Due Date: Sunday, February 28th, 11:59pm.

- The purpose of this assignment is to practice using loops and basic lists effectively.
- You will turn in a single python file following our naming convention: `netID_Lab#_P#.py`
- Include the following in comments at the top of your file: Name, G#, lecture and lab sections, and any references or comments we ought to know.
- Use the Piazza discussion forums (and professor/TA office hours) to obtain assistance.
- Remember, **do not publicly post code for projects on the forum**! Use private posts as needed (or when you're not sure). Have a specific question ready, and both show what you're thinking and show what you've tried independently before you got stuck. We will prod for more details if needed.

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.

Procedure

You will complete the following function definitions, each needing loops in their solutions. A few examples are provided with each description, and more exhaustive tests are provided in the accompanying testing file:

- http://cs.gmu.edu/~yzhong/112/p3/tester3p.py
- Invoke it as with prior assignments: `python3 tester3p.py yourcode.py`
- You can also limit what's tested like Lab 3 – name functions to test (e.g. just `count_odd_digits` and `replace`):

```
python3 tester3p.py yourcode.py count_odd_digits replace
```

Hint

In my solution, I only used the following things.

- basic math operators, relational operators, indexing expressions
- assignment statements, selection statements (if/elif/else), loops (while/for)
- functions: `len()`, `range()`. You can also define your own extra functions.
- when the answer is a list, I build it up from an initial `[]` by calling `append()` repeatedly.
- when the answer is a string, I build it up from an empty string and use concatenation (+) to add characters in.

Allowed/Disallowed Things

Follow the rules below regarding allowed/disallowed things for this project. If you find other options, **you need to ask ahead of time if it's okay to use something**. If you use something disallowed, you will lose points for using it.

- You may use variables, assignments (including augmented versions like +=), and basic operators: +, -, *, **, /, //, %, ==, !=, <, >, <=, >=, in, not in and, or, not.
- You may use selection statements (if/elif/else), and you **must** use at least one loop (while/for) per function, if not more! Note that if you call another function you wrote that has a loop to get the job done, that still counts as "using a loop". You may use break or continue if needed.
- You may these data types: int, float, string, bool, and list.
- You may these built-in functions: len(), range(), int(), str(), float(), input(), print(). **Note**: standard I/O (input() and print()) are not needed since we are defining functions. You can use them for debugging purpose but should remove them before submission.
- List operations: you can index and slice all you want. If you need to build up a list, you can start with the empty list and .append() values into it.
- No string methods are allowed: you must use the basic concatenation to build up a string.
- You must not modify the original list - if you want to do so, you must make a copy and then perform any modifications according to the rules above.
- You can define additional helper functions in your code using def.

- No modules may be imported.
- You may NOT use any built-in functions, keywords, or data types that are not listed above.
- Not everything listed above are needed for the project.

The whole point of this assignment is to practice writing loops, and seeing lists in the basic way. 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, so we've had to clarify the scope of exactly what you may use or not. There are a couple of additional clarifications below for individual functions, but most of them are reminders.

Functions

- `count_odd_digits(n):` Given a non-negative integer, count how many digits of it are odd numbers.
  - **Assume:** `n` is a non-negative integer; `n` could be zero.
  - `count_odd_digits(5)` → `5`
  - `count_odd_digits(24)` → `0`
  - `count_odd_digits(123450)` → `3` `#digits 1,3, and 5 are odd`

- `prime_divisors(num) (n):` A divisor is a number that divides something evenly. Given a positive integer `n`, find all divisors of `n` that are prime numbers and return them in a list from lowest to highest.
  - **Assume:** `n` is a positive integer.
  - `prime_divisors(13)` → `[13]`
  - `prime_divisors(12)` → `[2,3]`
  - `prime_divisors(1020)` → `[2,3,5,17]`

- **`remove_repeat(msg):`** A repeat is a sequence of identical characters in a row. Distant repetitions don't count as a repeat. Remove all but one character for each repeats in string **`msg`**, and return a copy with no repeat but otherwise identical to **`msg`**.
    - **Assume:** **`msg`** is a string of any length; **`msg`** could be an empty string.
    - `remove_repeat("bookkeeper")` → `'bokeper'`
    - `remove_repeat("aaabcaaddddeff")` → `'abcadef'`
    - `remove_repeat("a")` → `'a'`

- **`float_range(start,stop,step):`** We define our own floating-point version of range() here. It works the same as built-in range() function but can accept float values. Note: some floating point arguments will give you unexpected values due to the imprecision of floats in computer – you don't need to worry about them as far as your algorithm is correct.
    - **Assume:** **`start`**, **`stop`**, and **`step`** are integer or float numbers. **`step`** is a non-zero value.
    - **Restrictions**: you may not call **`range()`** . (Plus, it only takes integer arguments).
    - `float_range(0,1,0.25)` → `[0,0.25,0.5,0.75]`
    - `float_range(15.1,4,-2)` → `[15.1,13.1,11.1,9.1,7.1,5.1]`
    - `float_range(0,0.25,-1)` → `[]`

- **`replace(xs,old,new,limit):`** Given a list **`xs`**, return a **copy** of **`xs`** in which occurrences of **`old`** value has been replaced with **`new`** value. **`limit`** is an integer indicates the maximum number of replacements allowed; the earliest matches must be replaced. If **`limit`**>= no. of occurrences, replace all. If **`limit`**<=0, no replacements occur – just return an identical copy.
    - **Assume:** **`xs`** is a list; **`xs`** could be empty.
    - **Restrictions**: You may NOT modify the original list **`xs`**.
    - `replace([1,2,3,4],1,100,1)` → `[100,2,3,4]`
    - `replace([1,2,3,4],5,100,1)` → `[1,2,3,4]` `#no replacement`
    - `replace([1,2,3,4,1],1,100,1)` → `[100,2,3,4,1]` `#only the first 1 replaced`
    - `replace([1,2,3,4],1,100,0)` → `[1,2,3,4]` `#no replacement`

- **`sum_positive_2d(xss):`** Given a "two-dimensional list" of integers – that is, a list where each value in it is a list of integers – find all the positive numbers and sum them together.
    - **Assume:** **`xss`** is a list where each value is a list of integers; **`xss`** and its members could be empty.
    - **Restrictions**: you may not call **`sum()`**. (It would have included all values anyways). You may NOT modify the original list **`xss`**.
    - Examples:
        - `sum_positive_2d([[0,1,2],[3,4,5]])` → `15`
        - `sum_positive_2d([[0,1,2],[3,4,-5]])` → `10`
        - `sum_positive_2d([[0,-1,2],[-3,-4],[5],[]])` → `7`

Extra Credit Function

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

- **`rank(xs):`** Given a list of integers xs, return a list of the unique items in xs, ordered by their number of occurrences (more occurrences earlier in this resulting list). For ties in number of occurrences, you must preserve the order of the first occurrence of each. It's likely you'll want to create some extra lists to help figure out the answer.
  - **Assume: `xs`** is a list of integers.
  - **Restrictions**: you may not modify the original list, but you may use .sort() or sorted(), just for this function. Other built-in functions/methods not listed above are still not allowed, including .count().
  - `rank([1,5,2,5,4,5])` → `[5,1,2,4]` `# 5 occurred more than others`
  - `rank([1,4,2,4,2,3,3,3])` → `[3,4,2,1]` `# note: 4 occurred first`
  - `rank([4,3,2,1])` → `[4,3,2,1]` `# tie for occurrence`

Grading Rubric

```
Code passes shared tests:     75 (zero points for hard-coding)
Well-documented/submitted:    10
Appropriate loop usage:       15
TOTAL:                       100  +5 extra credit for rank()
```

Reminders on Turning It In:

No work is accepted more than 48 hours after the initial deadline, regardless of token usage. Tokens are automatically applied whenever they are available.

You can turn in your code as many times as you want; we only grade the last submission that is <=48 hours late. If you are getting perilously close to the deadline, it may be worth it to turn in an "almost-done" version about 30 minutes before the clock strikes midnight. If you don't solve anything substantial at the last moment, you don't need to worry about turning in code that may or may not be runnable, or worry about being late by just an infuriatingly small number of seconds – you've already got a good version turned in that you knew worked at least for part of the program.

You can (and should) check your submitted files. If you re-visit BlackBoard and navigate to your submission, you can double-check that you actually submitted a file (it's possible to skip that crucial step and turn in a no-files submission!), you can re-download that file, and then you can re-test that file to make sure you turned in the version you intended to turn in. It is your responsibility to turn in the correct file, on time, to the correct assignment.

Use a backup service. Do future you an enormous favor, and just keep all of your code in some automatically synced location, such as a Dropbox or Google Drive folder. Every semester someone's computer is lost/drowned/dead, or their USB drive is lost/destroyed, or their hard drive fails. Don't give these situations the chance to doom your project work!