Assignment 2

General Instructions:

1. Create an Eclipse Python project called: "Assignment2".
2. Copy all the files included in the Assignment zip file which include:
    - A2_template.py (You need to rename to A2_solution.py)
    - A2_test.py (testing file)
    - A2_output.txt (sample output)
    - utilities_tempate.py (rename to utilities.py)
    - engmix.txt (a dictionary file)
    - 3 plaintext files
    - 6 ciphertext files
3. Enter your credentials on top of the file: A2_solution.py. No submission will be accepted without credentials.
4. Your need to submit ONLY ONE file: called: "A2_solution.py". Do not export from eclipse or submit any additional files.
5. Deadline: Tuesday, October 20 at 11:55 pm.
6. You may not include any packages/libraries other than the ones already included.
7. You may create your own utility functions, as long you write them in the file: A2_solution.py. Add the function under the relevant task section. Every new function should have a docstring outlining input parameters, return values and a description.
8. Make sure to test your solution using the given test file: "A2_test.py". You may edit your local version of the test file. However, since you will not be submitting the testing file, such changes will be discarded.
9. Your output should exactly match the output provided in A2_output.txt. You will lose points for any mismatch including missing a space or a dot. Use a text comparison tool https://text-compare.com/ to verify that you have a matching output.

Grading Rubric:

Task	Points	Comments
Task 1: Utilities	1.2 pts	6 functions each is 0.2 pts
Task 2: Block Rotation Cipher	2.6 pts	Adjust_key (0.3), encryption (0.8), decryption (0.5), cryptanalysis (1)
Task 3: Wheatstone Playfair Cipher	3.8 pts	Format(0.8), restore (0.7), restore_word(0.7), encryption (0.8), decryption (0.8)
Task 4: Columnar Transposition	2.4 pts	Get_order (0.4), encryption (1) , decryption (1)

Task 1: Utility Functions (1.2 pts)

Provided Utilities:

The file utilities.py contain the following functions:

```
1- get_base(base_type)
2- analyze_text(text,dict_list)
3- is_plaintext(text,dict_list,threshold)
4- get_playfair_square()
```

The first three functions are similar to those provided in A1, with two minor modifications. First, the get_base is supported with more base types. Second the analyze_text and is_plaintext functions now receive a dictionary list instead of a dictionary file. This should enhance the efficiency as it avoids continuous reading from the file.

You need to copy the following functions from A1 into the utilities.py file:

```
5- file_to_text(filename)
6- text_to_file(text,filename)
7- new_matrix(r,c,fill)
8- load_dictionary(dict_file)
9- text_to_words(text)
```

The above functions need no modifications. In total, you should have nine functions in your utilities.py file.

For Task 1, you need to implement 6 utility functions. These functions will be useful for this assignment and future assignments. These utility functions should NOT be added under utilities.py file. Instead, you should add them under Task 1 in A2_solution.py.

[1] Get Positions

Implement the function: `get_positions(text,base)`. The function scans the given text for the characters defined in base. The function constructs and returns a 2D list containing all found characters along with their respective positions in the text.

Observe the following examples:

```
str1 = One Must Acknowledge With Cryptography No Amount of Violence Will Ever Solve
a Math Problem.

get_positions(str1,get_base("upper")) -->
[['O', 0], ['M', 4], ['A', 9], ['W', 21], ['C', 26], ['N', 39], ['A', 42], ['V',
52], ['W', 61], ['E', 66], ['S', 71], ['M', 79], ['P', 84]]
```

```
get_positions(str1,get_base("vwxyz")) -->
[['w', 14], ['y', 28], ['y', 37], ['v', 67], ['v', 74]]

get_positions(str1,get_base("special")) --> [['.', 91]]
```

No changes should be applied to the input parameters. If the function finds no characters. It returns an empty list.

[2] Clean Text:

Implement the following function:

```
"""
----------------------------------------------------
Parameters:   text (str)
              base (str)
Return:       updated_text (str)
Description:  Removes all base characters from text
Asserts:      text and base are strings
---------------------------------------------------
"""
def clean_text(text,base)
```

Below is sample testing using the same string used in the get_positions testing:

```
clean_text(str1,get_base("upper")) =
ne ust cknowledge ith ryptography o mount of iolence ill ver olve a ath roblem.

clean_text(str1,get_base("lower")) =
O M A W C N A  V W E S  M P.

clean_text(str1,"abcdefghij") =
On Must Aknowl Wt Cryptorpy No Amount o Voln Wll Evr Solv  Mt Prolm.

clean_text(str1," \n") =
OneMustAcknowledgeWithCryptographyNoAmountofViolenceWillEverSolveaMathProblem.
```

[3] Insert Positions:

Implement the following function:

```
"""--------------------------------------------------
Parameters:   text (str)
              positions (lsit): [[char1,pos1],[char2,pos2],...]]
Return:       updated_text (str)
Description:  Inserts all characters in the positions 2D list into their respective
Asserts:      text is a string and positions is a list
----------------------------------------------  """
```

```
def insert_positions(text, positions)
```

The function receives a string (text) along with a 2D list generated by the get_positions function. The function inserts all given characters in their given order. The function is sequential in its operation; meaning it inserts the first character (which updates the text length) and then inserts the second character (which also updates the text length) until all characters are inserted.

[4] Text to Blocks:

Implement the following function:

```
"""--------------------------------------------------------
Parameters:   text (str)
              block_size (int)
              padding (bool): False/default = no padding, True = padding
              pad (str): default = PAD
Return:       blocks (list)
Description:  Create a list containing strings each of given block size
              if padding flag is set, pad using given padding character
              if no padding character given, use global PAD
Asserts:      text is a string and b_size is a positive integer
------------------------------------------------------"""
def text_to_blocks(text,b_size,padding = 0,pad =PAD)
```

Below is a sample testing:

```
str6 = 'Cryptography is the fun part of math'
Testing text_to_blocks:
text_to_blocks(str6,4) →
      (['Cryp', 'togr', 'aphy', ' is ', 'the ', 'fun ', 'part', ' of ', 'math']
text_to_blocks(str6,5,0) →
      ['Crypt', 'ograp', 'hy is', ' the ', 'fun p', 'art o', 'f mat', 'h']
text_to_blocks(str6,7) →
      ['Cryptog', 'raphy i', 's the f', 'un part', ' of mat', 'h']
text_to_blocks(str6,8,1) →
      ['Cryptogr', 'aphy is ', 'the fun ', 'part of ', 'mathqqqq']
text_to_blocks(str6,10,1,'x') →
      ['Cryptograp', 'hy is the ', 'fun part o', 'f mathxxxx']
```

[5] Shift String:

Implement the following function:

```
"""--------------------------------------------------------
Parameters:   text (string): input string
              shifts (int): number of shifts
              direction (str): 'l' or 'r'
```

```
Return:        update_text (str)
Description:   Shift a given string by given number of shifts (circular shift)
               If shifts is a negative value, direction is changed
               If no direction is given or if it is not 'l' or 'r' set to 'l'
Asserts:       text is a string and shifts is an integer
-----------------------------------------------"""
def shift_string(s,n,d='l'):
```

Below is a sample testing:

```
shift_string(str6,4) →
tography is the fun part of mathCryp

shift_string(str6,-4) →
mathCryptography is the fun part of

shift_string(str6,5,'l') →
ography is the fun part of mathCrypt

shift_string(str6,5,'r') →
 mathCryptography is the fun part of

shift_string(str6,6,'k') →
graphy is the fun part of mathCrypto

shift_string(str6,300,) →
 is the fun part of mathCryptography
```

[6] 2D Index:

Implement the following function:

```
""" -----------------------------------------------------
Parameters:    input_list (list): 2D list
               item (?)
Return:        i,j (int,int)
Description:   Performs linear search on input list to find "item"
               returns i,j, where i is the row number and j is the column num
               if not found returns -1,-1
Asserts:       input_list is a list
-----------------------------------------------------"""
def index_2d(input_list,item):
```

Below is a sample testing:

```
list1 = [[1, 2, 3], [4, 5], [6, 7, 8, 9, 10], [11], [12, 13]]
2 is at (0, 1)
5 is at (1, 1)
6 is at (2, 0)
11 is at (3, 0)
```

```
12 is at (4, 0)
18 is at (-1, -1)
```

Task 2: Block Rotation Cipher (2.6 pts)

In this task you will implement the encryption, decryption and cryptanalysis of Block Rotation Cipher. The process will involve implementing four functions.

The Block Rotation Cipher uses a key of the format (b, r), where b is the block size and r is the number of rotations. All rotations are considered left circular shifts.

Start by implementing the following helper function:

```
"""---------------------------------------------------
Parameters:   key (b,r): tuple(int,int)
Return:       updated_key (b,r): tuple(int,int)
Description:  Private helper function for block rotate cipher
              Update the key to smallest positive value
              if an invalid key return (0,0)
---------------------------------------------------"""
def _adjust_key_block_rotate(key)
```

Below are sample tests for the above function:

```
(4,3.5)    --> (0, 0)
[4, 5]     --> (0, 0)
10         --> (0, 0)
(-2,1)     --> (0, 0)
(5,7)      --> (5, 2)
(3,11)     --> (3, 2)
(7,-6)     --> (7, 1)
(11,4)     --> (11, 4)
```

The above function should be invoked in the encryption and decryption functions. Inside these functions, if an invalid key is passed, an error message is printed and an empty string is returned.

Both encryption and decryption functions assume the use of padding for the last block. The default PAD constant is used. For details on how Block Rotate Cipher work, refer to your class notes.

In both functions, use (as necessary) the functions `get_positions`, `clean_text`, `insert_positions` and `text_to_blocks` that were developed in Task 1.

Note that the encryption and decryption process applies to all characters, except the newline character.

The cryptanalysis function receives a ciphertext and a list of arguments. The arguments represent some partial information that you know about the key.

There are three defined arguments [b0,bn,r]. The first argument is b0 denotes the minimum block size, bn the maximum block size and r is the rotations. When none of these arguments are known the value of the arguments will be [0,0,0].

Design your cryptanalysis to cover the various scenarios, which includes:

- knowing the block size, but not the rotations. This happens when values of b0 and bn are equal (and are both positive values), and value of r is 0.
- Knowing the rotation, but not the block size. This happens when values of b0 and bn are 0, and value of r is a positive number.
- Knowing the minimum or maximum block size. The rotations could be known or unknown for each scenario.

Whenever the value of bn is undefined (i.e. value of 0), the BLOCK_MAX_SIZE value should be used. Also, the value of b0 should not be less than 1 or more than bn.

Your cryptanalysis should use brute force efficiently, i.e. without unnecessary repetitions. However, you are not expected to use factorization in this task.

Task 3: Wheatstone Playfair Cipher (3.8 pts)

The Wheatstone Playfair Cipher is a relatively simple cipher that relies on biograph substitutions. However, the implementation of this scheme poses several challenges. In this task, you will build the encryption and decryption methods, along with supporting utility tools.

Both the encryption and decryption functions use a playfair square list as the key. This is generated through the utility function `get_playfair_square()`. This square contains 25 cells containing upper case alphabetical characters except `W`.

Encryption Process:

The encryption function receives a plaintext and the Playfair square as a key. Refer to your class notes for details on how the Playfair Cipher work.

There are two points to observe:

1- The Encryption function invokes the `_format_playfair(plaintext)` function to apply formatting rules before applying the actual encryption.
2- Encryption is only performed on alphabetical characters, both upper and lower case. Therefore, all non-alpha characters are left unchanged in their respective positions.

The formatting function applies the following formatting rules:

- Every `W` is converted to `VV`, and every `w` is converted to `vv`
- If the plaintext length (excluding non-alpha) is odd, an `x` is appended.
- Every pair that contains a double character `##` is converted to `#x`. The case of the character is preserved[1].

Below are some examples for testing the formatting function:

```
Light                      Lightx
lesson                     lesxon
door                       door
floor                      floxrx
window                     vxindovx
```

[1] There is a very rare scenario that could arise if there is a double xx in the word. There are no English words that has double x. However, this could result from scenarios like: Ox xaphoon. We can ignore such scenarios because of their negligible probability. If the plaintext contain a single word like: fox → foxx. This can be easily detected because fox is an English word but foxx is not. For the unfortunate case like: apex, the word ape is also an English word. In this case, the function should pick "ape" not "apex".

`Widow`	`VXidovvx`
`Are you happy?`	`Are you hapxy?x`
`What?! Angry!!`	`VXhat?! Angry!!`
`Wow!! That is wonderful!`	`VXovv!! That is vvonderful!x`

Decryption Process:

Decryption works in the reverse process of encryption, while preserving the case of characters and the position of non-alpha characters. The decryption itself is not difficult, but restoring the original plaintext has some challenges. For instance, how do we know whether the x in the plaintext is an actual x or one that has been a result of a double character?

In order to restore the original text, you need to create two functions:

1- `_restore_playfair(text)`: reformats a decrypted text back to its original English form
2- `_restore_word_playfair(word,dict_list)`: focuses on restoring a single word back to English. This function is invoked by `_restore_playfair`

The `_restore_playfair` function performs the following main tasks:

1- Converts every `VV/vv` back to `W/w`. Since there are no English words that has double `vv`, we know for sure that a `vv` points to a `w`[2].
2- Check every word that contains an `x` character whether it is an actual English word, or should be restored to a double character. This is done through the `_restore_word_playfair` function.
3- Check whether the last word contains an extra `'x'` resulting from the padding in the encryption process. For instance, `Lightx` is clearly `Light`. But what about `coax`? In this case, the word `coa` is not an English word, while `coax` is an English word.[3]
4- Watch out for scenarios similar to: `'out there'` which is interpreted as: `'ou tx he re'` (spaces added for clarification).

The `_restore_word_playfair` function focuses of detecting an `x` that is a result of a double character. It operators on a single word, not an entire text. Assume the following:

1- The word does not have more than 2 occurrences of the character x[4].

[2] You do not have to worry about scenarios like: "listserv video"
[3] See the previous footnote that addresses the scenario of "apex".
[4] This addresses scenarios like: 'anxious' (one original x), lesxon (one double char), afxix (one original and one double) and setxex (two double chars) and execution (two original x).

2- The word is either lower case, upper case or capitalized (i.e. first character is upper and rest are lower).

It is good to note that you can detect whether a word is English or not English by running the function: `is_plaintext(word,word_list,1)`. Using a threshold of 1 is similar to asking whether the word is in the dictionary (loaded in `word_list`).

Below are sample testing for two restore functions:

```
anxious  --> anxious
doxr     --> door
exl      --> eel
afxix    --> affix
EXCESX --> EXCESS
foxtbalx --> football
Lesxon Lightx   →    Lesson Light
vxindovx floxrx →    window floor
VXidovv floxr   →    Widow floor
```

Completing this task can take extensive time if poorly design. Make sure to conduct proper modular testing for each function.

Task 4: Columnar Transposition Cipher (2.4 pts)

The objective of this task is to write the encryption and decryption schemes for the columnar transposition. This requires writing three functions:

1- Utility function: `_get_order_ct(key)`
2- Encryption function: `e_ct(plaintext, key)`
3- Decryption function: `d_ct(ciphertext,key)`

The ***key*** is any English string containing ASCII characters from the space character (ASCII = 32) to tilde character ~ (ASCII = 126). The utility function `_get_order_ct(key)` performs two main tasks. The first is to check if the key is a valid input. If it is not valid (e.g. not a string or an empty string) it returns [ ].

For the order of characters use the stream in `get_base('all')` in addition to the space character.

The second objective is to return the order of characters in the key. This is represented as a list in which its elements represent the expected order.

For instance, if key = `cave` would return [1,0,3,2]. This is explained as:

- since `'a'` is the smallest character, then it should be placed in the first column → [1]
- Second smallest character in key is `'c'`, which should be put in the second column. Since the position of `'c'` is 0 → [1,0].
- The third character is `'e'` which is in position 3. This should be put in the third column → [1,0,3]
- The last character is `'v'` which is in position 2 → [1,0,3,2]

Below are some test samples:

```
Key order for  = []
Key order for r = [0]
Key order for ? = [0]
Key order for RAINY? = [1, 2, 3, 0, 4, 5]
Key order for dad = [1, 0]
Key order for face = [1, 2, 3, 0]
Key order for Face = [0, 1, 2, 3]
Key order for apple = [0, 3, 2, 1]
Key order for good day = [3, 4, 2, 0, 1, 5]
Key order for German = [0, 4, 1, 3, 5, 2]
```

The encryption and decryption functions are applied to all characters in the stream, except the whitespaces (space, tab and newline characters).

Whenever necessary padding with the default PAD character should be used by the encryption function. The decryption should get rid of any existing padding.

Note: There are various implementations of columnar transposition available online. If you use any as a reference, then you need to provide the reference. However, none of those implementations satisfy the requirements of this task. I strongly recommend to try it yourself, because that will empower you to understand how the cryptanalysis would work if you are requested to design a brute-force scheme.