
      get_combustibles(["doll","AA","AAA"],["AA","AAA","D"]) → ["AA","AAA"]
      get_combustibles(["doll","bike"],    ["AA","AAA","D"]) → []
      get_combustibles(["AA","belt","AA"], ["laptop","AA"])  → ["AA","AA"]
      ```

- `def cheap_products(names, prices, limit):` Given a list of strings `names`, a corresponding list of ints `prices`, and an int `limit`, create and return a list of the names of all items that cost no more than the given limit. Preserve the original order. The list might be empty.
    - **Assume:** `names` is a list of strings, `prices` is a list of ints, and `limit` is an int. `names` and `prices` are the same length (which may be zero).
    - ```
      cheap_products(["AA","car","hat"], [2,25000,12],  12) → ["AA","hat"]
      cheap_products(["AA","car","hat"], [2,25000,12],   2) → ["AA"]
      cheap_products(["AA","car"],       [2,25000],  56452) → ["AA","car"]
      ```

- `def box_sort(names, sizes):` Given a list of strings `names`, a corresponding list of ints `sizes`, we want to sort items by size so that each of our four sublists contains items in the smallest possible boxes of the following exact sizes: 2, 5, 25, and 50 units. Anything larger than 50 won't fit in a box and is simply ignored at this time. Create and return a list of the four sublists of items.
    - **Assume:** `names` is a list of strings, and `sizes` is a list of ints.
    - **Restrictions**: remember, you can't call any built-in sorting functions. It's not hard-coding to directly calculate based on the four given sizes, but keeping a list of box sizes may actually simplify your code.
      ```
      box_sort(['a','b','c','d'], [1,5,6,10]) → [['a'],['b'],['c','d'],[]]
      box_sort(['a','b','c'], [49,50,51])     → [[],[],[],['a','b']]
      ```

- `def packing_list(names, sizes, box_size):` Given a list of `names`, a corresponding list of int `sizes`, and an int `box_size`, this time we want to keep packing items into boxes of the given `box_size` until it's full, and then start filling another box of the same size. We return a list of filled boxes as a list of lists of strings. Oversized items are immediately placed into our output *in a list* by themselves, with an asterisk prefix to indicate that it does not fit. (We continue filling the current box after an oversized item). Items are only considered in their given ordering; do not reorder the items to seek a better packing list! Create and return this list of lists, each one containing items in one box or the single oversized item.
    - **Assume:** `names` is a list of strings, `sizes` is a list of ints, and `box_size` is an int. Order is preserved for all non-oversized items, and oversized items show up immediately before the box that was being filled at the time of consideration.

      ```
      # boxes of 2+1, 4, and 2.
      packing_list(['a','b','c','d'], [2,1,4,2],5) → [['a','b'],['c'],['d']]

      # while packing our second box, we find oversized item b, then finish our box.
      packing_list(['x','a','b','c'], [5,2,10,3], 5) → [['x'],['*b'],['a','c']]
      ```
      ```