Encryption and Decryption in Python Project

Create the Constructor

In this project, you are tasked to implement a Python class (call it `TranspositionCipher`) that can encrypt and decrypt a message using the transposition cipher–a simple yet effective way of encrypting a text in a way that becomes unreadable to anyone who doesn't possess the key to decryption. It relies on scrambling the words in plaintext by rearranging its characters according to a specific algorithm.

The project requires you to implement a `TranspositionCipher` class incorporating the following elements:

- A constructor function that accepts the cipher's key as an argument
- A method designated for encrypting a message requiring a single parameter–the plaintext message to be encrypted
- A method dedicated to decrypting a message that calls for one argument–the previously encrypted message in ciphertext format
- **Optional:** *As an additional challenge–not required for completing the project–you can implement a function outside the `TranspositionCipher` class, which hacks the columnar transposition cipher, i.e., it decrypts a ciphertext without knowing the key. The function should return the decrypted message and the key.*

Consider the following code skeleton included in the downloadable Transposition Cipher.ipynb file:

```
class TranspositionCipher(object):

    def __init__(self, key):
        pass

    def encrypt_message(self, message):
        pass

    def decrypt_message(self, message):
        pass
```

You might've noticed the `pass` keyword. `pass` is a unique statement in Python that does nothing. It can be used as a placeholder for future code. Nothing happens when the `pass` statement is executed, but you avoid getting an error when empty code is not allowed. In the context of this project, it's used in the class skeleton to indicate where you need to add your method implementations.

Your first task is implementing the constructor for the `TranspositionCipher` class in Python. The constructor will be responsible for initializing new class instances and should take a single argument: the cipher key. The keyword `self` refers to the instance that is being manipulated.

To complete the task, execute the following steps:

Encrypt a Message

Now that your constructor is implemented and the key is known, it's time to implement the functionality to encrypt a message within the `encrypt_message()` method. The method should take `self` and `message` as parameters, with `message` being the plaintext message to be encrypted in string format. The method should return the encrypted message as a string.

Encryption using the columnar transposition cipher is done as follows:

I. **Select a Message:** Start the encryption by determining the message you intend to encrypt.
Example: Let our secret message be
Learning Python is fun

II. **Select a Key:** Select a key, which can be any positive integer value.
Example: Let's choose six.

III. **Construct the Grid:** Start creating the grid by forming a row of cells equal in number to your selected key.
Example: Create a row with six cells.

IV. **Populate Initial Characters:** Fill in the initial characters of your message into the grid, ensuring each cell contains a single character.
Example: We can fit the first six characters in the grid.

L	e	a

V. **Complete the Grid:** Continually add rows having the same length as the original one. Populate them with the subsequent characters from your message until you have exhausted all characters. Remember that space also counts as a character.
Example: The grid that would fit all characters in the text has four rows and six columns.

L	e	a
n	g	
h	o	n
	f	u

VI. **Mark Unused Cells:** Disregard any remaining unpopulated cells within the grid.
Example: We need to ignore the final two cells of the last row because no characters are placed inside.

L	e	a

n	g	
h	o	n
	f	u

VII. **Encrypt the Message:** Perform the encryption by reading the populated grid column by column, beginning from the top-left corner. Ensure you exclude the unused cells.
Example: The resulting encrypted message reads
Lnh egofa nurP nnyiits

You can follow this guided version to realize one possible implementation of the encryption method using the columnar transposition cipher.

1. **Split the message into a list of characters:** The first step to encrypting a message is to split it into individual characters and optionally convert the characters to lower- or uppercase for an extra layer of security. This will create a list where each element is a single message character.
2. **Calculate the message's length:** Get the message's size and store the number in a variable called `message_length`. This will be used in the upcoming steps to ensure we stay within the list's boundaries when performing the transposition.
3. **Initialize an empty string for the encrypted message:** Set up an empty string to hold the encrypted message. As you work through the transposition, you'll add characters to this string in their new (scrambled) order.
4. **Notice the pattern in indices:** Let's use the example from the case description, where we manually encrypted the message '*Learning Python is fun*' using integer six as a key.

L	e	a
n	g	
h	o	n
	f	u

Notice the following indexing pattern:

L 0+0×6=0	e 1+0×6=1	a 2+0×6=2
n 0+1×6=6	g 1+1×6=7	 2+1×6=8
h 0+2×6=12	o 1+2×6=13	n 2+2×6=14
 0+3×6=18	f 1+3×6=19	u 2+3×6=20

The first addend in each calculation represents the column number which ranges from 0 to the key number minus 1, inclusive. The second addend is a product of two multipliers.

-

- The first one represents the row number and ranges from 0 to the ceiling of the division of message length by the key:

```
ceil(message_length/self.key)
```

The ceiling function–imported from the `math` Python library–rounds up the result of dividing two numbers. We can see how this is useful if we consider that to fill 22 characters (the length of the message) in rows of 6 cells each, we need more than three rows but not four full ones. The ceiling function retrieves the required number of rows for us.

- The second multiplier is the key itself.

Now, think about how to construct the encrypted message using nested for-loops and the list built in Step 1. Pay attention to the shaded cells, which are not part of the encryption process.

5. **Construct nested for-loops to perform the transposition:**
   a. Set up a loop to iterate through each column (from 0 to the key minus 1) and a nested loop to iterate through each row (from 0 to the calculated ceiling value minus 1).

```
for j in range(# set the range as key size #):

    for i in range(# set the range as the
                   # ceiling value of message
                   # length divided by key size #):
```

1.
   b. Within the inner loop, calculate the index as suggested in the previous step.

```
for j in range(# set the range as key size #):

    for i in range(# set the range as the
                   # ceiling value of message
                   # length divided by key size #):

        index = # Calculate the index in the original
                # message that corresponds to the
                # current position in the cipher grid
```

1.

c. Ensure that the calculated index is strictly smaller than the message length–i.e., ignore the shaded cells. If it is, add the character to the encrypted message string.

```
for j in range(# set the range as key size #):

    for i in range(# set the range as the
                   # ceiling value of message
                   # length divided by key size #):

        index = # Calculate the index in the original
                # message that corresponds to the
                # current position in the cipher grid

        if index < message_length:
            # Append the corresponding character from
            # the original message to the encrypted
            # message string
```

6. **Return the encrypted message:** Return the string storing the encrypted message.
7. **Test the code:** Create an instance of the `TranspositionCipher` class and set the `key` parameter to `6`. Encrypt the message *Learning Python is fun* and ensure you obtain the exact string as the one received by hand in the case description of the project.

Decrypt a Message

Great job so far! You've successfully encrypted your secret message.

Now, to share a secret communication with your friends, they need to be able to decrypt the message you've sent them. Therefore, your final task is to construct a decryption method that takes only the encrypted message as a parameter, which should return the decrypted message as a string.

A message's decryption works like the encryption process but inverted.

I. **Select a message to decrypt:** Select an encrypted message for which you are given the key.
Example: Let's use the following encrypted message:
Lnh egofa nurP nnyiits
We know the key for decrypting it is 6.

II. **Construct the grid:** Start creating the decryption grid by forming a row of cells equal in number to the ceiling of the ratio between the length of the message and the key.
Example: Create a row with `ceil(22/6) = 4` cells.

III. **Populate initial characters:** Fill in the initial characters of your encrypted message into the grid, ensuring each cell contains a single character.
Example: We can fit the first four characters in the grid.

L	n

IV. **Complete the grid:** Continually add rows having the same length as the original one. Populate them with the subsequent characters from your message until you have exhausted all characters. Remember that space also counts as a character. And this time, the two unused cells are the final ones from the last column rather than the last row.
Example: The grid that would fit all characters in the text has six rows and four columns.

L	n
e	g
a	
r	P
n	y
i	t

V. **Encrypt the message:** Perform the decryption by reading the populated grid column by column, beginning from the top-left corner. Ensure you exclude the unused cells.
Example: The resulting encrypted message reads:
Learning Python is fun

This guided version suggests one of many ways to implement the decryption method using the columnar transposition cipher.

1. **Split the message into a list of characters:** The first step to decrypting a message is to split it into individual characters, creating a list where each element is a single message character.

```
def decrypt_message(self, message):
    # Store each character in a list
    message_split =
```

2. **Calculate the length of the message:** Next, find the length of the message used to calculate the grid size and ensure we stay within bounds.

```
    # Compute the total length of the message
    message_length =
```

3. **Calculate the number of columns required:** The number of columns in the grid is calculated by finding the ceiling ratio between the message length and the key.

```
    # Calculate the ceiling value of the ratio
    # of the message length to the key size
```

```
message_ceil =
```

4. **Calculate the number of empty cells in the grid:** Find the number of cells that will store no characters and be ignored in the algorithm. To do that:
    a. Multiply the key (number of rows) by the ceiling found in the previous step (number of columns). This calculation will give the total number of cells in the grid.
    b. Subtract the encrypted message length from the product found in part 4a. The resulting number represents the number of unpopulated cells in the grid.

```
# Calculate the number of empty cells in the
# decryption grid by subtracting the actual
# message length from the total size of the grid
num_empty_cells =
```

5. **Initialize a grid of empty strings:** Create an empty grid to hold the characters of the encrypted message. The grid will have as many rows as the key and columns as the calculated ceiling. Each cell in the grid will store the empty string `''`. This usefulness will become more apparent in the upcoming steps–upon concatenating a string with an empty string, we retrieve the former string, i.e., no change is made to the original string.

```
# Initialize a 2D grid with empty strings,
# which will be filled with the message
# characters for decryption
message_grid =
```

6. **Initialize an empty string for the decrypted message:** Set up an empty string to hold the decrypted message. As you work through the transposition, you'll add characters to this string in their new (correct) order.

```
message_decrypted = ''
```

7. **Create an iterator:** Research how iterator objects in Python are initialized and used: https://docs.python.org/3/library/functions.html#iter
Declare an iterator object from the list of characters created in Step 1, allowing us to get the next character from the message each time we fill a cell in the grid.

```
iterator = iter(message_split)
```

8. **Construct nested for-loops for filling in the grid:** Set up a nested loop to iterate through each cell in the grid (each column within each row). In each iteration, fill in the current cell with the next character from the message unless you're in a cell that should be ignored. Once the process is finished, the ignored cells will store an empty string.
9. **Construct nested for-loops for decrypting the message:** Set up another nested loop to read the characters from the grid in their original order–down each column, then across to the next column. Add each character to the decrypted message string. In this process, we'll also add the ignored cells. But that shouldn't concern us because they're empty strings and won't affect the decrypted message.
10. **Return the decrypted message:** Finally, return the decrypted message.