# TDR-SDC TASK 1

# Problem Set 3

**Handed out:** Thursday, January 27th, 2022.

**Due: Thursday, February 3rd, 2022 at 06:15pm**

This problem set has two parts. The first part allows you to practice thinking about problems in a recursive fashion, taking advantage of the idea that one can reduce the problem to a simpler version of the same problem. In `ps4a.py`, you will write a recursive function that takes as input a string and figures out all the possible reorderings of the characters in the string.

The second part will give you experience in thinking about problems in terms of classes, each instance of which contains specific attributes as well as methods for manipulating them. In `ps4b.py`, you will use object-oriented programming to write a Caesar/shift cipher. In `ps4c.py` , you will use object-oriented programming to write a very simple substitution cipher.

As always, please do not rename the files we provide you with, change any of the provided helper functions, change function/method names, or delete provided docstrings. You will need to keep `words.txt` and `story.txt` in the same folder in which you store `ps4a.py`, `ps4b.py` and `ps4c.py`.

Finally, please consult the Style Guide on as we will be taking point deductions for violations (e.g. non-descriptive variable names and uncommented code). For this pset style guide numbers 6, 7 and 8 will be highly relevant so make sure you go over those before starting the pset, and again before you hand it in!

## Part A: Permutations of a string

A *permutation* is simply a name for a reordering. So the permutations of the string 'abc' are 'abc', 'acb', 'bac', 'bca', 'cab', and 'cba'. Note that a sequence is a permutation of itself (the trivial permutation). For this part of the pset you'll need to write a **recursive** function `get_permutations` that takes a string and returns a list of all its permutations. You will find this function helpful later in the pset for part C.

A couple of notes on the requirements: **Recursion MUST be used**, global variables may NOT be used. Additionally, it is okay to use loops to code the solution. The order of the returned permutations does not matter. Please also avoid returning duplicates in your final list.

**Suggested Approach**

In order to solve any recursive problem, we must have at least one base case and a recursive case (or cases). We can think of our base case as the simplest input we could have to this problem (for which determining the solution is trivial and requires no recursion) -- for this approach, our base case is if `sequence` is a single character (there's only one way to order a single character).

If `sequence` is longer than one character, we need to identify a simpler version of the problem that, if solved, will help us easily find all permutations for `sequence`. The pseudocode below gives one approach to recursively solving this problem.

Given an input string `sequence`:

- Base case:
  - if `sequence` is a single character, there's only one way to order it
    - return a singleton list containing `sequence`
- Recursive case:
  - suppose we have a method that can give us a list of all permutations of <u>all but the first character</u> in `sequence` (Hint: think recursion)
  - then the permutations of all characters in `sequence` would be **all the different ways** we can insert the first character into each permutation of the remaining characters
    - *example*: if our word was 'bust', we hold out the character 'b' and get the list ['ust', 'sut', 'stu', 'uts', 'tus', 'tsu']
      - then 'ust' gives us: '**b**ust', 'u**b**st', 'us**b**t', 'ust**b**'
      - 'sut' gives us: '**b**sut', 's**b**ut', 'su**b**t', 'sut**b**'
      - and so on …

Implement the `get_permutations(sequence)` function found in `ps4a.py` according to the specifications in the docstring. Write three test cases for your function in comments under `if__name__== '__main__'`. Each test case should display the input, expected output, and actual output. See the `if__name__== '__main__'` for an example.

# Part B: Cipher Like Caesar

Ever want to relive the glory days of your youth and pass secret messages to your friends? Well, here is your chance! But first, here is some vocabulary:

- Encryption - the process of obscuring or encoding messages to make them unreadable
- Decryption - making encrypted messages readable again by decoding them
- Cipher - algorithm for performing encryption and decryption
- Plaintext - the original message
- Ciphertext - the encrypted message. Note: a ciphertext still contains all of the original message information, even if it looks like gibberish.

### Caesar Cipher

The idea of the Caesar Cipher is to pick an integer and shift every letter of your message by that integer. In other words, suppose the shift is *k*. Then, all instances of the $i^{th}$ letter of the alphabet that appear in the plaintext should become the $(i + k)^{th}$ letter of the alphabet in the ciphertext. You will need to be careful with the case in which $i + k > 26$ (the length of the alphabet).

We will treat uppercase and lowercase letters individually, so that uppercase letters are always mapped to an uppercase letter, and lowercase letters are always mapped to a lowercase letter. If an uppercase letter maps to "A", then the same lowercase letter should map to "a". Punctuation and spaces should be retained and not changed. For example, a plaintext message with a comma should have a corresponding ciphertext with a comma in the same position.

### Examples:

| plaintext | shift | ciphertext |
|---|---|---|
| 'abcdef' | 2 | 'cdefgh' |
| 'Hello, World!' | 4 | 'Lipps, Asvph!' |
| '' | any value | '' |

### Classes and Inheritance

This is your first experience coding with classes! Get excited! We will have a `Message` class with two subclasses `PlaintextMessage` and `CiphertextMessage`. `Message` contains methods that could be used to apply a cipher to a string, either to

encrypt or to decrypt a message (since for Caesar codes this is the same action). `PlaintextMessage` has methods to encode a string using a specified shift value; our class will always create an encoded version of the message, and will have methods for changing the encoding. `CiphertextMessage` contains a method used to decode a string.

When you have completed your implementation, you can either create a `CiphertextMessage` instance using an encrypted string that someone provides you and try to decrypt it; or you can encrypt your own `PlaintextMessage` instance, then create a `CiphertextMessage` instance from the encrypted message within the `PlaintextMessage` instance, and try to decrypt it and see if it matches the original plaintext message.

**Your job will be to fill methods for all three of these classes according to the specifications given in the docstrings of `ps4b.py`.** Please remember that you never want to directly access attributes outside a class - that's why you have getter and setter methods. Don't overthink this; a getter method should just return an attribute and a set method should just set an attribute equal to the argument passed in. Although they seem simple, we need these methods in order to make sure that we are not manipulating attributes we shouldn't be. Directly using class attributes outside of the class itself instead of using getters and setters will result in a point deduction – and more importantly can cause you headaches as you design and implement object class hierarchies.

**Rules**
You do not have to use recursion in Part B, but you are welcome to. There are a couple of helper functions that we have implemented for you: `load_words` and `is_word` . You may use these in your solution and you do not need to understand them completely, but should read the associated comments. You should read and understand the helper code in the rest of the file and use it to guide your solution.

**Part 1: Message**

> Fill in the methods of the `Message` class found in `ps4b.py` according to the specifications in the docstrings.

We have provided skeleton code in the `Message` class for the following functions - your task is to implement them. Please see the docstring comment with each function for more information about the function specification.

- `__init__(self, text)`
- The getter methods

    - `get_message_text(self)`
        - Note: This should return an immutable version of the message text we added to this object in init. Luckily, strings are already immutable objects, so we can simply return that string.
    - `get_valid_words(self)`
        - Note: this should return a COPY of self.valid_words to prevent someone from accidentally mutating the original list.
- `build_shift_dict(self, shift)`
    - Note: you may find the `string` module's `ascii_lowercase` constant helpful here.
- `apply_shift(self, shift)`
    - Hint: use `build_shift_dict(self, shift)`. Remember that spaces and punctuation should not be changed by the cipher.

**Part 2: PlaintextMessage**

Fill in the methods of the `PlaintextMessage` class found in `ps4b.py` according to the specifications in the docstrings.

The methods you should fill in are:

- `__init__(self, text, shift)`
    - Use the parent class constructor to make your code more concise. Take a look at Style Guide #7 if you are confused.
- The getter methods
    - `get_shift(self)`
    - `get_encryption_dict(self)`
        - Note: this should return a COPY of self.encryption_dict to prevent someone from mutating the original dictionary.
    - `get_message_text_encrypted(self)`
- `change_shift(self, shift)`
    - Hint: think about what other methods you can use to make this easier. It shouldn't take more than a couple lines of code

**Part 3: CiphertextMessage**

Don't you want to know what your friends are saying? Given an encrypted message, if you know the shift used to encode the message, decoding it is trivial. If `message` is the encrypted message, and `s` is the shift used to encrypt the message, then `apply_shift(message, 26-s)` gives you the original plaintext message. Do you see why?

The problem, of course, is that you don't know the shift. But our encryption

method only has 26 distinct possible values for the shift! We know English is the main language of these emails, so if we can write a program that tries each shift and maximizes the number of English words in the decoded message, we can decrypt their cipher!

> Fill in the methods of the `CiphertextMessage` class found in `ps4b.py` according to the specifications in the docstrings.

The methods you should fill in are:

- `__init__(self, text)`
  - Hint: use the parent class constructor to make your code more concise. Take a look at Style Guide #7 if you are confused.
- `decrypt_message(self)`
  - Hint: you may find the helper function `is_word(wordlist, word)` and the string method `split` useful (https://docs.python.org/3/library/stdtypes.html#str.split, or alternatively `help(str.split)` in your console)
  - Note: `is_word` will ignore punctuation and other special characters when considering whether a word is valid.

**Part 4: Testing**

> Write two test cases for `PlaintextMessage` and two test cases for `CiphertextMessage` in comments under `if__name__== '__main__'` . Each test case should display the input, expected output, and actual output. An example can be found in `ps4b.py` . Then, decode the file story.txt and write the best shift value used to decrypt the story, and unencrypted story in a comment underneath your test cases.

Hint: The skeleton code contains a helper function `get_story_string` that returns the encrypted version of the story as a string. Create a `CiphertextMessage` object using the story string and use `decrypt_message` to return the appropriate shift value and unencrypted story.

## Part C: Substitution Cipher

A better way to hide your messages is to use a substitution cipher. In this approach, you create a hidden coding scheme, in which you substitute a randomly selected letter for each original letter. For the letter "a", you could pick any of the other 26 letters (including keeping "a"), for the letter "b", you could then pick any of the remaining 25 letters (other than what you selected for "a") and so on. You can probably see that the number of possibilities to test if you wanted to decrypt a substitution ciphered message is much larger than for a Caesar cipher (26! compared to 26). So for this problem, we are going to just consider substitution ciphers in which the vowels are encrypted, with lowercase and uppercase versions of a letter being mapped to corresponding letters. (For example, 'A' -> 'O' then 'a'->'o').

### Classes and Inheritance

Similar to the Caesar cipher, we are going to use classes to explore this idea. We will have a `SubMessage` class with general functions for handling Substitution Messages of this kind. We will also write a class with a more specific implementation and specification, `EncryptedSubMessage`, that inherits from the `SubMessage` class.

Your job will be to fill methods for both classes according to the specifications given in the docstrings of `ps4c.py`. Please remember that you never want to directly access attributes outside a class - that's why you have getter and setter methods. Again, don't overthink this; a get method should just return a variable and a set method should just set an attribute equal to the parameter passed in. Although they are simple, we need these methods in order to make sure that we are not manipulating attributes we shouldn't be. Directly using class attributes outside of the class itself instead of using getters and setters will result in a point deduction – and more importantly can cause you headaches as you design and implement object class hierarchies.

### Part 1: SubMessage

> Fill in the methods of the `SubMessage` class found in `ps4c.py` according to the specifications in the docstrings.

We have provided skeleton code for the following methods in the `SubMessage` class - your task is to implement them. Please see the docstring comment with each function for more information about the function specification.

- `__init__(self, text)`
- The getter methods
    - `get_message_text(self)`
    - `get_valid_words(self)`
        - Note: this should return a COPY of self.valid_words to prevent someone from mutating the original list.
- `build_transpose_dict(self, vowels_permutation)`
- `apply_transpose(self, transpose_dict)`
        - Note: Pay close attention to the input specification when testing.

**Part 2: EncryptedSubMessage**

Fill in the methods of the `EncryptedSubMessage` class found in `ps4c.py` according to the specifications in the docstrings.

Don't you want to know what your friends are saying? Given an encrypted message, if you know the substitution used to encode the message, decoding it is trivial. You could just replace each letter with the correct, decoded letter.

The problem, of course, is that you don't know the substitution. Even if we keep all the consonants the same, we still have a lot of options for trying different substitutions for the vowels. As with the Caesar cipher, we can use the trick of testing (giving a proposed substitution) to see if the decoded words are real English words. We then keep the decryption that yields the most valid English words. Note that we have provided constants containing the values of the uppercase vowels, lowercase vowels, uppercase consonants, and lowercase consonants, for your convenience.

In this part, you can use your `get_permutations` method from part A (it is already imported for you in the beginning of ps4c.py).

The methods you should fill in are:

- `__init__(self, text)`
    - Hint: use the parent class constructor to make your code more concise. Take a look at Style Guide #7 if you are confused.
- `decrypt_message(self)`

**Part 3: Testing**

Write two test cases for `SubMessage` and two test cases for

`EncryptedSubMessage` in comments under `if___name__== '__main__'` .
Each test case should contain the input, expected output, function call used, and actual output.