HW1

January 13, 2023

1 Homework 1

1.0.1 Name: [David Reynoso]

I affirm that I personally wrote the text, code, and comments in this homework assignment.

Jan. 13th, 2023

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.

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

1.1 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.

We gave you docstrings in this homework, and you should read them all! In future homeworks, as well as on exams, we will be looking for clear and informative comments and docstrings.

1.2 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!

```python
[2]: # This cell imports your functions defined in HW1.py
     from HW1 import make_count_dictionary
     from HW1 import get_triangular_numbers, get_consonants, get_list_of_powers,
      ↪get_list_of_even_powers
     from HW1 import random_walk
```

2 Problem 1

2.1 (a)

In the cell below, define a string variable `s` such that `print(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.

```python
[2]: # define s and print it here
     s = """Tired     : Doing math on your calculator.
     Wired     : Doing math in Python.
     Inspired : Training literal pythons to carry out long division using an abacus.
      ↪"""
     print(s)
```

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

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.

2.2 (b)

Output:

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

```
[4]: # creates a list of lists where we have first split by ": " then by "\n" stored
     ↪in subb
     subb = [word.split("\n") for word in s.split(": ")]
     # next is now a list that unpacked the lists in the subb list keeping order
     next = [subb[i][j] for i in range(len(subb)) for j in range(len(subb[i]))]
     # we can now just iterate and print
     for i in range(len(next)): print(next[i])
```

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

2.3 (c)

Output:

```
Tired
Wired
Inspired
```

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

```
[5]: # we take the list of substrings from splitting by "\n" and store it in
     ↪variable nonew
     nonew = s.split("\n")
     # we utilize the find method to find the index of first " " since we just want
     ↪the first word in each substring
     # in the list and we iterate and print
     for i in nonew: print(i[:i.find(" ")])
```

```
Tired
Wired
Inspired
```

2.4 (d)

Output:

```
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.

```
[6]: # we take the list of substrings from splitting by "\n" and store it in
     ↪variable wowza
     wowza = s.split('\n')
```

```
# we create a new list from taking only the elts of wowza with length less than␣
 ↪43
# aka the two lines we want
endlist = [i for i in wowza if (len(i) < 43)]
# iterate and print from such list
for i in endlist: print(i)
```

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

2.5 (e)

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`.

```
[7]: def print_s_change(jimmy):
         # we use the replace method to make the desired replacements for both words/
     ↪phrases in the string
         print(jimmy.replace("math", "data science").replace("long division",␣
     ↪"machine learning"))
```

```
[9]: # test on s
     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
abacus.
```

3 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}
```

[10]:
```
# test your make_count_dictionary here, like this:
L = ["a", "a", "b", "c"]
count_dict = make_count_dictionary(L)
print(count_dict)

# come up with other test cases (examples).
L = ["aasda", "afasa", "bfsafasd", "csafas","sdasda", "jimmy", "john", "jimmy",␣
 ↪"john"]
count_dict = make_count_dictionary(L)
print(count_dict)
```

```
{'a': 2, 'b': 1, 'c': 1}
{'aasda': 1, 'afasa': 1, 'bfsafasd': 1, 'csafas': 1, 'sdasda': 1, 'jimmy': 2,
'john': 2}
```

4   Problem 3: Write the script `gimme_an_odd_number.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'
```

When `gimme_an_odd_number.py` is 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.

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`?

4.0.1   Example

```
# run gimme_an_odd_number.py

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

5

5 Problem 4

Write list comprehensions which produce the specified list.

For problems c and d, you can either write a nested list comprehension (https://docs.python.org/3/tutorial/datastructures.html#nested-list-comprehensions), or use one loop + list comprehension inside. I would recommend the following steps: - write something that works with two loops (nested loops) - condense the inner loop into a list comprehension - try turning the whole thing into a nested list comprehension

5.1 (a) Define `get_triangular_numbers` in HW1.py

The kth 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

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

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.

```
[12]:  # test your get_triangular_numbers here
       k = 3
       print(get_triangular_numbers(k))

       k = 6
       print(get_triangular_numbers(k))

       k = 10
       print(get_triangular_numbers(k))
```

```
[1, 3, 6]
[1, 3, 6, 10, 15, 21]
[1, 3, 6, 10, 15, 21, 28, 36, 45, 55]
```

5.2 (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"]
```

6

```
[13]: # test your get_consonants here
      palabra = "make it so, number one"
      print(get_consonants(palabra))
```

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

5.3 (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 `ith` 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]]`.

```
[14]: # test your get_list_of_powers here
      X = [5, 6, 7]
      k=2
      print(get_list_of_powers(X,k))
```

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

5.4 (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]]`.

```
[15]: # test your get_list_of_even_powers here
      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,
117649, 5764801]]
```

6 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.
- **steps** is a list of integers, and it is a log of the results of the coin flips.
- Please read the provided docstrings for more details on what the autograder expects.

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. You don't have to use `"heads"` and `"tails"` when using this function – can you think of a more useful choice set?

```
[3]: import random
     from matplotlib import pyplot as plt
```

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

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

```
[5]: # test your random_walk here
     print(random_walk(10, -10))
```

```
(10, [0, 1, 0, 1, 2, 1, 2, 1, 0, 1, 2, 3, 4, 5, 4, 3, 2, 1, 2, 3, 4, 5, 6, 7, 6,
7, 8, 7, 8, 9], [1, -1, 1, 1, -1, 1, -1, -1, 1, 1, 1, 1, 1, -1, -1, -1, -1, 1,
1, 1, 1, 1, 1, -1, 1, 1, -1, 1, 1, 1])
```

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.

```
[11]: pos, positions, steps = random_walk(5000, -5000)
      plt.figure(figsize=(12, 8))
      plt.plot(positions)
      plt.xlabel('Timestep')
      plt.ylabel('Position')
      plt.title('Random Walk')
      plt.show()
```

Random Walk