**QUESTION 1: MODULE TESTS**

***1.1. Coding Basics***

Module 1: 100

Module 2: 100

Module 3: 100

Module 4: 100

***1.2. Intermediate***

Module 1: 100

Module 2: 100

Module 3: 100

Module 4: 100

***1.3. Calculation***

Sum: 100 + 100 + 100 + 100 + 100 + 100 + 100 + 100 = 800

800 ÷ 8 = 100

100 x 0.1 = 10

Total: **10** marks

**QUESTION 2: FINAL TEST**

***2.1. Result and Calculation***

Result: 87

Sum: 87 x 0.2 = 17.4

Total (rounded up): **18** marks

**QUESTION 3: PRACTICAL ACTIVITY**

***3.1. Important Note***

For correctly-displayed code (below), please copy it into a text editor with *word-wrap off*, or open the provided Python file (final\_assessment.cipher\_generator\_v2.py) from the zip submission.

***3.2. Program Copied and Pasted***

class Encryption:

"""

This class provides a Caesar based cipher generator.

"""

def \_\_init\_\_(self) -> None:

""""

Initialises the class with a default shift value of 0; aids graceful exit on KeyboardInterrupt.

"""

self.\_\_shift\_value: int = 0

self.\_\_word: str = ''

def get\_user\_input(self) -> None:

""""

This method asks the user for the word or text to be encrypted, and for their birth month which acts as a key.

Input validation is performed; and, provides option to exit gracefully without mandatory input.

"""

self.\_\_word = input("Please enter word(s) or text to be encrypted: ")

while True: # Loop continues until input is valid or user exits gracefully.

try:

bday\_m = int(input("Please enter your birth month (01 to 12): "))

if bday\_m >= 1 and bday\_m <= 12: # Validate input 'before' updating self.\_\_shift\_value to guard against invalid or unwanted encryption on KeyboardInterrupt.

self.\_\_shift\_value = bday\_m

break

except KeyboardInterrupt: # Enables user to exit without mandatory input.

print("Program ended gracefully: Word was not encrypted:")

break

except:

print("Invalid Input - Please only enter a valid birthday month (1 to 12):") # Repeats until valid, or KeyboardInterrupt.

def encrypt(self) -> str:

"""

Encrypts the user's word/text based on the Caesar algorithm and returns the cipher.

Non-alphabetical characters are simply copied to the new cipher string, preserving original position.

Modulus based formula allows wrapping back around to 'A' after 'Z'.

"""

cipher: str = '' # Initialise cipher string.

ascii\_uppercase\_start = ord('A') # ASCII start for uppercase, used in Modulus formula below.

ascii\_lowercase\_start = ord('a') # '' '' lowercase '' ''

for c in self.\_\_word: # Loop through user's word or text.

ascii\_letter = ord(c) # Get current character's ASCII code.

if c.isalpha() == False: # If a character in the user's word/text is not a letter, do not encrypt it; only copy it:

cipher += c

elif c.isupper():

cipher += chr((ascii\_letter + self.\_\_shift\_value - ascii\_uppercase\_start) % 26 + ascii\_uppercase\_start) # After Z, formula allows wrapping back around to A (Padhye et al., 2018).

elif c.islower():

cipher += chr((ascii\_letter + self.\_\_shift\_value - ascii\_lowercase\_start) % 26 + ascii\_lowercase\_start) # '' ''

return cipher

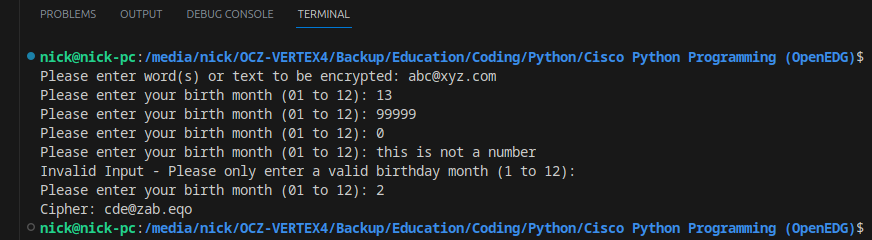
cipher\_generator = Encryption() # Create encryption object

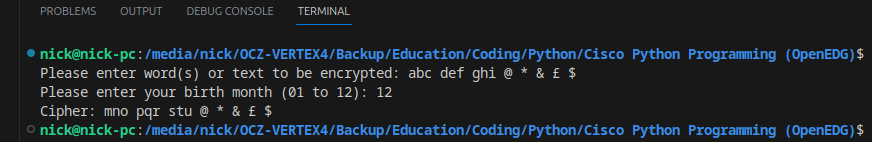
cipher\_generator.get\_user\_input() # Get cipher's user defined paramaters

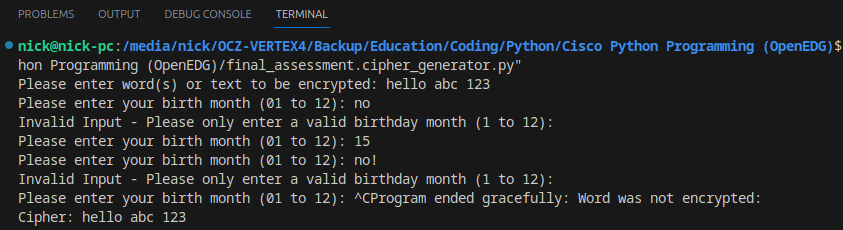
print("Cipher:", cipher\_generator.encrypt()) # Print encrypted output.

***3.3. Program Testing***

Three screenshots providing evidence the program runs flawlessly with complete input validation and exception handling. Figure 1 tests edge cases, out-of-range integers, and non-integers. Figure 2 tests handling of special characters and spaces. In figure 3, the ‘KeyboardInterrupt’ (control+C) is pressed to exit the program gracefully without “killing” the terminal.

Figure 1: Demonstrates handling of out-of-range integer and non-integer input.

Figure 2: Shows correct handling of special characters and spaces which are preserved.

Figure 3: Demonstrates using the keyboard interrupt (control+C) for graceful exit.

**QUESTION 4: ACADEMIC ESSAY**

***4.1. Introduction***

This essay discusses a cipher generator written in Python. It is divided into three subsections, covering design, commands selected, and alignment with current professional practices.

***4.2. Commands and Functionality***

Upon instantiation, the get\_user\_input() method is called, prompting the user for word(s) or text and a shift value (key). Input validation is executed through a try-except block inside an infinite while loop, which only breaks upon valid input between 1 and 12. The program permits graceful exit via KeyboardInterrupt. The encrypt() method is executed when the print function is called:

First, ASCII codes for the alphabets are retrieved using the ord() function. A for-loop then traverses the user's text, collecting each character's ASCII code. An if-elif construct processes each character one at a time. Non-alphabetic characters are directly appended to the cipher string, while alphabetic characters are shifted via the Caesar algorithm.

The modulus operator '%' is crucial for the algorithm, enabling the alphabet to loop from 'Z' back to 'A'. This technique employs modular arithmetic, often overlooked in traditional mathematics but indispensable in computer science (Ringenberg, 2021). The modulus ensures that the remainder will not surpass the alphabet's length of 26. This concept aligns with how 12-hour clocks are grappled with in early childhood: 9 + 4 equals 1, not 13. Using a modulus of 12 clarifies this, as 13 % 12 equals 1 (Ringenberg, 2021). Mathematically, the Caesar encryption can be defined by the formula: 'E\_k(x) = (x + k) mod 26' (Padhye et al., 2018).

***4.3. Design Approach***

The design paradigm chosen was Object-Oriented Programming (OOP), with all functionality being encapsulated in the Encryption class. This modular approach makes the program more easily debugged, maintained, and expanded. For example, we could safely add a new method to decrypt the cipher with a known key, or to battle-test it with brute force. Exception handling for input validation with immediate descriptive feedback (and graceful exit option) improves the user experience. Private variables were chosen for the unencrypted word and the (birthday month) key.

***4.4. Professional Practice Adherence***

“Truly” private variables that can’t be accessed do not exist in Python, but using them as a convention is still preferred practice. Name mangling is more secure than nothing, as it helps avoid accidents (Python Software Foundation, 2023), which aligns with the principle of least privilege, enhancing code security. A positive user experience during program interaction conveys professionalism. The code is also well-commented, with all classes and methods initially introduced with a doc string; further, all important lines are individually commented. Methods like get\_user\_input() and encrypt() are compartmentalised for easier debugging. Testing was also completed (see section 3.3).

Rother (2017) advises adding a condition to ensure the code only runs when the script is executed directly, not automatically on import. Rother also mentions the importance of using descriptive variable names and using type hints to make the code easier to read. These practices were all adopted in the Cipher program. Lastly, the many different operands requiring ASCII conversion are stored as constants before being used in the modulus formula, making the formula more intuitive. However, in conclusion, the code’s compactness could be improved by performing the ASCII conversions inside the formula, instead of creating separate variables.

Word Count *(question 4)*: 539

**References**

Padhye, S., Sahu, R.A. and Saraswat, V. (2018) *Introduction to Cryptography.* 1st edn. Milton: CRC Press. Available at: <https://doi.org/10.1201/9781315114590>.

Python Software Foundation (2023) *9.6 Private Variables*. Available at: [https://docs.python.org/3/tutorial/classes.html#private-variables](https://docs.python.org/3/tutorial/classes.html" \l "private-variables) (Accessed: 7 September 2023).

Ringenberg, J. (2021) *An Investigation of Students’ Understanding of the Modulo Operator.* Master Thesis. University of Nebraska at Omaha. Available at: <https://unomaha.primo.exlibrisgroup.com/permalink/01UON_OMA/1tllovp/cdi_proquest_journals_2584353635> (Accessed: 7 September 2023).

Rother, K. (2017) Pro Python Best Practices: Debugging, Testing and Maintenance. Berkeley, CA: Apress L. P. Available at: <https://doi.org/10.1007/978-1-4842-2241-6>.