Assignment 04
Due Wednesday October 9th at 10:00am EST (no late submissions)

Assignment Guidelines:

- This assignment covers material in Module 4 up to slide 37 (ie. No abstract list functions).
- Submission details:
  - Solutions to these questions must be placed in files `a04q1.py`, `a04q2.py`, `a04q3.py`, and `a04q4.py`, respectively, and must be completed using Python 3.
  - Download the interface file from the course Web page to ensure that all function names are spelled correctly and each function has the correct number and order of parameters.
  - All solutions must be submitted to MarkUs. No solutions will be accepted through email, even if you are having issues with MarkUs.
  - Verify using MarkUs and your basic test results that your files were properly submitted and are readable on MarkUs.
  - For full style marks, your program must follow the Python section of the CS116 Style Guide.
  - Be sure to review the Academic Integrity policy on the Assignments page
  - Helper functions need design recipe elements but not examples and tests.
- Download the testing module from the course web page. Include `import check` in each solution file.
  - When a function produces a floating point value, you *must* use `check.within` for your testing. Unless told otherwise, you may use a tolerance of 0.00001 in your tests.
  - Test data for all questions will always meet the stated assumptions for consumed values.
- Restrictions:
  - Do not import any modules other than `math` and `check`.
  - You are always allowed to define your own helper functions, as long as they meet the assignment restrictions. Do **not** use Python constructs from later modules (e.g. abstract list functions or loops) or the command `zip`. Use only the functions and methods as follows:
    - `abs`, `len`, `max` and `min`
    - Any method or constant in the `math` module
    - Type casting including `int()`, `str()`, `float()`, `bool()`, `list()`
    - The command `type()`
    - Any basic arithmetic operation (including `+, -, *, /, //, %, **`)
    - Any basic logical operators (`not`, `and`, `or`)
    - String or list slicing and indexing as well as string or list operations using the operators above as used in module 4
    - Any string or list methods (including the `in` operation)
    - `input` and `print` as well as the formatting parameter `end` and method `format`. Note that all prompts must match exactly in order to obtain marks so ensure that you do not alter these prompts
  - Do **not** mutate any passed parameters unless instructions dictate otherwise.
  - While you may use global *constants* in your solutions, **do not** use global *variables* for anything other than testing.
  - Read each question carefully for additional restrictions.
  - **The solutions you submit must be entirely your own work. Do not look up either full or partial solutions on the Internet or in printed sources.**

1 Tsunami

Write a function

`tsunami(s)`

which consumes a string consisting of only lower case Latin alphabet letters and returns a list of strings where for each index *i*, the *i*th string in the list has exactly the *i*th letter capitalized.

Reminder for this entire assignment, do **not** use abstract list functions.

Sample:

```
tsunami("word") => ["Word", "wOrd", "woRd", "worD"]
```

2 Reversals

Write a function

`reversal(s)`

that takes in a string s of single space separated words (with no spaces at the beginning or end) and your goal is to return a string with each individual word reversed.

Sample

```
reversal("live on time") => "evil no emit"
```

HINT: Check out what the string method `split()` does.

3 First and Last

Write a function

`first_and_last(L)`

which consumes a list L of strings, mutates the list so that every non-empty string that begins and ends with the same character is removed and returns the number of such values that were removed.

Sample:

```
L = ["abba", "banana", "dumpling", "level", "trait"]
first_and_last(L) => 3
```

and L is mutated to `["banana", "dumpling"]`

4 Card Sharks

In 2019, a revival of the popular game show *Card Sharks* aired once again on ABC. In this question, we will simulate a simplified version of their bonus round. For this problem, we use the following data definition:

```
## A card is a natural number from 1 to 13
```

In the bonus round, a contestant is given a list of cards which the contestant views one card at a time in an attempt to determine whether or not subsequent cards are higher or lower than the previous cards. The contestant is given an amount of money to wager on each turn of the cards in an effort to make more money.

In this problem, the contestant will always start with $1000. On each card the contestant makes a wager as to whether or not they believe the next card is higher or lower than the previous card (remember contestants only see one card at a time). They pick higher or lower and if correct, they win their wager. If the cards are equal, also known as a *push*, the contestant neither wins nor loses money. If the contestant is wrong, they lose their wager. Use the natural order of the natural numbers when deciding if the next card card is higher or lower than the previous card. The contestant must bet non-negative integer amounts per card and you cannot bet on a tie.

Play continues until the contestant runs out of cards. Notice that if you run out of money, the game will continue to play even though the contestant will only be able to bet 0 dollars and will not be able to make any money.

Write a function

`bonus_round(cards)`

that plays this game given a list of `card` values of size at least 2 (in the parameter `cards`) and returns the amount of money the player has at the end. Your interface file includes prompts for entering a wager and for determining whether the next card is higher or lower. You will also need to print text corresponding to:

- First card and last card texts (print the last card text after the first card text if there are only two cards)
- The wager and higher/lower prompts
- The next card value
- Being correct, incorrect, or tied (push)
- Current dollar amount

The sample below shows all possible interactions a user can have with your game that you must be able to replicate. You may assume that all user inputs are in accordance with the above description. In particular, the input for wagers will always be in the specified range of numbers given (even if both are 0) and the words `higher` or `lower` will be given when prompted. You may test your function with either `set_print_exact` or `set_screen`.

Sample:

```
bonus_round([1,2,3,1,5,5]) => 5000
```

given the following interactions:

```
Your first card is a 1.
Enter a wager between 0 and 1000: 1000
Is the next card higher or lower than 1? higher
The card is a 2.
You are correct! You now have 2000 dollars.
Enter a wager between 0 and 2000: 2000
Is the next card higher or lower than 2? higher
The card is a 3.
You are correct! You now have 4000 dollars.
Enter a wager between 0 and 4000: 1500
Is the next card higher or lower than 3? higher
The card is a 1.
You are incorrect. You now have 2500 dollars.
Enter a wager between 0 and 2500: 2500
Is the next card higher or lower than 1? higher
The card is a 5.
You are correct! You now have 5000 dollars.
Last card.
Enter a wager between 0 and 5000: 2500
Is the next card higher or lower than 5? lower
The card is a 5.
Push. You now have 5000 dollars.
```

Note that each line above ends with a newline character.