CS 112 – Project #3 **Loops, Lists**

Due Date: Sunday June 27th, 11:59pm.

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.

Refer to our "Project Basics" in its entirety for information on naming your file, performing testing, grading details, and more. Consider it a part of this project's requirements, because it is.

- Project Basics document: https://mymasonportal.gmu.edu/bbcswebdav/xid-210009785_1
- Project Three tester file: https://mymasonportal.gmu.edu/bbcswebdav/xid-210098587_1
- no template provided – create your file from scratch and include the functions defined below.

What can and can't I use or do?

Many built-in functions in python would make some of 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 usage of sets, dictionaries, file I/O, or self-made classes. Just focus on lists and loops.
- no built-in functions other than those in the "can" list.
- no `zip()` function, and no list comprehensions
- you must not modify the original list - you must make a copy and then mangle the copy.

Allowed:

- variables, assignments, selection statements, loops (of course!), indexing/slicing.
- basic operators: +, -, *, /, //, %, ==, !=, <, >, <=, >=, in, not in, and, or, not. you may use variables,
- if 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: `len()`, `range()`, `str()`, `int()`, `.append()`,`.extend()`, `.pop()`
- you can write 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, indexing; assignment, selection, and loop statements; just two built-ins, `len()`, `range()`.
- when the answer is a list, I build it up from an initial [] by calling append() repeatedly.
- you may use selection statements, and you'll *need* to use at least one loop per function, if not more!

Functions

- **`odd_product(xs):`** Given a list of integers, find all the odd numbers and multiply them together. When no odds are present, the answer is `1`.
    - **Assume:** `xs` is a list with zero or more integers in it. Only integers will ever be present.
    - **Restrictions**: no extra restrictions.

```
  odd_product([1,2,3,4,5])       →    15
  odd_product([2,4,6])           →    1
  odd_product([])                →    1
```

- **`has_duplicates(xs):`** When a sequence has two equal values anywhere in the sequence, it has a duplicate. Determine if the sequence `xs` has any duplicates, and return as a `bool`.
    - **Assume:** `xs` is a sequence of any length (e.g., it could be a string or list).
    - **Restrictions**: remember, you may not call `count()` or related built-in functions.

```
has_duplicates("meter")            →    True      # two e's found
has_duplicates([1,3,5,2,4])        →    False
```

- **`span(nums):`** Given a list of numbers, return the difference between the largest and smallest number. If the list is empty, return zero.
    - **Assume:** `nums` is a list of any length, and all values are numbers.
    - **Restrictions**: remember, you may not call `min()`,`max()` or other related built-in functions.

```
span([1,0,-10,8.5])    →    18.5 # largest is 8.5; smallest is -10
span([3,3,3]           →    0    # 3 is the only value
span([])               →    0    # no value at all.
```

- **`truncatable_prime(n):`** check whether a number is a "right-truncatable prime" and return `bool` result. A right-truncatable prime is a prime which remains prime when the last (rightmost) digit is successively removed. *(hint: how can you remove the last digit?...)*
    - **Assume:** `n` is a positive integer.
    - **Restrictions**: no extra restrictions.

```
  truncatable_prime(2)            →    True
  truncatable_prime(113)          →    False #113 and 11 are primes, but 1 is not
  truncatable_prime(5939)         →    True #5939,593,59,and 5 are all primes
```

- **`remove_echo(xs):`** An echo is a contiguous sequence of identical values. Given `xs` as a list of values, this function returns a copy of `xs` with all but one value for each echo removed.
    - **Assume:** `xs` is a sequence of any length.
    - **Restrictions**: remember, you may not call `.index()`; do not modify xs.

```
remove_echo([1,1,3,3,3,2,1,2,2])            → [1,3,2,1,2]
remove_echo([1,3,2,1,2])                    → [1,3,2,1,2]           # no echo
remove_echo(["B","a","l","l","o","o","n"]) → ["B","a","l","o","n"] # a list of strings
remove_echo([])                             → []
```

- **`second_smallest(xs):`** Given a list, `xs`, figure out what the second-smallest value in the list is. When there are duplicates for the smallest, like `[2,4,3,2,5]`, the 'second copy' is the second-smallest.
    - **Assume:** `xs` is a list of `int`s with at least two values in it.
    - **Restrictions**: you may not call any sorting operations – use loop(s) and track what you see as you navigate through the list. You may not call `min()` or `max()` style functions either – use relational operators (like `<`) to perform the comparisons.

```
  second_smallest( [1, 2, 3, 4, 5]  ) →     2
  second_smallest( [5, 1, -4, -3, 2]) →    -3
  second_smallest( [2, 4, 3, 2, 5]  ) →     2
```

Extra Credit Function

Solve this problem for extra credit (up to +5%) and good practice.

- **`mode(xs):`** The mode of a list is the value that occurs the greatest number of times. When there is a tie for most occurrences, there are multiple mode values. Given a list of integers `xs`, return a list of all mode values. You must record them in the same order as their first occurrences are ordered in the original list.
  - **Assume:** `xs` is a list of integers.
  - **Restrictions**: you may not modify the original list, and you may not call any sorting functionality. But, you'll need to create your own lists, at a minimum for the answer, and it's likely you'll want to create some extra lists to help figure out the answer.
  - `mode([1,5,2,5,4,5])      →    [5]              # 5 occurred more than others`
  - `mode([1,4,2,4,2,3])      →    [4, 2]           # 4 occurred first`
  - `mode([1,2,3,4])          →    [1, 2, 3, 4]     # all-way tie!`

Grading

```
Code passes shared tests:      90
Well-commented/submitted:      10
TOTAL:                        100  +5 extra credit for mode()
```