EMAT10007: Introduction to Computer Programming Assignment

Leonardo Coppi

December 2021

1 Introduction

Contained within the `code` folder and run by the `main.py` program is an interactive Python program that allows the user to input a message or file to be encrypted/(auto)decrypted using a Caesar cipher, leaving non-alphabetic characters unchanged.

The rotation can be manually entered or randomly assigned. Additionally, the given message can be brute-force decrypted. All code sections are stored in separate modules, and all employ input sanitization to ensure code re-usability; encryption & decryption functions also tolerate all-case input.

The program then extracts statistics on the decrypted message, sanitizing words by removing non-alphabetic characters. Total number of words, unique number of words, shortest word, longest word, and most common letter are printed and saved to `metrics.txt`. A bar chart is also generated from the frequency of the most common words and saved as `bar_chart.py`.

2 Analysis

2.1 Part 1

Part 1.1-1.3's user selection code is contained within the `main.py` file.

To comply with the requirement that (*"If no input is given, or incorrect input is given ... the program should print an error message and prompt the user for the input again."*), the input requests are wrapped in `while True` loops, `break`ing out on correct input. Only the first letter is taken (`[0]`) to allow for both single letter and word inputs to increase the chances of valid input. Additionally, `.upper()` ensures the input is not case-sensitive.

The `random` module is imported to generate a random rotation value, chosen because it is built-in to Python [1], and used to obtain a random integer between 1-26. This limit is due to rotations lower than 0 or above 26 wrapping around.

Part 1.4's Caesar cipher code is contained within the `functions.py` file.

Though Part 1.4 highlights that the message should be returned in upper case, both the `encrypt()` and `decrypt()` functions purposefully tolerate upper-case and lower-case inputs. That being said, compliance with the assignment is maintained through `.upper()`.

The functions convert the character into its ASCII value and use that to determine whether it is a letter or upper case. This is used to set the bounds of the character's value to stay a letter. Rotation is only applied if the character is a letter to reduce code execution. Rotation values below 0 are normalized to a positive value [2], while values over 26 result in a loop removing/adding 26 from the character's value until the new value is back within the acceptable range. The inefficiency of this is addressed in the conclusion.

2.2 Part 2

Part 2's code is contained within the `metrics.py` file.

Part 2 is a class, as object-oriented execution around the initial message makes more sense than having individual stand-alone functions be given the same input manually.

The class's `__init__` function splits the message into individual words and iterates over each word's characters. Using list comprehension, each character is added to a new list if in the alphabet, which is then joined into a string and appended to a clean words list. Though using Python's `re` (regex) module would be much easier and efficient, it has not been covered in class wheres list comprehension and nested loops have.

As many of Python's internal functions as possible were used to calculate the metrics. For example, to obtain unique words, transforming the list into a set will remove duplicates and make membership testing quicker. [3] The rest uses the `.min()` and `.max()` functions to sort the list of all words. The set is not used for this despite containing fewer entries due to its unordered nature.

Since the metrics module is imported by `main.py`, the `metrics.txt` file is created in the same directory as the main program. This is intended.

[1] https://docs.python.org/3/library/random.html

[2] https://stackoverflow.com/questions/6685057/modular-addition-in-python

[3] https://wiki.python.org/moin/PythonSpeed

Part 2.3 and 2.4 have essentially the same output in two different formats. Where 2.3 is a list of the five most common words, 2.4 is a dictionary of the same words in addition to the frequency in which they occur. Consequently, 2.3 can be simplified to the keys of 2.4's dictionary.

Despite Python 3.7 introducing ordered dictionaries, current industry guidance suggests to use the OrderedDict module from collections citing it increases compatibility [4] and has more explicit intent signalling [5]

The function loops over all the words, adding words not already present to a temporary dictionary with a value of one. If the word is already present, the value is incremented. This results in a dictionary containing the frequency of all words in the message.

To ensure only the five most common words are contained, words with the same frequency are removed. Thus the function iterates over the dictionary until the dictionary length is less than six, removing all words with the same frequency (dictionary key). The frequency to remove starts at 1, and with each iteration increases. For example, the first run removes all dictionary entries where the value = one. The next iteration will remove all entries where the value equals two.

2.3 Part 3

For Part 3, the code is added at the start of `main.py`.

Due to the multiple entry points for the `message` variable to be entered, it is initially set to `None`, which can be checked to see if a function has modified it.

The rest of this section is very similar to Part 1 with the `while True` loops and method of taking user input as described above. Compliance with Part 1.4 is again maintained through an `.upper()` call.

2.4 Part 4

Part 4's code is contained within the `automated.py` file.

To follow best practices suggested, opening `words.txt` is wrapped in a `try-except` block. Despite this, the program still requires the word file, and a custom error is `raise`d on failure to load.

Cleaning the input to be decrypted is very similar to Part 2, except only the first ten words are obtained through `[:9]`. The list of words is loaded as a set to improve comparison speed. [6]

The automatic decryption function then iterates over the range of all possible rotations (1-26), decrypting the shortened message, `.split`ting it into a set and comparing it with the set of words given.

Each possible decryption is presented with the shortened message, and on user confirmation the entire decrypted message is returned. Once again, only the first letter of the `.upper()` user input is taken for the reasons above. If the user denies all possible decryptions, or none were found, `False` is returned.

2.5 Part 5

Since the bar charts suggested in Part 5 are generated from the metrics, the code is placed within the `Metrics` class.

`PyPlots` in `MatPlotLib` has been used in class, so the program uses the same module. Since the main program imports `metrics.py`, the pdf is created in the main program's directory, which is the intended functionality.

I also wanted to practice creating multiple modules to be imported into the main program and thus satisfied the second Part 5 request of creating a Caesar cipher Python module.

3 Conclusion

Further attempts at more of Part 5 would have been good with more time. While the request to create a Caesar cipher Python module imported to perform encryption and decryption operations within the main program, the Caesar Cipher functions, while functional, could be made into a class. Additionally, they 'feel' pretty inefficient. Using modular mathematics would simplify and make the code more efficient, however a lack of confidence and time prevented that concept from being properly explored. I look forward to improving this on my own time.

The exercise only asks for uppercase encryption. Despite this, the encryption and decryption functions support upper-case and lower-case functionality. Thus, while extra lines of code have been sacrificed on a function that may never be used, it is always better to code 'complete' functions that can handle all reasonable input that may be given.

In Part 2.1, using regex to remove all non-alphabet characters from words would have been quicker, more efficient, and more of an industry standard. The omission was a conscious choice, and an improved version of the program would employ regex.

[4] https://gandenberger.org/2018/03/10/ordered-dicts-vs-ordereddict/

[5] https://realpython.com/python-ordereddict/

[6] https://wiki.python.org/moin/PythonSpeed