Problem 1. (*Palindrome*) Implement the function `_isPalindrome()` in `palindrome.py` such that it returns `True` if the argument s is a palindrome (ie, reads the same forwards and backwards), and `False` otherwise. You may assume that s is all lower case and doesn't include any whitespace characters.

```
>_ ~/workspace/module4/assignment4
$ python3 palindrome.py bolton
False
$ python3 palindrome.py madam
True
```

Directions:

- Repeat for each $i \in [0, n/2]$, where n is the number of characters in s:
  - If the character at i is different from the character at $n - i - i$, then s is not a palindrome, so return `False`.
- s is a palindrome, so return `True`.

Problem 2. (*Reverse*) Implement the function `_reverse()` in `reverse.py` that reverses the one-dimensional list a in place, ie, without creating a new list.

```
>_ ~/workspace/module4/assignment4
$ python3 reverse.py to be or not to be that is the question
question the is that be to not or be to
```

Directions:

- Repeat for each $i \in [0, n/2]$, where n is the number of elments in a:
  - Exchange the element at i in a with the element at $n - i - 1$.

Problem 3. (*Euclidean Distance*) Implement the function `_distance()` in `distance.py` that returns the Euclidean distance between the vectors x and y represented as one-dimensional lists of floats. The Euclidean distance is calculated as the square root of the sums of the squares of the differences between the corresponding entries. You may assume that x and y have the same length.

```
>_ ~/workspace/module4/assignment4
$ python3 distance.py 2 1 0 0 1
1.4142135623730951
$ python3 distance.py 5 -9 1 10 -1 1 -5 9 6 7 4
13.0
```

Directions:

- Set *distance* to 0.
- Repeat for each $i \in [0, n - 1]$:
  - Add square of $x[i] - y[i]$ to *distance*.
- Return the square root of *distance*.

Problem 4. (*Transpose*) Implement the function `_transpose()` in `transpose.py` that creates and returns a new matrix that is the transpose of the matrix represented by the argument a. Note that a need not have the same number rows and columns. Recall that the transpose of an m-by-n matrix A is an n-by-m matrix B such that $B_{ij} = A_{ji}$, where $0 \leq i < n$ and $0 \leq j < m$.

```
>_ ~/workspace/module4/assignment4
$ python3 transpose.py 2 3 1 2 3 4 5 6
1.0 4.0
2.0 5.0
3.0 6.0
```

Directions:

- Set c (the transpose of a) to a 2D list with n rows and m columns, with all the elements set to 0.0.
- Repeat for each $i \in [0, n-1]$:
  - Repeat for each $j \in [0, m-1]$:
    - Set $c[i][j]$ to $a[j][i]$.
- Return c.

Problem 5. (*Password Checker*) Implement the function `_isValid()` in `password_checker.py` that returns `True` if the given password string meets the following requirements, and `False` otherwise:

- Is at least eight characters long
- Contains at least one digit (0-9)
- Contains at least one uppercase letter
- Contains at least one lowercase letter
- Contains at least one character that is neither a letter nor a number

```
>_ ~/workspace/module4/assignment4
$ python3 password_checker.py Abcde1fg
False
$ python3 password_checker.py Abcde1@g
True
```

Directions:

- If *pwd* is long enough, set corresponding flag to `True`.
- Repeat for each character $c \in pwd$:
  - If c is a digit, set corresponding flag to `True`.
  - Otherwise, if c is in upper case, set corresponding flag to `True`.
  - Otherwise, if c is in lower case, set corresponding flag to `True`.
  - Otherwise, if c is not alphanumeric, set corresponding flag to `True`.
- Return `True` if all the flags are `True`, and `False` otherwise.

Problem 6. (*Spell Checker*) Write a program `spell_checker.py` that accepts words (one per line) from standard input; looks up each word in the file `data/misspellings.txt` that maps misspelled words to their correct spellings; and if it exists (ie, is misspelled), writes the word to standard output along with the correct spelling.

```
>_ ~/workspace/module4/assignment4
$ python3 spell_checker.py
seperate <enter>
seperate -> separate
sucess <enter>
sucess -> success
```

```
<ctrl-d>
```

Directions:

- Set *inStream* to an input stream built from the file `data/misspellings.txt`.
- Set *lines* to the list of lines read from *inStream*.
- Set *misspellings* to a new `dict` object.
- Repeat for each $line \in lines$:
  - Strip *line* of the newline character at the end.
  - Set *tokens* to the list obtained by splitting line.
  - Insert the key/value pair *tokens*[0]/*tokens*[1] into *misspellings*.
- Set *word* to a word read from standard input.
- Repeat as long as *word* is not the `EOF` character:
  - Strip *word* of the newline character at the end.
  - If *word* exists in *misspellings*, then it is misspelled. So write *word* and its correction to standard output, separated by the string `" -> "`.
  - Set *word* to the next word read from standard input.

Problem 7. (*Word Occurrences*) Write a program `word_occurrences.py` that accepts $filename$ (str) as command-line argument and words from standard input (one per line); and writes to standard output the word along with the indices (ie, locations) where it appears in the file whose name is $filename$ — writes "Word not found" if the word does not appear in the file.

```
>_ ~/workspace/module4/assignment4
$ python3 word_occurrences.py data/beatles.txt
dead <enter>
dead -> [3297, 4118, 4145, 4197]
parrot <enter>
Word not found
world <enter>
world -> [46, 56, 112, 122, 172, 182, 1769, 3587, 3596, 3695, 6785, 6795, 6851, 6861, 6911, 6921]
<ctrl-d>
```

Directions:

- Set *inStream* to an input stream built from $filename$.
- Set *lines* to the list of lines read from *inStream*.
- Set *occurrences* to a new `dict` object.
- Set *index* to 0.
- Repeat for each $line \in lines$:
  - Strip *line* of the newline character at the end.
  - Set *words* to the list obtained by splitting line.
  - Repeat for each $word \in words$:
    - If *word* does not exist in *occurrences*, insert it as the key with an empty list as the corresponding value.
    - Append *index* to the list corresponding to *word*.
    - Increment index by 1.
- Set *word* to a word read from standard input.

- Repeat as long as *word* is not the `EOF` character:
  - Strip *word* of the newline character at the end.
  - If *word* exists in *occurrences*, write *word* and the corresponding list to standard output, separated by the string `" -> "`.
  - Otherwise, write the string `"Word not found"`.
  - Set *word* to the next word read from standard input.

Files to Submit

1. `palindrome.py`
2. `reverse.py`
3. `distance.py`
4. `transpose.py`
5. `password_checker.py`
6. `spell_checker.py`
7. `word_occurrences.py`