Python Fundamentals

Final Coding Exam

This assignment consists of 3 parts. At most you can score 20 points. There is also a bonus of 5 Extra Credits in Part 4 that can be earned.

You can receive partial credit for partial solutions.

Submit your solutions to all parts as individual notebooks (.ipynb file) and a PDF to Brightspace.

Note that you should print out and display output under cells when you have completed steps in all the parts (not only the final output of a part).

For part 3, please do not rename the files we provide you with, or any of the provided helper functions, or function/method names, or delete provided docstrings. You will need to keep `words.txt` and `story.txt` and `permutations.py`.

Note: full disclosure, questions 1 and 3 have been adapted from other sources and one may find solutions to these online, but I urge you not to.

1. **Recursion traces (6 points)** - write a function called `trace()` that takes as its input any recursive function and outputs a *traced* version of the same i.e., shows schematically, all the *parameters passed*, and *values returned* at each level. The levels should follow the indentation scheme as shown in the examples below. You cannot use the built-in `@trace` method or global variables.

   You can use this recursion visualizer to aid with understanding of the recursive process of a function.

def factorial(n):
    if n == 1:
        return 1
    else:
        return n * factorial(n-1)
	|-- factorial(7)
	| |-- factorial(6)
	| | |-- factorial(5)
	| | | |-- factorial(4)
	| | | | |-- factorial(3)
	| | | | | |-- factorial(2)
	| | | | | | |-- factorial(1)
	| | | | | | | |-- return 1
	| | | | | | |-- return 2
	| | | | | |-- return 6
	| | | | |-- return 24
	| | | |-- return 120
	| | |-- return 720
	| |-- return 5040
	5040

def fibonacci(n):
    if n == 0 or n == 1:
        return n
    else:
        return fibonacci(n - 1) +
fibonacci(n - 2)
	|-- fibonacci(3)
	| |-- fibonacci(2)
	| | |-- fibonacci(1)
	| | | |-- return 1
	| | |-- fibonacci(0)
	| | | |-- return 0
	| | |-- return 1
	| |-- fibonacci(1)
	| | |-- return 1
	| |-- return 2
	2

def accumufact(n, fact=1):
    if n == 1:
        return fact
    else:
        return accumufact(n-1, n*fact)
	|-- accumufact(5)
	| |-- accumufact(4, 5)
	| | |-- accumufact(3, 20)
	| | | |-- accumufact(2, 60)
	| | | | |-- accumufact(1, 120)
	| | | | | |-- return 120
	| | | | |-- return 120
	| | | |-- return 120
	| | |-- return 120
	| |-- return 120
	120

def gcd(p, q):
    if q == 0:
        return p
    else:
        return gcd(q, p%q)
	|-- gcd(165, 27)
	| |-- gcd(27, 3)
	| | |-- gcd(3, 0)
	| | | |-- return 3
	| | |-- return 3
	| |-- return 3
	3

2. **IMDb movie rating analysis (6 points)** - find out how to use the Cinemagoer library (you will have to run "`!pip install cinemagoer`" in your notebook to access the library) in Python through library documentation and perform the following tasks:
    a. Find the highest rated movie.
    b. Find the genre that is highest rated out of the top 250 rated movies.
    c. Find the production company with the greatest number of movies rated above 8.
    d. Find the director of the highest rated movie and list their top 10 rated movies.

e. Create a small quiz program where the user can enter either the movie title or movie ID and get 2-3 questions about the movie as a quiz.

3. **Encrypt it! (8 points)** - A good 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 very large (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. ('A' -> 'O' then 'a'->'o').

Classes and Inheritance

You are going to use classes to explore this idea. You will have a `SubMessage` class with general functions for handling Substitution Messages of this kind. You 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 `ps3.ipynb`. 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 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 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 `ps3.ipynb` according to the specifications in the docstrings.

You are 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 `ps3.ipynb` 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 you keep all the consonants the same, you still have a lot of options for trying different substitutions for the vowels. As with the Caesar cipher, you can use the trick of testing (giving a proposed substitution) to see if the decoded words are real English words. You then keep the decryption that yields the most valid English words. Note that you are provided with constants containing the values of the uppercase vowels, lowercase vowels, uppercase consonants, and lowercase consonants, for your convenience. In this part, you can use the `get_permutations` method from the `permutations.py` file provided to you (it is already imported for you in the beginning of `ps3.ipynb`). The methods you should fill in are:

- `__init__(self, text)`
  - **Hint**: use the parent class constructor to make your code more concise.
- `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.

4. **BONUS (5 points)**: recreate the animation shown here. This can have varying levels of complexity which will also map to how you will be graded:
    a. Basic: create only the grid system and points that can move on the grid with each iteration/frame change and present the frames individually.
    b. Intermediate: based on proximity of point (decide a threshold) connections forming or breaking between any two points.
    c. Advanced: exact recreation with the animation across frames.