Homework 1

Total Points: 100 (Correctness and Style)

Due: Tuesday, January 16th, 11:59pm

Starter Files

Download hw01.zip. Inside the archive, you will find starter files for the questions of this homework. You cannot import anything to solve the problems.

Part 0: Integrity of Scholarship Agreement

Before starting the homework, please carefully read and fill out this integrity of scholarship agreement form. You **will NOT receive scores** in this class until you submit the form.

Click here to sign the form

Part 1. Coding and Docstring Style

Starting from HW1 **you will be graded on code style**. Since often your code will be shared among other people, you'd not want them to find your code hard to use or understand. This becomes crucial especially when you get into the industry during your internship or full-time, and your fellow developers will demand good styling from your code. This is why we assign this homework at first to help you build the habit of enforcing styling.

In order to keep everything in order, the Python community has agreed on *recommended* programming styles to help everyone write code in a common style that makes the most sense for shared code. This style is captured in PEP-8. We are not going to enforce all requirements only the most common ones:

There are two ways you could check for the style:

1) Upload your code to Gradescope and our script will tell you what is missing/incorrect. **Note, there are still a few requirements that will be graded manually.**

2) Uploading code to Gradescope each time you need to test something is too annoying. Instead, you can follow the steps below and check the style of your code as you write it.

Link to style guide (also located below and on course website)

Style Requirements

You will be graded for the style of programming on all homework assignments. A few key requirements for style are given below:

This is a complete Python style guide: Python Style Guide.

However, we will only test you on the smaller set of rules below:

- **Illegal Import Statements**: You should not import any package unless instructed to;
- **Module Docstring**: Every file that you submit should have a module docstring at the very top. In our assignments, it means to fill in name and PID;
- **Method Docstring**: Every method you create should have a docstring (i.e. method description)
    - Each docstring is surrounded by triple quotes (""") instead of triple single quotes (''')
    - This includes any inner function and helper function;
    - You don't need docstrings for lambda functions;
    - You may replace the entire # box with your method description;
    - The description should briefly describe what the method does, instead of what steps you take to achieve the result. Example:
        - Correct: Takes in a list of numbers, doubles each of them, and returns the doubled list;
        - Incorrect: Initializes an empty list and loops through the input list. For each number, I double it and append to the resulting list.
    - It's recommended to include input argument type and information like the examples given in lab01, but not required.
- **Line Limit**: All lines should be limited to a maximum of 79 characters.
    - Setup rulers in your editor
        - Sublime Text (reference): Go to *Preferences* > *Settings - User*, and add a new line in {}
          `"rulers": [80]`
        - You may also google "<editor name> set ruler" for your editor of choice
    - You may use backslash (\) to break up lines that might overflow:

```
with open('/path/to/some/file/you/want/to/read') as file_1, \
     open('/path/to/some/file/being/written', 'w') as file_2:
    file_2.write(file_1.read())
```

    - You should follow the same rule in the docstring part. For expected doctest results, remove the leading whitespaces:

```
def foo():
    """
    # Correct
    >>> foo()
    [[1, 2, 3], [1, 2, 3], \
[1, 2, 3]]

    # Incorrect
    >>> foo()
    [[1, 2, 3], [1, 2, 3], \
    [1, 2, 3]]
    """
    result = []
    for _ in range(3):
        result.append([1, 2, 3])
    return result
```

- **Magic Numbers**: Avoid using magic numbers (i.e. any number except 0, 1, -1) directly
    - If you need to use, say, 17 in your code, create a variable with a meaningful name and use the variable instead
    - The reason behind is that numbers have meanings. 17 might mean distance in some context and age in another context. You should define meaningful variable names to differentiate between the meanings.
    - Do not use a variable name like "SEVENTEEN" because it does not define the meaning clearly, so it will be considered as meaningless variable name (see the next rule)
    - Exceptions: We will not deduct points if magic numbers appear in your code in the following scenarios
    -

ALLOWED MAGIC NUMBER EXCEPTIONS
1. Checking for mod n, where n can be any number. You don't have to make a variable for the number n.
`number_to_check % 2 == 0 # Example for mod 2` `number_to_check % 3 == 0 # Example for mod 3`
2. The distance formula (2D & 3D examples are given, but the distance formula is ok for any dimension N).
`a ** 2 = b ** 2 + c ** 2  # 2D distance formula` `d ** 2 = x1 ** 2 + x2 ** 2 + x3 ** 2 # 3D distance formula`
3. Root formulas such as square root, cube root etc:
`square_root = a_number ** (1/2)` `cube_root   = a_number ** (1/3)`

- **Meaningless Variable Names**: All variable and function names should be descriptive
    - Function names typically evoke operations applied to arguments by the interpreter (e.g., print, add, square) or the name of the quantity that results (e.g., max, abs, sum);
    - Parameter names should evoke the role of the parameter in the function, not just the kind of argument that is allowed. For example, if the variable stores a list of student names, then it could be called student_names instead of lst;
    - Single letter parameter names are acceptable when their role is obvious, but avoid "l" (lowercase ell), "O" (capital oh), or "I" (capital i) to avoid confusion with numerals;
    - Try to only use i, j, k as index names and avoid them for other uses;
    - Avoid using built-in function names as variable names anywhere, as they break the built-in functions. For example, you should not use sum, dict, map, etc. as variable names.
- **Bad Variable Style**: You should always use snake_case when coding in Python
    - Variable and function names are lowercase, with words separated by underscores, e.g. unusual_sum()
    - Single-word names are preferred.
- **Indentation**: All indentations MUST be 4 spaces instead of TABs.
    - You can automatically convert tabs into spaces in the settings of your editor. Search for the editor setting “soft tabs” and set the soft tab length to 4;
    - Another indication is that when you upload to gradescope, tabs will have length 8 while 4 spaces will only have length 4.
- **Doctests**: For EACH function created, you should add **at least 3 doctest of your own** unless explicitly instructed otherwise.
    - You should think comprehensively about cases that would possibly break your code rather than testing it on easy-to-pass scenarios;
    - If you create any helper functions, you still need to add docstrings/doctests for them;
    - If you create inner functions, you only need to add docstrings (no doctests required);
    - It’s recommended to add as many doctests as you’re comfortable with to test the code thoroughly because your code correctness will be graded by hidden tests.

To further clarify the rules, we have compiled a set of example problems and solutions with an emphasis on style requirements. Feel free to ask any questions via Edstem.

Style Guide Examples

Submission

By the end of this homework, you should have submitted the homework via Gradescope. You may submit more than once before the deadline; only the final submission will be graded. Refer to Lab00 directions for submission.

Testing

At any point of the homework, use the following command to test your work:

```
>>> python3 -m doctest hw01.py
```

Part 2. Important Requirements and Questions

1. **DO NOT IMPORT ANY PACKAGES.**
2. **You should only use BASIC LOOPS (for/while) and/or CONDITIONAL STATEMENTS (if/elif/else) in your method structures.** Specifically, you should NOT use list/dictionary comprehensions, map(), filter(), zip(), reduce(), or lambda functions to solve the questions for this homework. Other built-in methods are allowed. If you don't recognize these names, that's totally fine. You can ask on ED if you have any doubts.
3. Add your own doctests (at least three per function) as the given doctests are not sufficient. **You will be graded on the doctests**. The doctests encompass **5 points** of your submission, so make sure you add them to each function!
4. Make sure you abide by the style guide outlined above. Style encompasses **10 points** of your submission, so make sure you follow it.
    a. Remember to add the docstring descriptions. They should be a brief description of the problem in your own words. Don't just copy and paste from the writeup.
    b. Remember to define magic numbers (anything other than -1,0,1) as variables.
    c. Remember to keep each line of code under 80 characters. This applies to the docstrings and doctests as well. You can define a ruler at 79 characters in your editor to make it easier to see if the length goes over.
5. **For this homework** you can assume that **the input given to you is valid and does not have any errors**. For example, if a function takes a positive integer we will only test your code on positive integers.

Question 0:

Quiz 0 (1 point)

Log in to Canvas and take Quiz 0 to refresh your understanding of the syllabus (you can find it on our website: dsc20.org). Make sure you understand how to find and take it. All reading quizzes will be there. The deadline for this quiz is the same as Homework 1's deadline.

Question 1:

Make sure that you completed part 0 (AI Form). Your assignment will NOT be graded without it.

Once you turned over the picture of the Pythonia and started reading, something odd had happened! All words disappeared and the warning appeared: "**JUMP!**". Write a function that takes three parameters: two strings and a non-negative integer. The strings represent numerical locations separated by a comma and a space. The third parameter is a jump length.

- If the distance between two locations is greater than a jump length, return a string "You are sent back home.",
- Otherwise return "Nice jump!
  You are safe to continue."

Hints:

- It is guaranteed that both strings will be separated by a comma and a space. What method can you use to extract the numbers? (Take a look at lab01)

Input	Output	Reason
`"1, 2", "4, 6", 10`	`Nice jump!` `You are safe to continue.`	* The distance from 1, 2 to 4, 6 is less than or equal to 10, so return 'Nice jump! …'

Question 1.1:

Write a (helper) function that takes in two lists of strings that represent coordinates and calculates (and returns) the Euclidean distance (see the formula below) between two given points. Notice that the parameter list is different from the original question. You do not need to round. For doctest purposes, you may round the returned result, see the last doctest for example.

$$d = \sqrt{(x_1 - x_2)^2 + (y_1 - y_2)^2}$$

```
def helper_distance(coord1, coord2):
    """
    >>> helper_distance(['1', '2'], ['4', '6'])
    5.0
```

```
    >>> round(helper_distance(['3', '2'], ['40', '6']), 3)
    37.216
    """
```

Question 1.2:

Now using your function from 1.1, solve the original question. In general, try to break a problem into smaller steps, write shorter helper functions and use them to solve a more complex problem.

```
def safe_location(your_location, safe_location, jump_length):
    """
    >>> your_loc = "1, 2"
    >>> safe_loc = "4, 6"
    >>> ans = safe_location(your_loc, safe_loc, 10)
    >>> print(ans)
    Nice jump!
    You are safe to continue.

    >>> your_loc = "3, 2"
    >>> safe_loc = "40, 6"
    >>> safe_location(your_loc, safe_loc, 15)
    'You are sent back home.'
    """
```

Question 2:

Congrats, you made it into a safe location and looked at the manuscript again. This time the letters form a new task asking you to "**Create a pattern**".

Question 2.1:

Write a function that takes two parameters: a number of repetitions (a positive integer) and a symbol (as a single-character string). Your function should return a string that consists of given characters, repeated a given number of times.

Hint: Lecture tells you how to do it.

```
def first_pattern(repeat, symbol):
```

```
    """
    Creates a string of symbols
    ---
    Parameters:
    repeat: positive integer
    symbol: a single character
    ---
    Returns a string of symbols repeated a given number of times.

    >>> first_pattern(3, '*')
    '***'
    >>> first_pattern(6, '$')
    '$$$$$$'
    """
```

Question 2.2:

Write a function that takes four parameters:

- `repeat`: a positive integer
- `symbol`: a single character (possibly empty)
- `word`: a string (possibly empty)
- `repeat_again`: a positive integer

Your function should return a string, where a symbol is repeated `repeat` number of times, then `word` is attached, the symbol is repeated `repeat_again` number of times and everything is in the **reverse** order.

```
def second_pattern(repeat, symbol, word, repeat_again):
    """
    >>> second_pattern(3, '*', 'pattern', 2)
    '**nrettap***'
    >>> second_pattern(6, '$', 'journey', 3)
    '$$$yenruoj$$$$$$'
    >>> second_pattern(3, '', '', 2)
    ''
    """
```

Question 2.3:

Write a function that takes two parameters: a string (might be empty) and a boolean. Your function should return a new string depending on the boolean parameter:

- If the `switch` is True, return a string where the `word` is repeated three times: first time in capitals, then lowercase and then capital letters again.
- If the `switch` is False, return a string where the `word` is repeated three times: first time in lowercase, then capitals and then lowercase letters again.

```
def third_pattern(word, switch):
    """
    >>> third_pattern("trip", True)
    'TRIPtripTRIP'
    >>> third_pattern("dangerous", False)
    'dangerousDANGEROUSdangerous'
    """
```

Question 2.4:

Write a function that takes three parameters: list of symbols (as characters), list of repetitions (as positive integers), and a boolean. Note that the lists can be of any length but lengths of these lists will always be the same. Your function returns a string containing characters from the symbols list repeated the corresponding number of times if the boolean parameter is True, and reversed otherwise. If both lists are empty, return an empty string.

For example,

Input	Output	Reason
`['-', '+'], [4, 5], True`	`----++++`	'-' is repeated 4 times, '+' is repeated 5 times and everything is put together.

```
def fourth_pattern(symbols, repeats, switch):
    """
    >>> fourth_pattern(['-', '+'], [4, 5], True)
    '----++++'
    >>> fourth_pattern(['-', '+'], [4, 5],  False)
    '++++----'
    """
```

Question 3:

You thought that the patterns would never end, but all of the sudden a new message appeared: "Be wise!". At the same time three bridges appeared: one was over flames, another was over a deep river and the third one was over a flowery field. You decided to take the shortest path but in a *wise* way.

Write a function that takes in three descriptions as strings and returns the shortest one. If there is a tie, you should return the bridge name associated with the shortest description that appears **first** in the parameter list.

```
def shortest_bridge(bridge1, bridge2, bridge3):
    """
    >>> shortest_bridge("flame", "deep river", "flower")
    'bridge1'
    >>> shortest_bridge("hot flames", "river", "flower")
    'bridge2'
    >>> shortest_bridge("flame", "river", "calm and green")
    'bridge1'
    >>> shortest_bridge("very dangerous", "deep river", "red flower")
    'bridge2'
    """
```

Question 4:

As you examine the bridges more closely, you'll notice that each one has a sign with some numbers. You assumed these are the number of people who have safely crossed each bridge. So you want to calculate the average and pick the bridge with the highest number. Again, you want to be *wise* about it and you will find an average without minimum and maximum value.

Very often, it is useful to split a problem into smaller problems and solve them one by one.

Question 4.1:

Write a function that takes a non-empty list of a number of people who crossed a bridge and calculates the average for the given list **according to our definition of**

the average above. Round the number to the second decimal place and return your answer. If the length of the list is less than or equal to two, then return 0.0

Hints:

- `min()`, `max()` might be useful
- `sum()` might be useful
- Use `round()` built-in function.

Input	Output	Reason
`[2, 4, 10, 7, 9]`	`6.67`	Drop 2 (minimum) and 10 (maximum). The average is `(4 + 7 + 9)/3 = 6.6666…` Rounded to the second decimal place is **6.67**

```
def average_without_min_max(people):
    """
    >>> average_without_min_max([10, 10, 1, 16])
    10.0
    >>> average_without_min_max([2, 4, 10, 7, 9])
    6.67
    >>> average_without_min_max([2, 0, 10, 5])
    3.5
    """
```

Question 4.2:

Once you tested your function for a single list, you can re-use it to solve the original problem. This time you are given a *people* list of non-empty lists, where each list represents different bridges and a number of people crossed it. You are also given a list of names of the bridges corresponding to the list of people.
Write a function that outputs a name of the bridge with the highest average (as described above). You can assume that the lengths of both lists are the same.
Special instructions:

- If there is a tie, return the highest average that appears first.
- In case of an empty list, return "No bridges to choose from".
- If all the bridges have average rating 0.0, treat 0.0 as the best rating and return the first route.

Hints:

- You should reuse the function from Question 4.1
- max() will be helpful
- The list's `.index()` might be helpful (look up the documentation!)
- The function should work with *any* number of bridges.

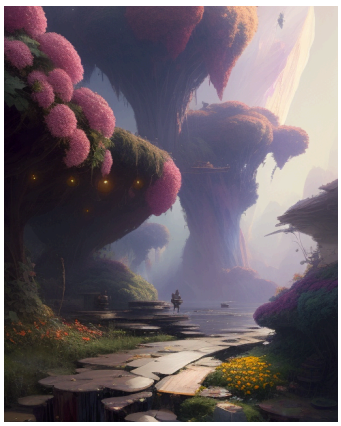
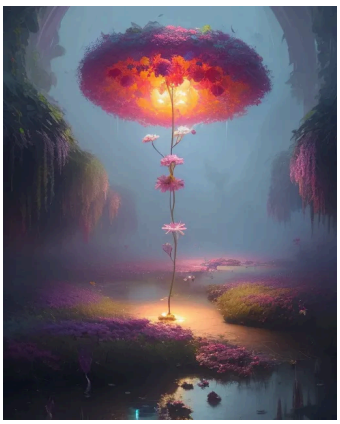
```
def best_average_bridge(people, bridges):
    """
    Finds the best bridge name based on the average
    ---
    >>> ratings = [[1, 10, 5], [4, 6, 9]]
    >>> names = ["flames", "river"]
    >>> best_average_bridge(ratings, names)
    'river'

    >>> ratings = [[6, 10, 5, 3, 8], [10, 10, 9], [6, 6, 90]]
    >>> names = ["flames", "flower", "river"]
    >>> best_average_bridge(ratings, names)
    'flower'

    >>> ratings = [[], [], []]
    >>> names = ["flames", "flower", "river"]
    >>> best_average_bridge(ratings, names)
    'flames'

    >>> best_average_bridge([], [])
    'No bridges to choose from'
    """
```

Question 5:

	
Awakentia - a flower that wakes people up	*Sleeperose* - a flower that puts everyone to sleep.

Your calculations have shown to you that the bridge over the flower field is the safest in terms of the average number of people who crossed it, so you decided to follow it as well. But the longer you walked, the stranger you started to feel. The last thing you remember is you falling down the flower bridge. When you woke up, you noticed many flowers around you with a strong smell. It helped you to run away from this trap and you found yourself back in front of the bridges.

Write a function that takes a list of flower names (as strings) and returns:

- '*Awakentia*' if there are more *Awakentia* type in the list
- '*Sleeperose*' if there are more *Sleeperose* type in the list
- True, if there is a tie and the number of these flowers are positive
- False, if none of these flowers appear in the list.

Note: The strings are case-sensitive, meaning that we only consider `*Awakentia*` instead of `*awakentia*`, `AWEKENTIA`, etc.

Hints:

1. You may initialize variables that track the numbers for '*Awakentia*' and '*Sleeperose*';
2. Another option is to use a built-in function `count()`

```
def sleepy_vs_awake(flowers):
    """
    >>> sleepy_vs_awake(["Sleeperose", "Rose", "Sleeperose"])
    'Sleeperose'
    >>> sleepy_vs_awake(["Sleeperose", "Awakentia", "rose"])
    True
    >>> sleepy_vs_awake(["rose", "sleeperose", "awakentia", "daisy"])
```

```
    False
    >>> sleepy_vs_awake(["rose", "sleeperose", "Awakentia", "daisy"])
    'Awakentia'
    """
```

Question 6:

Since many people want to get to Pythonia, you want to save as many as possible from falling off the bridges. Therefore, you put up a sign that helps others.

Write a function that takes in the type of the bridge, hint, and your name, as strings. It then returns a sign as a string. Assume all inputs are valid and they might be empty strings, but you do not need to make a separate case for them.

When you `print` the *returned* string to the console, it should look like this:

> Dear traveler,
> Please be careful with the <*type of the bridge*>. My advice is: <*hint*>
>
> See you in Pythonia: <*your name*>

Notes:

1. The token **<BLANKLINE>** in the doctest denotes a blank line in the output. You should not append this token in the returned string. You should always add this token in your **doctest** if you expect the output to have a blank line, instead of just hitting the Enter key;
2. To achieve this, try to search how to change to a new line in a string;
3. Note in the doctest we are *printing* the results returned from the function, so you should not have print() statements in the function;
4. There is a single space after each colon.
5. If you find your answer exactly the same as the expected answer but the doctest fails, try to check extra whitespaces.

```
def sign(bridge_type, hint, your_name):
```

```
    """
    >>> print(sign("Flower bridge", "Do the opposite.", "Dr. Who"))
    Dear traveler,
     Please be careful with the Flower bridge. My advice is: Do the
opposite.
    <BLANKLINE>
    See you in Pythonia: Dr. Who
    >>> print(sign("Flame Bridge", "Do not be afraid of it.", ""))
    Dear traveler,
    Please be careful with the Flame Bridge. My advice is: Do not be \
afraid of it.
    <BLANKLINE>
    See you in Pythonia:
    """
```

Question 7:

Finally, you made it to another side, safe and sound. You were about to step into a new world but your path is blocked and the manuscript said that you need to create your own nickname in order to enter. Write a function that takes three parameters: first and last names (as strings) and an age (as in integer). Your function should return:

- 'Error', if the first and/or last names are not strings (either or both are not strings).
- 'Error' ,if the age parameter is not integer, less than 0, or greater or equal to a 100.
- nickname (as a string) constructed in the following way:
    - Every other character is taken (starting from index 1) from the first name.
    - Every letter in the last name is capitalized.
    - Age is split, and the digit on the tenth place appears at the beginning, and the digit on the one's place appears at the end of the output.
    - The resulting characters are put together to create a nickname as a string, then return it.

```
def my_nickname(first, last, age):
    """
    >>> my_nickname("Marina", "Langlois", 25)
    '2aiaLANGLOIS5'
```

```
>>> my_nickname("Marina", "Langlois", 2.5)
'Error'
>>> my_nickname("", "Langlois", 25)
'2LANGLOIS5'
>>> my_nickname("", "Langlois", 5)
'0LANGLOIS5'
"""
```

Submission

By the end of this homework, you should have submitted `hw01.py` via Gradescope. You may submit more than once before the deadline; only the final submission will be graded.

Important Notes:

1. You may submit more than once before the deadline; only the final submission will be graded;
2. We will be grading this lab solely based on hidden tests, so you should test your code thoroughly by writing more test cases (encouraged to go over the required 3, add as many as you like);
3. Unlike lab00, you should expect to see "-/100.0". You will see your score when the grades are released;
4. You should **wait for the Autograder to finish** running before leaving the site to ensure that the tests are properly run;
5. If your autograder fails to run, try to consult the Gradescope Common Errors section on the course website before making an ED post;
6. If you have any other questions, please post to ED.

Warning: If you used Virtual Studio Code as your editor, please double check if there are weird import statements that you don't recognize before submitting. Make sure to remove these as they will fail Gradescope runs!