STA 141B Assignment 1

Due **October 13, 2023** by **11:59pm**. Submit your work by uploading it to Gradescope through Canvas.

Please rename this file as **"LastName_FirstName_hw1"** and export it as as pdf-file.

The objective of this assignment is to solidify your understanding of programming fundamentals: defining functions, conditional statements, loops, and recursion. Subsequent assignments will delve deeper into data science topics.

Instructions:

1. Provide your solutions in new cells following each exercise description. Create as many new cells as necessary. Use code cells for your Python scripts and Markdown cells for explanatory text or answers to non-coding questions. Answer all textual questions in complete sentences.
2. Prioritize code readability. Just as in writing a book, the clarity of each line matters. Adopt the **one-statement-per-line** rule. If you have a lengthy code statement, consider breaking it into multiple lines for clarity. (Please note: violating the one-statement-per-line rule will result in a one-point deduction for each offending line.)
3. To help understand and maintain code, you should always add comments to explain your code. Use the hash symbol (#) to start writing a comment (homework without any comments will automatically receive 0 points).
4. Submit your final work as a **.pdf** file on **Gradescope**. To convert your .ipynb file into one of these formats, navigate to "File", select "Download as", and then choose either "PDF via LaTeX" or "HTML". If "PDF via LaTeX" does not work for you, export to "HTML", and then use Chrome to print the .html file into PDF. Gradescope only accepts PDF files.
5. This assignment will be graded on your proficiency in programming. Be sure to demonstrate your abilities and submit your own, correct and readable solutions.

Writing Functions

Exercise 1.a (2 points). Write a function `my_quantile` that takes a list of numbers as input and returns their 0.75-quantile (third quartile) as output. The 0.75-quantile is the value below which 75% of the data falls.

Use the following method to implement the quantile:

- Use the median to divide the ordered data set into two-halves. Implement this fuction by yourself.
- If there is an odd number of data points in the original ordered data set, '''do not include''' the median (the central value in the ordered list) in either half.
- If there is an even number of data points in the original ordered data set, split this data set exactly in half.
- The lower quartile value is the median of the lower half of the data. The upper quartile value is the median of the upper half of the data.

Hint: You may want to use the `sorted` function and appropriate indexing.

Exercise 1.b (2 points). For the function you wrote in Exercise 1.a, what happens if the input list is empty or contains non-numeric elements? Modify your function to handle these cases. Your updated function, named `better_quantile`, should return `None` for these unusual cases. Ensure that the function doesn't print the string "None" – the output should be genuinely empty in these cases.

Hint: Think about how you can check the type of each element in the list. A similar problem is discussed in Section 6.8 of Think Python.

Exercise 1.c (1 point). Familiarize yourself with writing docstrings by reading Section 4.9 of Think Python. Enhance your function from Exercise 1.b by adding a docstring that elucidates its purpose, input, and output. Name this function `best_quantile`. Illustrate that your docstring can be accessed using Python's inbuilt `help` function.

Fibonacci Words

A Fibonacci word is a specific sequence of 0s and 1s constructed by concatenating strings in a unique manner. The generation starts with:

```
S0 = "0"
S1 = "01"
```

Subsequent sequences are produced by concatenating the previous two sequences. For instance, `S2` is derived from concatenating `S1` and `S0`, which results in:

```
S2 = "010"
S3 = "01001"
... and so forth.
```

Exercise 2 (3 points).

Task: Write a function named `fib` that computes the Fibonacci words. Your function should accept an argument `n`, where `n` indicates the position of the Fibonacci word to compute. The command `fib(3)` should return `'01001'`.

Output: Use your function to display the first 10 Fibonacci words.

Hint: You may find it helpful to use multiple assignment to swap the values of variables.

Finding Exponential Roots

The Newton-Raphson algorithm is an algorithm for finding the zeroes of a mathematical function. We can use the Newton-Raphson algorithm to find zeroes of the function

$$f(x) = x^p e^x - c$$

where p and c are constants. For example, if we choose $p = 2$ and $c = 5$, the Newton-Raphson algorithm finds solutions to

$$0 = x^2 e^x - 5$$

In other words, we can use the algorithm to find square roots. By changing p, we can also find other kinds of exponential roots.

The algorithm works by starting from an initial guess x_0 and then iteratively evaluating

$$x_{n+1} = x_n - \frac{f(x_n)}{f'(x_n)}$$

for $n = 0, 1, 2, \ldots, N$ until reaching a result X_{N+1} with acceptable accuracy. The initial guess does not need to be an excellent guess, but can affect which zero is found.

For our specific function f, note that $f'(x) = (x + p)x^{p-1}e^x$.

Exercise 3 (7 points). Write a function `root` that uses the Newton-Raphson algorithm to compute one of the p-th roots for a constant c. Your function does not need to find complex roots, only real roots. Your function should have arguments

- `c`, the constant,
- `p`, the power,
- `x0`, the initial guess,

- `tol = 10e-4`, the tolerance with provided default value and
- `N`, the maximum number of iterations.

The function should iterate until converence, where we define convergence as `abs(x_new - x_old) < tol`. If no convergence occurs in `N` iterations, the function should print a warning.

`root` should return the approximated x-value and the number of iterations in a tuple.

Test:

```
> root(5, 2, -8)
No convergence attained!
(202.97084459397027, 100)

> root(5, 2, -1)
(1.2168715486319333, 8)

> root(5, 2, 1)
(1.2168714891788939, 4)

> root(2, 2, 100, 0.1, 200)
(0.9020685461433838, 110)

> root(10, 10, -8)
(-32.51417697618235, 14)

> root(10, 10, -1)
(1.1249754478019025, 10)
```

Run the test cases and report the return of the calls: `root(1, 2, 1)`, `root(0, 1, 1)` and `root(2, 1, -7)`.

Hint: You may find it helpful to define the function and its derivative using `lambda` functions. You may use `import numpy`, and `numpy.exp` and `numpy.log`. Also, remember that log-transforming can improve numerical stability!