Homework 1

Name: William Martinez

Colaborators: None

This homework focuses on topics related to basic data types, collections, and iterations.

I encourage collaborating with your peers, but the final text, code, and comments in this homework assignment should still be written by you. Please check to the collaboration policy on BruinLearn.

Pay special attention to the instructions - should your function `print` something or `return` something?

Submission instructions:

- Submit `HW1.py` and `HW1.ipynb` compressed in a single file on Gradescope under "HW1 - Autograder". Compress the two files directly, not compressing the folder containing the two files. Do **NOT** change the file name. The style and readability of your code will be checked by the reader aka human grader.
- Convert this notebook into a pdf file and submit it on GradeScope under "HW1 - PDF". Make sure the figure in the last part is visible.

Comments and Docstrings

You will be graded in part on the quality of your documentation and explanation of your code. Here's what we expect:

- **Comments**: Use comments liberally to explain the purpose of short snippets of code.
- **Docstrings**: Functions (and later, classes) should be accompanied by a *docstring*. Briefly, the docstring should provide enough information that a user could correctly use your function ***without seeing the code.*** In somewhat more detail, the docstring should include the following information:
  - One or more sentences describing the overall purpose of the function.
  - An explanation of each of the inputs, including what they mean, their required data types, and any additional assumptions made about them.
  - An explanation of the outputs.

In future homeworks, we will be looking for clear and informative comments and

docstrings.

Code Structure

In general, there are many good ways to solve a given problem. However, just getting the right result isn't enough to guarantee that your code is of high quality. Check the logic of your solutions to make sure that:

- You aren't making any unnecessary steps, like creating variables you don't use.
- You are effectively making use of the tools in the course, especially control flow.
- Your code is readable. Each line is short (under 80 characters), and doesn't have long tangles of functions or `()` parentheses.

Ok, let's go!

In [ ]:
```
# This cell imports your functions defined in HW1.py
from HW1 import print_s, print_s_lines, print_s_parts
from HW1 import print_s_some, print_s_change
from HW1 import make_count_dictionary
from HW1 import gimme_an_odd_number
from HW1 import get_triangular_numbers, get_consonants
from HW1 import get_list_of_powers, get_list_of_even_powers
from HW1 import random_walk

# This is for problem 5
import random
from matplotlib import pyplot as plt
random.seed(203)
```

Problem 1

(a) Define variable `s` in the cell below

Take a look at the function `print_s` in HW1.py, and understand what that function does.

In the cell below, define a string variable `s` such that `print_s(s)` prints:

```
Tired    : Doing math on your calculator.
Wired    : Doing math in Python.
Inspired : Training literal pythons to carry out long
division using an abacus.
```

The potentially tricky part here is dealing with the newlines. You can choose to use newline characters, or use triple quotes. See:

https://docs.python.org/3/tutorial/introduction.html#strings.

In [ ]:
```
s = '''\
Tired    : Doing math on your calculator.
Wired    : Doing math in Python.
Inspired : Training literal pythons to carry out long division using an abac
'''

print_s(s)
```

```
Tired    : Doing math on your calculator.
Wired    : Doing math in Python.
Inspired : Training literal pythons to carry out long division using an abac
us.
```

Next, write Python commands which use `s` to print the specified outputs. Feel free to use loops and comprehensions; however, keep your code as concise as possible. Each solution should require at most three short lines of code.

For full credit, you should minimize the use of positional indexing (e.g. `s[5:10]`) when possible.

(b) Define function `print_s_lines` in HW1.py

When `print_s_lines(s)` is run with the previously defined `s`, it should print:

```
Tired
Doing math on your calculator.
Wired
Doing math in Python.
Inspired
Training literal pythons to carry out long division using an
abacus.
```

In [ ]:
```
print_s_lines(s)
```

```
Tired
Doing math on your calculator.
Wired
Doing math in Python.
Inspired
Training literal pythons to carry out long division using an abacus.
```

(c) Define `print_s_parts` in HW1.py

When `print_s_parts(s)` is run with the previously defined `s`, it should print:

```
Tired
Wired
```

```
Inspired
```

Hint: look at the endings of words. A small amount of positional indexing might be handy here.

In [ ]:
```
print_s_parts(s)
```

```
Tired
Wired
Inspired
```

(d) Define `print_s_some` in HW1.py

When `print_s_some(s)` is run with the previously defined `s`, it should print:

```
Tired    : Doing math on your calculator.
Wired    : Doing math in Python.
```

Hint: These two lines are shorter than the other one. You are NOT allowed to use the fact that these are the first two sentences of the text.

In [ ]:
```
print_s_some(s)
```

```
Tired    : Doing math on your calculator.
Wired    : Doing math in Python.
```

(e) Define `print_s_change` in HW1.py

When `print_s_change(s)` is run with the previously defined `s`, it should print:

```
Tired    : Doing data science on your calculator.
Wired    : Doing data science in Python.
Inspired : Training literal pythons to carry out machine
learning using an abacus.
```

Hint: `str.replace`.

In [ ]:
```
print_s_change(s)
```

```
Tired    : Doing data science on your calculator.
Wired    : Doing data science in Python.
Inspired : Training literal pythons to carry out machine learning using an a
bacus.
```

Problem 2: Define `make_count_dictionary` in HW1.py

The function `make_count_dictionary` takes a list `L` and returns a dictionary `D` where:

- The *keys* of `D` are the unique elements of `L` (i.e. each element of `L` appears only once).
- The value `D[i]` is the number of times that `i` appears in list `L`.

Make sure your function has a descriptive docstring and is sufficiently commented.

Your code should work for lists of strings, lists of integers, and lists containing both strings and integers.

For example:

```
# input
L = ["a", "a", "b", "c"]
# output
{"a" : 2, "b" : 1, "c" : 1}
```

Attend to Efficiency

A good way to solve this problem is using the `list.count()` method. However, you should carefully check the structure of your code to ensure that you are not calling `list.count()` an unnecessary number of times. Consider the supplied example above: how many times should list.count() be called?

There are also other good solutions to this problem which do not use list.count(). Here as well, make sure that you are not performing unnecessary computations.

In [ ]:
```
L = ["hello","a", "a", "b", "c", 2, 2, 3]

make_count_dictionary(L)
```

Out[ ]:
```
{'hello': 1, 'a': 2, 'b': 1, 'c': 1, 2: 2, 3: 1}
```

Problem 3: Define `gimme_an_odd_number` in HW1.py

The `input()` function allows you to accept typed input from a user as a string. For example,

```
x = input("Please enter an integer.")
# user types 7
x
# output
'7'
```

Function `gimme_an_odd_number` does not take any inputs. When it's run, it prompts to `"Please enter an integer."`. If the user inputs an even integer, the code should re-prompt them with the same message. If the user has entered an odd integer, the function should print a list of all numbers that the user has given so far, and also return the same list.

You may assume that the user will only input strings of integers such as `"3"` or `"42"`.

Hint: Try `while` and associated tools.

Hint: Which built-in Python function (https://docs.python.org/3/library/functions.html) can turn string `"3"` to integer `3`?

Example

```
# run gimme_an_odd_number()

> Please enter an integer.6
> Please enter an integer.8
> Please enter an integer.4
> Please enter an integer.9
> [6, 8, 4, 9]
```

In [ ]:
```
gimme_an_odd_number()
```

Out[ ]:
```
[1]
```

Problem 4

Write list comprehensions which produce the specified list. Each list comprehension should fit on one line and be no longer than 80 characters.

(a) Define `get_triangular_numbers` in HW1.py

The `k` th triangular number (https://en.wikipedia.org/wiki/Triangular_number) is the sum of natural numbers up to and including `k`. Write `get_triangular_numbers` such that for a given `k`, it returns a list of the first `k` triangular numbers.

For example, the sixth triangular number is

$$1 + 2 + 3 + 4 + 5 + 6 = 21,$$

Formula for triangle numbers:

$$\left(\frac{n(n+1)}{2}\right)$$

and running `get_triangular_numbers` with an argument of `k=6` should output `[1, 3, 6, 10, 15, 21]`. You function should have a docstring.

In [ ]:
```
k = 6

get_triangular_numbers(k)
```

Out[ ]:
```
[1, 3, 6, 10, 15, 21]
```

(b) Define `get_consonants` in HW1.py

The function `get_consonants` taks a string `s` as an input, and returns a list of the letters in `s` **except for vowels, spaces, commas, and periods.** For the purposes of this example, an English vowel is any of the letters `["a", "e", "i", "o", "u"]`. For example:

```
s = "make it so, number one"
print(get_consonants(s))
["m", "k", "t", "s", "n", "m", "b", "r", "n"]
```

Hint: Consider the following code:

```
l = "a"
l not in ["e", "w"]
```

Each element in the returned is is one character long, is not a vowel, space, comma, nor period, is in `s`, and may appear multiple times. The elements appear in the same order as the letters in `s``.

In [ ]:
```
get_consonants('make it so, number one')
```

Out[ ]:
```
['m', 'k', 't', 's', 'n', 'm', 'b', 'r', 'n']
```

(c) Define `get_list_of_powers` in HW1.py

The function `get_list_of_powers` takes in a list `X` and integer `k` and returns a list `L` whose elements are themselves lists. The `i` th element of `L` contains the powers of `X[i]` from `0` to `k`.

For example, running `get_list_of_powers` with inputs `X = [5, 6, 7]` and `k = 2` will return `[[1, 5, 25], [1, 6, 36], [1, 7, 49]]`. The `i` th element is a list of the powers of `X[i]` from `0` to (and including) `k`, in increasing order.

In [ ]:
```
X = [5, 6, 7]
k = 2

print(get_list_of_powers(X, k))
```

```
[[1, 5, 25], [1, 6, 36], [1, 7, 49]]
```

(d) Define `get_list_of_even_powers` in HW1.py

As in **(c)**, the function `get_list_of_even_powers` takes in a list `X` and inter `k`, and returns a list `L` whose elements are themselves lists. But now `L` includes only even powers of elements of `X`. For example, running `get_list_of_even_powers` with inputs `X = [5, 6, 7]` and `k = 8` should return `[[1, 25, 625, 15625, 390625], [1, 36, 1296, 46656, 1679616], [1, 49, 2401, 117649, 5764801]]`.

The `i` th element is a list of the EVEN powers of `X[i]` from `0` to (and including) `k`, in increasing order.

In [ ]:
```
X = [5, 6, 7]
k = 8

print(get_list_of_even_powers(X, k))
```

```
[[1, 25, 625, 15625, 390625], [1, 36, 1296, 46656, 1679616], [1, 49, 2401, 1
17649, 5764801]]
```

Problem 5: Define `random_walk` in HW1.py

In this problem, we'll simulate the *simple random walk*, perhaps the most important discrete-time stochastic process. Random walks are commonly used to model phenomena in physics, chemistry, biology, and finance. In the simple random walk, at each timestep we flip a fair coin. If heads, we move foward one step; if tails, we move backwards. Let "forwards" be represented by positive integers, and "backwards" be represented by negative integers. For example, if we are currently three steps backwards from the starting point, our position is `-3`.

Write `random_walk` to simulate a random walk. Your function should:

- Take an upper and lower bound as inputs.
- Return three variables `pos`, `positions`, `steps`, in that order.
- `pos` is an integer, and indicates the walk's final position at termination.
- `positions` is a list of integers, and it is a log of the position of the walk at each time step. Includes the initial position but excludes the final position.
- `steps` is a list of integers, and it is a log of the results of the coin flips. Values of `-1`s and `1`s.

When the walk reaches the upper or lower bound, print a message such as `Upper bound at 3 reached` and terminate the walk.

Your code should include at least one instance of an `elif` statement and at least one instance of a `break` statement.

Hint To simulate a fair coin toss, try running the following cell multiple times. Use `+1`'s and `-1`'s instead of `"heads"` and `"tails"` for your function!

In [ ]:
```
for _ in range(10):
    x = random.choice(["heads","tails"])
    print(x)
```

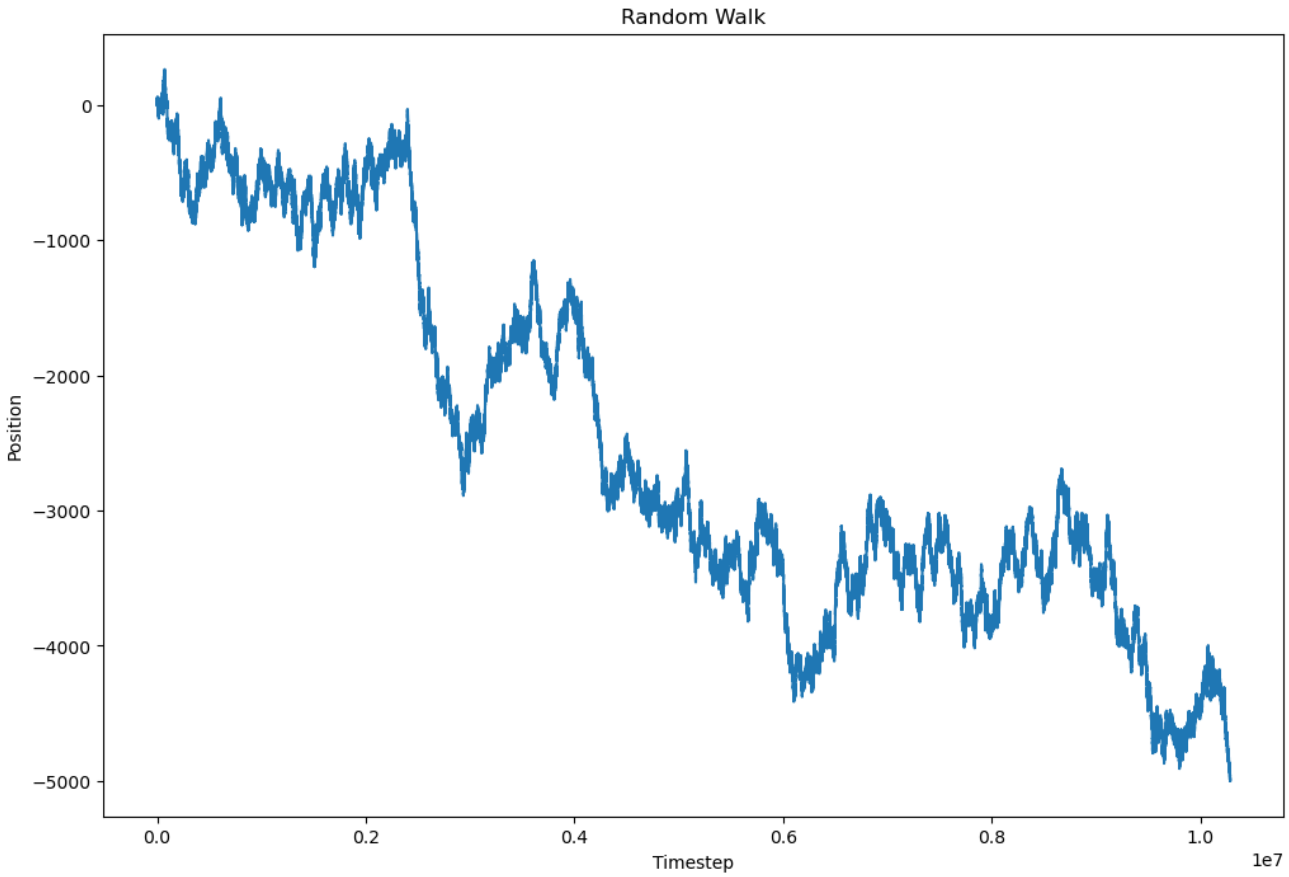
```
heads
heads
tails
heads
heads
tails
heads
heads
heads
tails
```

Finally, you might be interested in visualizing the walk. Run the following cell to produce a plot. When the bounds are set very large, the resulting visualization can be quite intriguing and attractive. It is not necessary for you to understand the syntax of these commands at this stage.

In [ ]:
```
# Assign pos, positions, and steps the equivalent return values from
# random_walk
pos, positions, steps = random_walk(5000, -5000)
# Characterisitcs of the return values from random_walk().
print(pos)
print(len(positions))
print(positions[-10:])
print(len(steps))
print(steps[-10:])

# plot of the position wrt number of coin tosses
plt.figure(figsize=(12, 8))
plt.plot(positions)
plt.xlabel('Timestep')
plt.ylabel('Position')
plt.title('Random Walk')
plt.show()
```

```
Lower bound at -5000 reached
-5000
10286040
[-4998, -4999, -4998, -4999, -4998, -4997, -4998, -4999, -4998, -4999]
10286040
[-1, 1, -1, 1, 1, -1, -1, 1, -1, -1]
```


Random Walk
0
−1000
−2000
−3000
−4000
−5000
Position
0.0
0.2
0.4
0.6
0.8
1.0
1e7
Timestep

hw1/HW1.py

```python
# PIC 16A HW1
# Name: William Martinez
# Collaborators: None
# Date: 4/18/24

import random # This is only needed in Problem 5
# random.seed(1)

# Problem 1

def print_s(s):
    """
    Prints a given string.
    ---
    Args:
        s: A string.
    Returns:
        None
    """
    print(s)

# you do not have to add docstrings for the rest of these print_s_* functions.

def print_s_lines(s):
    # replace colon with new line
    print(s.replace(': ', '\n'))

def print_s_parts(s):
    # remove all spaces then replace colon with new line.
    sos = s.replace(' ', '').replace(':', '\n')
    # Keep everyother element
    sos = sos.split(sep = '\n')[::2]
    # join list, sos, by new lines then print
    print('\n'.join(sos))

def print_s_some(s):
    # split string into list by new line
    # perform a decending sort, then drop fisrt element.
    sos = sorted(s.split('\n'), key = len, reverse = True)[1:]
    # join list into a string.
    sos = '\n'.join(sos)
    print(sos)

def print_s_change(s):
    # replace math with data science
    sc = s.replace('math', 'data science')
    # replace long division with machine learning
    sc = sc.replace('long division', 'machine learning')
    print(sc)

# Problem 2
```

```python

def make_count_dictionary(L):
    """
    Return a dictionary of the frequency of elements in a list.
    ---
    Args:
        L: A list
    Returns:
        D: A dictionary of element frequencies
    """
    # Initialize d_c and d_v
    d_k = []
    d_v = []
    # Distinct elements in list, L, keeping the same order of list, L
    for i in L:
        if i not in d_k:
            d_k.append(i)
    # Frequency of distinct elements
    for i in d_k:
        d_v.append(L.count(i))
    # Dictionary of the frequencies of distinct elements
    D = dict(zip(d_k, d_v))
    return D

# Problem 3

def gimme_an_odd_number():
    """
    A loop that terminates when a user inputs an odd number.
    Returns a list of user inputted numbers.
    ---
    Args:
        None
    Returns:
        usr_list: A list of user responses
    """
    # initialize user responses
    usr = 0
    usr_list = []
    # Keep requesting integers till until input is odd. Append all responses
    # to usr_list then print
    while (usr % 2 == 0):
        usr = int(input("Please enter an integer."))
        usr_list.append(usr)
    # print(usr_list)
    return print(usr_list)

# Problem 4

def get_triangular_numbers(k):
    """
    Returns a list of the number of objects needed to make
    a k-sided, equalateral triangle from 1 to k.
```

```
    ---
    Args:
        k: An Integer
    Returns:
        num_list: A list of the number of objects needed to make
        a k-sided, equalateral triangle from 1 to k
    """
    # initialize num_list
    num_list = []
    # Append the integer corresponding to the number of objects needed to make
    # a k-sided, equalateral triangle from 1 to k
    for i in range(1, k + 1):
        num_list.append(int(i * (i + 1) / 2)) # Formula for triangle numbers
    return num_list

def get_consonants(s):
    """
    Returns a list of characters that are not a vowel, space, comma, or period.
    ---
    Args:
        s: A string
    Returns:
        cp_list: A list
    """
    # list of characters that are a vowel, space, comma, or period.
    rm_list = ["a", "e", "i", "o", "u", " ", ",", ".",]
    # initialize cp_list
    cp_list = []
    # for each character, drop
    for i in s:
        if i in rm_list:
            pass
        else:
            cp_list.append(i)
    return cp_list


def get_list_of_powers(X, k):
    """
    Returns a 2 dimentional list of integers. Each element is a list
    of the powers of an element of X from 0 to k.
    ---
    Args:
        X: List of integers
        k: An Integer
    Returns:
        L: A 2-dimentional list
    """
    # Initialize the 2 dimentional list, L
    L = []
    # Initial the sub list, L_sub, then append the
    # the powers of each element X from 0 to k
    for i in X:
        L_sub = []
```

```
        for j in range(0, k + 1):
            L_sub.append(i**j)
        # Append each sub list, L_sub, to the 2 dimentional list, L
        L.append(L_sub)
    return L


def get_list_of_even_powers(X, k):
    """
    Returns a 2 dimentional list of integers. Each element is a list
    of the even powers of an element of X from 0 to k.
    ---
    Args:
        X: List of integers
        k: An Integer
    Returns:
        L: A 2-dimentional list
    """
    # Initialize the 2 dimentional list, L
    L = []
    # Initial the sub list, L_sub, then append the
    # the even powers of each element X from 0 to k
    for i in X:
        L_sub = []
        for j in range(0, k + 1, 2):
            L_sub.append(i**j)
        # Append each sub list, L_sub, to the 2 dimentional list, L
        L.append(L_sub)
    return L



# Problem 5

def random_walk(ub, lb):
    """
    Returns the last postion, position history, and step history for a fair
    coin toss where head moves forwards (+1) and tails moves backwards (-1).
    ---
    Args:
        ub: An integer that represents the upperbound position. When pos reaches
        ub, the loop stops.
        lb: An integer that represents the lowererbound position. When pos
        reaches sub, the loop stops.
    Returns:
        pos: An integer that represents the last position.
        positions: A List that is a log of the position history.
        steps: A list that is a log of the step history.
    """
    # Initialize pos, positions, and steps. Start pos and positions at 0
    pos = 0
    positions = [0]
    steps = []
```

```
    # Loop random coin flips
    while True:
        # Break loop if upper or lower bounds reached. Drop last element.
        if (pos == ub):
            print("Upper bound at {} reached".format(ub))
            positions = positions[:-1]
            break
        elif (pos == lb):
            print("Lower bound at {} reached".format(lb))
            positions = positions[:-1]
            break
        # Perform coin flip, assign +1 to heads and -1 to tails. Append current
        # position, pos, to positions and append step result to steps.
        else:
            x = random.choice(["heads", "tails"])
            if x == "heads":
                pos += 1
                positions.append(pos)
                steps.append(1)
            elif x == "tails":
                pos -= 1
                positions.append(pos)
                steps.append(-1)
    return pos, positions, steps


# If you uncomment these two lines, you can run
# the gimme_an_odd_number() function by
# running this script on your IDE or terminal.
# Of course you can run the function in notebook as well.
# Make sure this stays commented when you submit
# your code.
#
# if __name__ == "__main__":
#     gimme_an_odd_number()
```