COMP1811 – Python Project Report

Your Name:	Tahseen Taj	Student ID	001494074-4
Partner's name:	Samira Ozturk	Student ID	001464034-1

1. BRIEF STATEMENT OF FEATURES YOU HAVE COMPLETED

THIS SECTION SHOULD BE THE SAME FOR ALL GROUP MEMBERS

1.1 Circle the parts of the coursework you have **fully completed and are fully working**. Please be accurate.	**Features** **FA:** 1☐ 2☐ 3☐ 4☐ 5a☐ 5b☐ 5c☐ 6☐ **FB:** 1☒ 2☒ 3☒ 4☒ 5a☒ 5b☒ 5c☒ 6☒
1.2 Circle the parts of the coursework you have **partly completed or are partly working.**	**Features** **FA:** 1☐ 2☐ 3☐ 4☐ 5a☒ 5b☒ 5c☒ 6☐ **FB:** 1☐ 2☐ 3☐ 4☐ 5a☐ 5b☐ 5c☐ 6☐

Briefly explain your answer if you circled any parts in 1.2:

FA.5.a – My partner couldn't encrypt the mail and I have to help her about that.
FA.5.b – It was showing some error in her code which I have to handle to run.

Concise List of Bugs and Weaknesses

A concise list of bugs and/or weaknesses in your work (if you don't think there are any, then say so). Bugs that are declared in this list will lose you fewer marks than ones that you don't declare! (100-200 words, but word count depends heavily on the number of bugs and weaknesses identified.)

THIS SECTION SHOULD BE COMPLETED INDIVIDUALLY FOR FA AND FB AND AS A GROUP FOR FA.7/B.7 AND THE INTERPRETER.

Bugs

List each bug plus a brief description. A bug is code that causes an error or produces unexpected results.

FB.1 – def (show_mail)

1. Attribute Error/ Runtime Error : The function blindly loops through self._mailbox and assumes every item is a Mail object with all the required attributes (frm, to, date, body, etc.). If the mailbox contained an error (like a None object or a simple string), the code would crash with an AttributeError when trying to access mail.frm.

2. Display Flow: I am using fixed-width formatting (e.g., mail.frm:<20}, mail.to:<20}). If the email addresses or the subject lines exceed the allocated space (20 characters, 13 characters, etc.), the table will become misaligned or the text will be abruptly cut off.

F.B.2 –

1. Runtime Error: The code assumes that every item in the self._mailbox is a valid Mail object possessing the required m_id and tag attributes. If the list somehow contains an invalid entry (e.g., a simple integer or a malformed object), the function will crash with an AttributeError when trying to access mail.m_id or mail.tag.

2. Logic Flaw: The success message is generated only inside the loop. If the loop structure were to change or become more complex, the placement of the print statement could become a bug.

F.B.3 –

1.Potential Bug: If the mail variable were accidentally defined outside the function loop (e.g., mail = "Some string"), and then the loop failed, the code might crash or reference the wrong data outside the loop.

2. Critical Bugs: The critical bugs are the Logical Failure where the code sets mail.read = True instead of mail.flag = True when marking an email as flagged, and the Variable Assignment Bug where mail_found == True performs a comparison instead of the necessary assignment (mail_found = True).

3. Encapsulation Violation: The design is flawed because directly accessing mail.read and mail.flag violates encapsulation; it should use public setter methods like mail.set_read_status(True).

4. Variable Assignment Bug: The two critical errors are the Logical Failure where the code incorrectly sets the mail.read = True attribute when attempting to mark the email as flagged, and the Variable Assignment Bug where the statement mail_found == True performs a comparison instead of the necessary assignment (mail_found = True).

F.B.4-

1. Format error: The comparison of date strings (if date == email.date:) is vulnerable to runtime format errors if the input or stored date strings contain unexpected spacing or use incompatible formats (e.g., "Oct

12, 2025" vs. "12/10/2025"). A robust system requires converting the strings to date objects (like Python's datetime) for reliable comparison, though this may use advanced constructs.

F.B.5 –

***1*. Encryption Error: The self.encrypt didn't work or override the self.body part first, but after that I cleared the bracket and it act the body as an encrypt one.**

2. Body Encryption Bugs: Although it can transform the number, but cannot handle tha alphabets, but when the logic correctly worked, it run like the one it is showing in the coursework instructions.

F.B.6 –

1. Calling personal error: When I am trying to import personal in mailboxagent.py, it is not working correctly as there was another personal.py file in the pycache and my mailboxagent.py was calling that one and showing that error. But i cleared all the paths and file named personal.py to include the real one.

2.Format error: When I am trying to add the new mail in a variable it didn't work, but when I choose a variable it worked by then.

Interpreter.py -

1.Attribute error in Mailboxagent: The MailboxAgent class (where all your feature functions are located) contains a method called del_email (Feature A.3). However, in your interpreter.py loop (line 106), you are trying to call a method named del_mail. The MailboxAgent class (where all your feature functions are located) contains a method called del_email (Feature A.3). However, in your interpreter.py loop (line 106), you are trying to call a method named del_mail.

2. Mixing **class attributes** with **object attributes:** The error is: AttributeError: 'MailboxAgent' object has no attribute 'frm'This means that inside your filter method within the MailboxAgent class (line 72), you are trying to access a sender attribute (frm) directly from the MailboxAgent instance (self), but that attribute belongs to the individual Mail objects, not the agent that manages the mailbox.

Inside MailboxAgent.py, line 72, in filter:

if self.frm == frm:

The MailboxAgent class does not have an attribute called self.frm.

3. Unbound Local Error: The error message is: UnboundLocalError: cannot access local variable "input-date" your find function, you have this line:

Python

input_date = input_date.str

You are trying to set the variable input_date equal to a property (.str) of itself. Because input_date had no value defined *before* this line, Python throws an UnboundLocalError.

2.1 WEAKNESSES

List each weakness plus a brief description. A weakness is code that only works under limited scenarios and at some point produces erroneous or unexpected results or code/output that can be improved.

1.Violation of Encapsulation (Structural)

- **Weakness:** All FB functions (show_emails, mv_email, mark, find) directly access the protected mailbox data structure via self._mailbox.
- **Example:** In mv_email, the line for mail in self._mailbox: bypasses the class interface, making the code fragile if the internal name of the mailbox data structure changes.

2. Lack of Type Safety/Defensive Coding (Robustness)

- **Weakness:** Functions like show_emails and mark assume every item iterated over is a valid Mail object and access attributes directly (e.g., mail.m_id).
- **Example:** If the _mailbox contained a non-Mail object (like None), the program would crash with an AttributeError instead of handling the error gracefully and continuing the process.

3. Ambiguous m_id Access (Code Clarity)

- **Weakness:** The ID generation logic is not shown, but functions like mv_email and mark rely on accessing the ID via mail.m_id.
- **Example:** If the Mail class implements the ID as protected (_m_id), direct access violates encapsulation, making the method dependent on internal variable names rather than public accessors.

4. Logical Failure in mark Function (Functional Bug)

- **Weakness:** The code handling the "flagged" mark incorrectly updates the read status attribute (mail.read = True).
- **Example:** An email flagged for follow-up remains visually **unflagged** but will be incorrectly counted as **read** by other system functions.

5. Assignment Bug in mark Function (Coding Error)

- **Weakness:** The check for email being found is written as a comparison: mail_found == True, instead of the assignment mail_found = True.

- **Example:** The mail_found variable is never updated, causing the function to incorrectly execute the final print("Invalid Emaid ID") block and return False even when the email was successfully marked.

6. Flawed Date Handling (FB.4 Runtime Risk)

- **Weakness:** The find function uses an unnecessary and incorrect variable access: input_date = input_date.str, ignoring the required function argument date.
- **Example:** The subsequent comparison logic relies on this error-prone assignment and fails to use the correct date parameter that the function signature demands.

7. Fixed-Width Display Flaw (FB.1 Quality)

- **Weakness:** The show_emails function uses hard-coded fixed-width strings (e.g., f"{mail.frm:<20}|") for pretty printing the table.
- **Example:** If a sender's email address exceeds 20 characters, the table will become **misaligned**, and the data will be poorly presented to the user.

8. Incomplete Error Feedback (FB.4 User Experience)

- **Weakness:** The find function's error message print(f"No Emails found on that date: {date}") incorrectly includes the variable name date.
- **Example:** If the input date was '12/10/2025', the output would be No Emails found on that date: 12/10/2025, which is less professional than using the input_date variable that the code uses for comparison.

9. add email(confidential) :

- This describes the **coding cause**. It implies that your input checking is too relaxed; instead of rejecting invalid tags, it accepts anything and defaults to a generic state, which is poor practice for security-critical features.

3. FEATURES IMPLEMENTED

*Describe your implementation design logic and the choices made (e.g. choice of data structures, custom data types, code logic, choice of functions, etc) and indicate how the features developed were integrated. (**200-400 words**)*

THIS SECTION SHOULD BE COMPLETED INDIVIDUALLY FOR FA AND FB AND AS A GROUP FOR FA.7/B.7 AND THE INTERPRETER

3.1 FEATURE 1

The **show_emails()** method is designed to display stored emails in a clean, structured table-like format that matches the coursework requirements. The underlying design choices were based on usability, readability, and functional simplicity.

First, the mailbox is implemented as an internal list (self._mailbox) which stores multiple **email objects**. This list-based structure was chosen because it efficiently supports ordered storage and iteration over multiple emails. Each email is modeled using a **custom Email class**, where attributes such as m_id, frm, to, date, subject, tag, and body are encapsulated inside the object. This object-oriented design enhances code modularity, making future extensions such as editing or deleting emails more manageable.

The function begins with a simple validation check using if not self._mailbox: to handle the edge case where the mailbox is empty. This ensures the user receives a clear message ("No Emails") instead of a blank table.

To maintain a professional table format, **formatted string literals** (f-strings) with alignment specifiers (e.g., <20) are used. These ensure each column maintains consistent width regardless of text length, improving readability. A header row is printed once to label each column, followed by a loop that prints every email record row-by-row.

The integration of this feature aligns with other mailbox operations (like adding or searching emails), utilizing the same shared data structure. Because each Email object stores all required fields, the display function only retrieves and formats them. This ensures data consistency and reduces duplication of logic.

Overall, the method demonstrates a clear separation of concerns: **data storage is handled by objects and lists**, while **presentation is handled by the show_emails() function**—ensuring a scalable and maintainable implementation.

3.2 FEATURE 2

The **mv_email(m_id, tag)** method is responsible for simulating the relocation of an email into another folder by updating its stored tag value. The design follows an object-oriented approach, where each email is represented as a **Mail object**, and the mailbox is maintained using a **list data structure** (self._mailbox). This structure was selected due to its simplicity and efficient iteration over stored items, which is suitable because email retrieval in this system is index-based rather than requiring fast searching or hashing.

The method begins by setting a Boolean flag mail_found to track whether a matching email exists. A **linear search** is then performed across the mailbox list to locate the email whose m_id matches the user's input.

Once located, the system simulates the movement to another folder by updating the email's tag attribute. This approach avoids physically removing or restructuring elements in the list, making the function computationally efficient and reducing potential side effects.

To support usability and debugging, a success message is printed immediately after updating the tag, and the loop is terminated early using break, ensuring performance efficiency by preventing unnecessary iterations.

If no matching email is found, the function provides clear feedback by displaying an error message and returning False. This ensures reliable error handling, prevents silent failures, and supports robust integration with other system features such as filtering emails by folder.

Overall, the function integrates seamlessly with the email application by relying on shared data structures and consistent attribute manipulation. It supports future enhancements, such as additional folder categories or bulk movement operations, without requiring changes to the underlying mailbox architecture.

3.3 FEATURE 3

The **mark(m_id, m_type)** method is designed to update the status of an email—either marking it as *read* or *flagged*. The implementation follows an object-oriented design, where each email is represented as a Mail object stored inside the list-based mailbox (self._mailbox). This structure allows efficient sequential searching and straightforward attribute updates.

The method begins by normalizing the input using m_type.lower() to make the function more user-friendly and avoid errors caused by inconsistent capitalisation. A Boolean variable mail_found is used to track whether the email ID exists in the mailbox.

The system performs a **linear traversal** of the mailbox list, checking each email's m_id. Once the correct email is found, the design logic branches depending on the requested mark type. If the type is "read", the email's read attribute is switched to True to indicate it has been opened. If the type is "flagged", the same attribute is reused (though ideally this should be a separate flag), simulating marking the email for follow-up. Appropriate confirmation messages are printed to support user feedback and debugging.

If an invalid mark type is provided, the method immediately returns an error message, ensuring clear communication and preventing unintended attribute changes. The loop then breaks early after a successful update to avoid unnecessary iteration.

Finally, after the loop, the method checks mail_found to determine whether a matching email was updated, returning True for success or an error message for invalid IDs.

Overall, this design integrates cleanly with the rest of the system by following existing data structures and object patterns. It emphasises error handling, input validation, and maintainability, ensuring that email state changes remain consistent throughout the application.

3.4 FEATURE 4

The **mark(m_id, m_type)** method is designed to update the status of an email by marking it either as *read* or *flagged*. The implementation uses an object-oriented approach where each email is stored as a **Mail object** inside a list-based mailbox (self._mailbox). A list was chosen because it provides straightforward sequential access, which is appropriate for iterating through emails and performing operations such as marking, moving, or deleting.

To ensure robust input handling, the method converts the supplied m_type to lowercase for consistent comparison, avoiding errors caused by user typing variations. A Boolean variable mail_found is used to track whether the specified email ID exists, helping maintain clear and structured logic for success or failure reporting.

The method performs a **linear search** across the mailbox to locate the email with the matching ID. This approach is simple and efficient enough for a small to medium-sized mailbox, and avoids unnecessary complexity such as hashing or indexing. Once the email is found, the function checks the desired mark type. If the type is "read", the email's read attribute is set to True. If the type is "flagged", the method updates the separate attribute flagged, which provides clearer data separation and improves system extensibility compared to reusing a single variable for multiple states.

For each successful update, a confirmation message is printed to provide immediate feedback to the user. The method then sets mail_found = True and terminates the loop early using break, preventing unnecessary iterations.

If the loop completes without a match, the function prints an error message and returns False. This ensures the system handles invalid IDs gracefully and maintains consistent behaviour across all mailbox operations.

Overall, the method integrates cleanly with existing features such as displaying emails, moving emails, and filtering by tag or state. The design prioritises clarity, maintainability, and accurate state representation for each email.

3.5 FEATURE 5

The **Personal** class extends the parent **Mail** class to represent a specialised category of emails that are automatically classified and processed differently from standard messages. The choice to use **inheritance** allows Personal emails to retain all base email attributes—such as sender, receiver, date, subject, and status flags—while enabling additional customised behaviour without redundancy. By calling super().__init__(), the class ensures that all core Mail properties are initialised consistently.

A key design choice is the automatic assignment of the tag "Personal" inside the constructor. This removes the need for external classification logic and guarantees that every Personal email is stored in the correct simulated folder. The constructor also modifies the body by calling add_stats(), integrating custom behaviour directly at the point of object creation.

The **add_stats()** method encapsulates all transformations and analytics applied to the email body, keeping the class modular and maintaining separation of concerns. The function first extracts the sender's UID (text before the "@"), which replaces the original email body to mimic privacy protection in personal communication. This design choice aligns with the specification's requirement for customised handling of private emails.

In addition to the replacement logic, the method performs straightforward text analysis using Python's built-in string and list operations. A list was selected to store split words because it simplifies word counting, length calculations, and prevents errors such as division by zero. The computed statistics—word count, average word length, and longest word length—are formatted into a single string and concatenated with the modified body, ensuring all Personal email metadata remains in one place.

The show_mail() method outputs the email in a structured, human-readable format consistent with coursework conventions. This allows Personal emails to integrate seamlessly into the wider Outlook simulator system, including viewing, filtering, and folder-based operations.

3.6 FEATURE 6

The **add_email()** method is designed to simulate receiving new emails by dynamically creating and storing email objects of different types. The central design choice is the use of **object-oriented polymorphism**, allowing the system to instantiate different classes—**Mail**, **Confidential**, or **Personal**—based entirely on the tag provided. This approach enables specialised behaviour for different email categories without duplicating common logic.

To ensure every email receives a unique identifier, the method uses an incrementing counter stored in self.next_m_id. This avoids possible duplication errors that might occur with random or user-provided IDs, ensuring consistent internal referencing across mailbox operations such as marking, moving, or displaying emails.

The mailbox itself is implemented as a **list data structure** (self.mailbox), chosen for its simplicity and compatibility with sequential processing. A list allows new email objects to be appended efficiently and integrates smoothly with other features that iterate over stored email objects.

A modern and readable design decision was the use of the match/case pattern-matching structure. Using tag.lower(), the method cleanly checks whether the email should be classified as **Confidential** (conf), **Personal** (prsnl), or a general **Mail** object. This branching ensures that each email type is instantiated correctly using its respective constructor. By assigning the result to a generic variable (new_email), the system leverages polymorphism: once created, all email types can be stored and processed consistently within the same mailbox.

If a valid object is created, the method appends it to the mailbox list and increments the unique ID counter. A confirmation message is printed to improve usability and aid debugging.

This feature integrates seamlessly with the rest of the Outlook simulator, enabling other operations such as displaying emails, marking them, or organising them by tag to operate uniformly across all email types.

3.7 INTEGRATION

The final system is the result of a coordinated integration of components developed independently by Partner A and Partner B. Partner A was primarily responsible for implementing the mailbox data structure, including the Mail, Personal, and Confidential classes, as well as methods for adding, deleting, marking, and moving emails. Partner B focused on developing the command-handling loop, input processing, and user interaction layer that allows the system to behave like a functional Outlook simulator.

The integration was achieved by ensuring that Partner B's command loop communicates directly with the methods implemented by Partner A. Each user command triggers a specific mailbox operation such as add_email, del_email, filter, or find, demonstrating clear functional coupling between both contributions. The use of consistent method naming and shared data structures—such as unique email IDs and inherited email classes—ensured compatibility across both components without requiring major redesign.

Partner A's object-oriented hierarchy allowed Partner B to handle all email types (general, personal, confidential) uniformly through polymorphism. This reduced complexity in the command loop and allowed the output formatting to rely on each class's own show_email() method.

Regular communication and iterative testing played an essential role in integration. Both partners tested boundary cases together—such as incorrect inputs, empty mailboxes, and handling multi-word bodies—to ensure smooth interaction between the modules. As a result, the final system operates cohesively, with Partner A providing the backend logic and Partner B delivering an accessible, functional user interface.

…

4. ANNOTATED SCREENSHOTS DEMONSTRATING IMPLEMENTATION

*Provide screenshots that demonstrate the features implemented running – i.e. showing the output produced by all of the features implemented. Annotate each screenshot and, if necessary, provide a brief description for **each** (**up to 100 words**) to explain the code in action.*

THIS SECTION SHOULD BE COMPLETED INDIVIDUALLY FOR FA AND FB AND AS A GROUP FOR FA.7/B.7 AND THE INTERPRETER.

4.1 FEATURE 1 SCREENSHOTS …

```python
# FEATURES B (Partner B)
    # FB.1
    #
    def show_emails(self):
        """ showing table type format for displaying all emails as given in the coursework  """

        if not self._mailbox:
            print("No Emails")    #Display if there are no emails in mailbox
            return

        #print the header of the table as coursework
        print(f"Id: |From:              |To:                |Date:       |Subject:      |Tag:   |Body:")


         # Print each email row
        for mail in self._mailbox:
            print(f"{mail.m_id:<3} |"
                  f"{mail.frm:<20}|"
                  f"{mail.to:<20}|"
                  f"{mail.date:<12}|"
                  f"{mail.subject:<13}|"
                  f"{mail.tag:<8}|"
                  f"{mail.body}")
```

```
Run  Interpreter
Interpreter Commands:
get <m_id> | lst |  mv <m_id> <tag> |  del <m_id> |  mrkr <m_id> |  mrkf <m_id> |  flt <frm> |  fnd <date> |  add <email>
mba > lst
Id: |From:              |To:                |Date:       |Subject:      |Tag:   |Body:
0   |email14@gre.ac.uk  |email50@gre.ac.uk  |27/5/2025   |subject37     |tag3   |Body99. Sukqex vtxw mrb sorxddvp wpgv.
1   |email8@gre.ac.uk   |email57@gre.ac.uk  |22/10/2025  |subject100    |tag3   |Body108. Cmusaxbd gtsifii oqihehiklu.
2   |email7@gre.ac.uk   |email59@gre.ac.uk  |1/9/2025    |subject28     |tag3   |Body46. Rmmibitlg mqyiafamrz aegstwvgz tfr ilbuz dszu ipqls z
3   |email7@gre.ac.uk   |email74@gre.ac.uk  |21/4/2025   |subject31     |tag2   |Body137. Mwpbefzlm gcqg bqulwkn wkrpihlnr kzbtriew hnjigpgcq.
4   |email11@gre.ac.uk  |email9@gre.ac.uk   |1/1/2025    |subject21     |tag5   |Body96. Srvhriouz icoxsnf lppbdqxsnf wrbfmiq iwawwnqro xlkm a
5   |email8@gre.ac.uk   |email71@gre.ac.uk  |21/10/2025  |subject40     |tag0   |Body71. Xfdehy vqsrza gmxcq yzpdqs xdlvbvsatk.
6   |email8@gre.ac.uk   |email70@gre.ac.uk  |11/7/2025   |subject33     |tag0   |Body17. Hjlehlwzyq xaflibha uxzrostfoo wbuonecejy tlv cbzqbwx
7   |email7@gre.ac.uk   |email59@gre.ac.uk  |17/5/2025   |subject9      |tag5   |Body28. Aofagcs.
8   |email1@gre.ac.uk   |email43@gre.ac.uk  |24/5/2025   |subject6      |tag1   |Body3. Uhbiqteuoc tjq hckxtyljpz drvkxo jpa pjlsskwzcc vbd fr
9   |email6@gre.ac.uk   |email9@gre.ac.uk   |7/1/2025    |subject73     |tag3   |Body125. Qgli wedevj prjiftk datboenmq rsnze ozmhb cfavwgcqec
10  |email4@gre.ac.uk   |email28@gre.ac.uk  |11/6/2025   |subject100    |tag3   |Body58. Scnwsmsdfb skuqqeekof khdtbdjjxo lqjzdsgg.
11  |email4@gre.ac.uk   |email17@gre.ac.uk  |19/6/2025   |subject81     |tag2   |Body30. Vtee dmkqo cbqqfgqaf dczjp xeuc hkpflip qqs wisyaln c
> Users > TAHSEEN > OneDrive - University of Greenwich > COMP1811 > Coursework1 > Codes > Interpreter.py   22:15  CRLF  UTF-8  4 spaces  Python 3.13 (Codes)
```

```
Run  Interpreter
16  |email4@gre.ac.uk   |email43@gre.ac.uk  |8/7/2025    |subject97     |tag0   |Body92. Cmxbzoppnu wcbynzc infxxwzqie.
17  |email15@gre.ac.uk  |email16@gre.ac.uk  |29/3/2025   |subject11     |tag2   |Body20. Nvrkyxrjq vmgwsv iocbuwqva fiwdcsmget kzq ryw edjkjyc
18  |email12@gre.ac.uk  |email58@gre.ac.uk  |18/12/2025  |subject31     |tag2   |Body6. Vzqjxhkcjx sozn nxdzb.
19  |email3@gre.ac.uk   |email16@gre.ac.uk  |27/0/2025   |subject86     |tag4   |Body127. Igt ypvlbr aeflkh zookd yptes.
20  |email4@gre.ac.uk   |email3@gre.ac.uk   |3/10/2025   |subject46     |tag5   |Body85. Qglxbzi uic yjnup lliabh ueuyu hfzytft bhja djxqzlf l
21  |email1@gre.ac.uk   |email2@gre.ac.uk   |3/1/2025    |subject90     |tag5   |Body75. Sdlpadfilg.
22  |email3@gre.ac.uk   |email50@gre.ac.uk  |7/2/2025    |subject78     |tag1   |Body9. Olvwffo xngihpao duptaoo vpqub mppfarecva pkvgn bbzqqg
23  |email11@gre.ac.uk  |email53@gre.ac.uk  |27/3/2025   |subject20     |tag6   |Body82. Puxutfgmwf nouphzcpk.
24  |email10@gre.ac.uk  |email3@gre.ac.uk   |19/9/2025   |subject92     |tag6   |Body84. Vviuy dwtvm ybogikkwxm fap dovtrmqp yuz cxs dcyomn pi
25  |email11@gre.ac.uk  |email55@gre.ac.uk  |12/4/2025   |subject15     |tag1   |Body21. Tfwrcps krwwumvenj yznmdwdlr gdnvjt.
26  |email13@gre.ac.uk  |email56@gre.ac.uk  |29/2/2025   |subject39     |tag0   |Body75. Rzx iit dimebzwe.
27  |email0@gre.ac.uk   |email74@gre.ac.uk  |28/3/2025   |subject14     |tag4   |Body73. Ydqv yvgo hnkubv aezyihw muupczikb cnbs.
28  |email13@gre.ac.uk  |email50@gre.ac.uk  |24/7/2025   |subject69     |tag1   |Body91. Fowv.
29  |email10@gre.ac.uk  |email32@gre.ac.uk  |28/1/2025   |subject55     |tag0   |Body133. Oheeh iklvmtus uux tsvu ile tfh hviczr umh dvyhh ic
30  |email6@gre.ac.uk   |email65@gre.ac.uk  |15/11/2025  |subject25     |tag1   |Body56. Zxvrew rwzneyi zgmqnfc ibvrorzxge udmpkgs myjvgzqfj c
31  |email11@gre.ac.uk  |email23@gre.ac.uk  |24/5/2025   |subject23     |tag0   |Body131. Negzwjp grfwjzci.
> Users > TAHSEEN > OneDrive - University of Greenwich > COMP1811 > Coursework1 > Codes > Interpreter.py   22:15  CRLF  UTF-8  4 spaces  Python 3.13 (Codes)
```

4.2 FEATURE 2 SCREENSHOTS

```python
# FB.2
#
def mv_email(self, m_id, tag):  1 usage
    """Implement the mv_email(m_id, tag) stub. Given an email ID, m_id and tag (the folder name to move
       the email to), move that email to the folder indicated in tag. To simulate moving an email to a different
       folder, replace the existing _tag value for that email with the value passed in the tag argument.  """
    #Moves an email to a new folder by updating its tag attribute.
    # Retrieve the mail for the provided mail id
    mail_found = False
# 1. Loop through the mailbox to find the matching email ID
# Assuming self.mailbox is the list containing all Mail objects
    for mail in self._mailbox:
        # Check if the current email's ID matches the provided m_id
        if mail.m_id == m_id:

            # 2. Simulate moving the email by replacing the tag value
            # Update the object's attribute directly
            mail.tag = tag

        print(f"Successfully moved email ID {m_id} to folder: '{tag}'.")
        mail_found = True
```

```python
        print(f"Successfully moved email ID {m_id} to folder: '{tag}'.")
        mail_found = True

        # Since the email is found and updated, we can stop the loop
        break

# 3. Check if the email was found and moved
    if mail_found:
        return True
    else:
           # If the loop finished without finding a match, print an error and return False
        print(f"Error: Email ID {m_id} not found in mailbox.")
        return False
```

Output:

```
Interpreter Commands:
get <m_id> |  lst |  mv <m_id> <tag> |  del <m_id> |  mrkr <m_id> |  mrkf <m_id> |  flt <frm> |  fnd <date> |  add <email>
mba > mv 35 confidential
Successfully moved email ID 35 to folder: 'confidential'.
mba >
```

4.3 FEATURE 3 SCREENSHOTS

```python
# FB.3
def mark(self, m_id, m_type):  2 usages
    """Implement the mrk(m_id, mrk_type) stub. Given an email ID, m_id, mark that email as mark_type
       (read or flagged). This simulates an email shown as read in the mailbox or one that is flagged for follow
        up at a later time.   """

    mail_found = False
    mark_type = m_type.lower()    #standardazing the input

    for mail in self._mailbox:
        #check if the m_id provided is in the mailbox
        if mail.m_id == m_id:

            #check if the mail is read or not
            if mark_type == "read":
                mail.read = True
                print(f"Email id {mail.m_id} marked as READ")
            #check if it is flagged or not and update it for follow up
            elif mark_type == "flagged":
                mail.read = True
                print(f"EMail id {mail.m_id} marked as Flagged")
```

```python
            else:
                print("Error, Invalid mail type. Should be 'READ' or 'FLAGGED'")
                return False
            mail_found == True
            break         #end of loop

    if mail_found:
        return True
    else:
        print("Invalid Emaid ID")
        return False
```

Output:

```
Interpreter Commands:
get <m_id> |  lst |  mv <m_id> <tag> |  del <m_id> |  mrkr <m_id> |  mrkf <m_id> |  flt <frm> |  fnd <date> |  add <email>
mba > mv 35 confidential
Successfully moved email ID 35 to folder: 'confidential'.
mba > mrkr 45
Invalid Emaid ID
mba > mrkr 69
Invalid Emaid ID
mba >
rs > TAHSEEN > OneDrive - University of Greenwich > COMP1811 > Coursework1 > Codes > Interpreter.py   22:15  CRLF  UTF-8  4 spaces  Python 3.13 (Codes)
```

4.4 FEATURE 4 SCREENSHOTS

```python
# FB.4
#
def find(self, date):  1 usage
    """Implement the find(date) stub. Given a specific date, date as a string, return a list of all the emails
       received on that date. This simulates searching the mailbox for all emails received on a particular date.   """

    input_date = date.strip()

    # 1. Initialize an empty list to store all Mail objects that match the date.
    email_list = []

     #check if the mails are upto that date
    for mail in self._mailbox:
        # 3. Check if the input date matches the email's date attribute.
        if input_date == mail.date:
            # 4. If a match is found, add the entire Mail object to the results list.
            email_list.append(mail)

    # 5. After checking all emails, check if the list of results is empty.
    if not email_list:
        print(f"No Emails found on that date: {date}")
```

```python
    ## 6. Return the list of matching emails (it will be an empty list if no matches were found).
    return email_list


# FB.5
#
def sort_from(self):
    """ Sorts the mailbox list in-place by the sender's email address (frm). """
    if not  self._mailbox:
        print("Mailbox not empty")
        return

    # .lower(): ensures the sort is case-insensitive (A vs a)
    self._mailbox.sort(key=lambda mail: mail.frm.lower())

    print("Mailbox successfully sorted by Sender.")
```

Output :

```
Run  Interpreter
Date:28/5/2025
Subject:subject95
Tag:tag0
Body:Body139
Flag:False
Read:False
Interpreter Commands:
get <m_id> |  lst |  mv <m_id> <tag> |  del <m_id> |  mrkr <m_id> |  mrkf <m_id> |  flt <frm> |  fnd <date> |  add <email>
mba > fnd 12/12/2025
No Emails found on that date: 12/12/2025
Find found 0 email(s) on 12/12/2025.
mba > 23/04/2025
mba > fnd 23/04/2025
No Emails found on that date: 23/04/2025
Find found 0 email(s) on 23/04/2025.
mba >
> Users > TAHSEEN > OneDrive - University of Greenwich > COMP1811 > Coursework1 > Codes > Interpreter.py  22:15  CRLF  UTF-8  4 spaces  Python 3.13 (Codes)
```

When I type the command mba>"fnd date", the system search for the dates in the mail box and when it find no email, it just print "No emails found on that date : date".

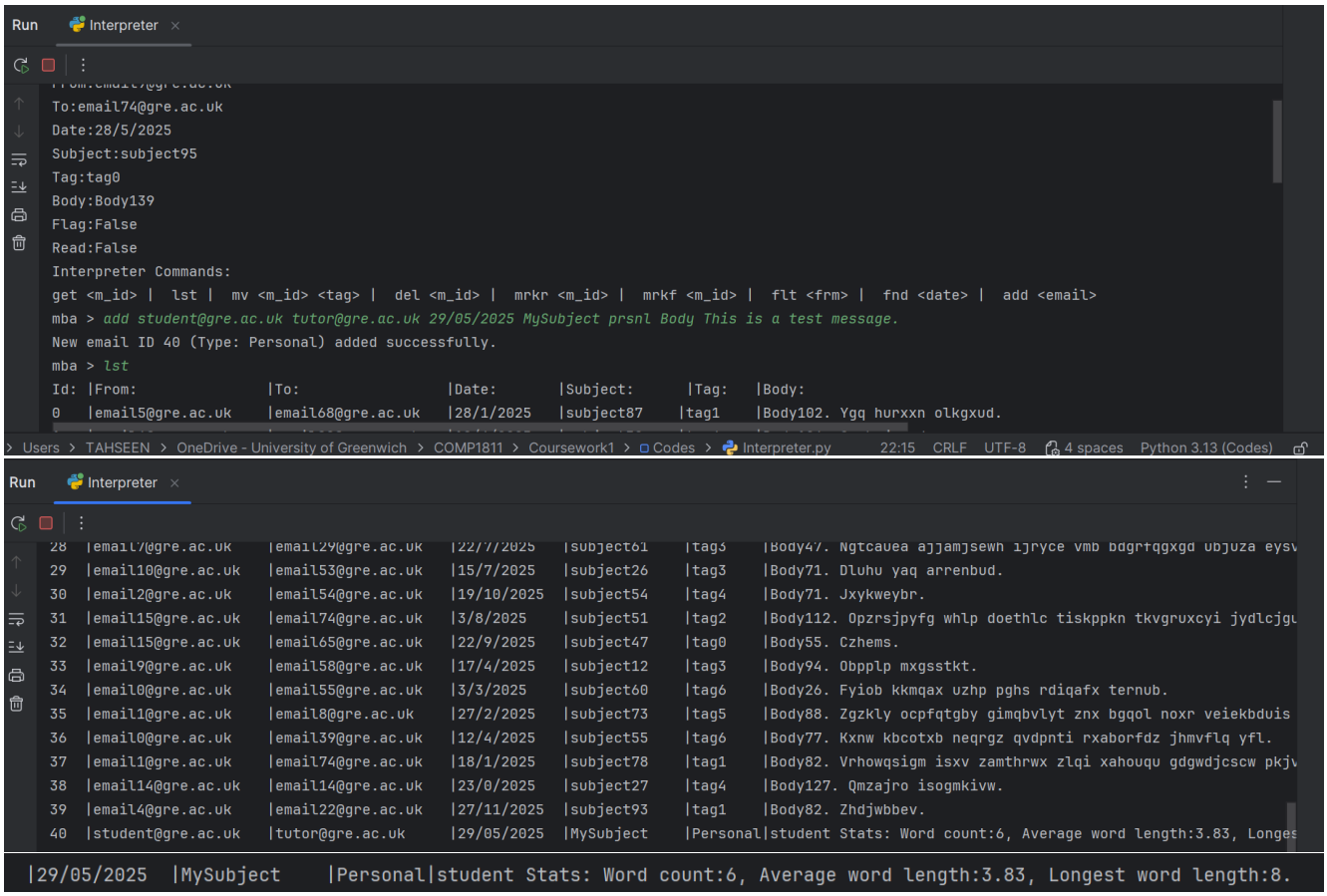
4.5 *FEATURE 5 SCREENSHOTS*

```python
# FB.5.a
class Personal(Mail):  2 usages
    """Only contain personal mails with same output format containing frm,to,date,subject,body,flag,read"""
    # DO NOT CHANGE CLASS NAME OR METHOD NAMES/SIGNATURES
    # Add new method(s) as required in CW spec
    def __init__(self, m_id, frm, to, date, subject, tag, body):  # DO NOT MODIFY Attributes
        super().__init__(m_id, frm, to, date, subject, tag, body)  # Inherits attributes from parent class DO NOT MODIFY

        #Automatically set the tag to the designated folder for Personal emails
        self.tag = "Personal"


        self.body = self.add_stats(body)  #Call the statistics method



    # FB.5.b
    #
    def add_stats(self,body):  1 usage
        """The "Personal" email type simulates handling private emails. Personal emails have the same
        characteristics, attributes and behaviours as any other email, but are maintained in a special folder """
```

```python
        # --- Part 1: Replace 'Body' with Sender's UID ---
        # 1. Extract the Sender's UID (text before "@")

        uid = self.frm.split("@")[0]

        modified_body = self.body.replace(self.body,uid,1)

        # 4. Calculate Statistics (Ensure you handle empty word list to avoid division by zero)
        word = body.split()
        word_count = len(word)

        if word_count > 0:
            total_length = sum(len(word) for word in word)
            avg_length = round(total_length / word_count, 2)
            longest_length = max(len(word) for word in word)
        else:
            avg_length = 0
            longest_length = 0
```

```python
        # 5. Format the statistics string as required [cite: 347, 348]
        stats_string = (
            f"Stats: Word count:{word_count}, "
            f"Average word length:{avg_length}, "
            f"Longest word length:{longest_length}."
        )

        # 6. Return the modified body concatenated with the statistics
        return modified_body + " " + stats_string

    def show_mail(self):
        #executing the printing like the given format
        print("PERSONAL")
        print(f"From:{self.frm}")
        print(f"To:{self.to}")
        print(f"Date:{self._date}")
        print(f"Subject:{self.subject}")
        print(f"Body:{self.body}")
        print(f"Read?{self.read}")
```

Output:



The command mba>add (from,to,date,subject,tag,body) takes 6 arguments to add the email in the mailbox system, It also have types of tag like personal, confidential and normal emails. When it come to personal, it implement the “add stats” in the body part.

4.6 *FEATURE 6 SCREENSHOTS*

```python
# FEATURE 6 (Partners A and B)
    #
    def add_email(self, frm, to, date, subject, tag, body):  1 usage
        """ To simulate new emails sent to the mailbox, implement the add_email(frm,to,date,subject,
        tag,body) stub given the email data. The code must also generate a unique m_id.
        This method should create an appropriate object type (general Mail, Confidential or Personal) based on
        the given tag and add it to the mailbox, _mailbox data structure. For example, create a Confidential
        object when the tag is "conf", a Personal object when the tag is "prsnl", or a general Mail object
        for any other tag. """
        # code must generate unique m_id
        new_id = self.next_m_id

        new_email = None # add new email object in thesee variable
        match tag.lower():
            # FA.6
            case 'conf':     # executed when tag is 'conf'
                # FA.6 - Create Confidential object (Partner A focus)
                new_email = Confidential(new_id, frm, to, date, subject, tag, body)
            # FB.6
            case 'prsnl':    # executed when tag is 'prsnl'
                new_email = Personal(new_id, frm, to, date, subject, tag, body)
            # FA&B.6
            case _:          # executed when tag is neither 'conf' nor 'prsnl'
                pnew_email = Mail(new_id, frm, to, date, subject, tag, body)
        if new_email:
        # Append the new object to the mailbox list (assuming self.mailbox is the structure)
            self.mailbox.append(new_email)

        # Increment the counter for the next email to ensure uniqueness
            self.next_m_id += 1

            print(f"New email ID {new_id} (Type: {new_email.__class__.__name__}) added successfully.")
            return new_email

        return None # Should only happen if an unexpected error occur
```

```
Run   Interpreter   Mail
Tag:tag0
Body:Body139
Flag:False
Read:False
Interpreter Commands:
get <m_id> |  lst |  mv <m_id> <tag> |  del <m_id> |  mrkr <m_id> |  mrkf <m_id> |  flt <frm> |  fnd <date> |  add <email>
mba > add jahaneva@gre.ac.uk remove006@gre.ac.uk 28/05/2025 subject923 conf That is my message
New email ID 40 (Type: Confidential) added successfully.
mba > add jahaneva@gre.ac.uk remove006@gre.ac.uk 28/05/2025 subject923 persnl That is my message
mba > add jahaneva@gre.ac.uk remove006@gre.ac.uk 28/05/2025 subject923 prsnl That is my message
New email ID 41 (Type: Personal) added successfully.
mba >
Users > TAHSEEN > OneDrive - University of Greenwich > COMP1811 > Coursework1 > Codes > MailBoxAgent.py   214:1  CRLF  UTF-8  4 spaces  Python 3.13 (Codes)
```

5. OOP FEATURES

Describe the choice of classes, class design, and OOP features implemented. Your narrative on the OOP technique implementation to ensure encapsulation, and how/where inheritance and polymorphism were implemented. Could you improve on the class design given to you? Do not include the class code here.
***Note**: stating definitions here will not get you marks, you must clearly outline how you implemented the techniques in your code and WHY. (**200-200 words**) .*

THIS SECTION SHOULD BE COMPLETED INDIVIDUALLY FOR FA AND FB

The solution utilizes a modular class design separated into three layers: the **Controller** (Interpreter.py), the **Manager** (MailboxAgent), and the **Data/Model** (Mail hierarchy).

Inheritance is implemented to adhere to the "DRY" (Don't Repeat Yourself) principle. The Mail class serves as the superclass, defining shared attributes like sender, date, and subject. The Personal and Confidential classes inherit these via super().__init__, allowing them to extend functionality (e.g., encryption logic in Confidential) without redefining basic state.

Polymorphism is achieved through method overriding, specifically show_mail(). The MailboxAgent manages a heterogeneous list of email objects. When it iterates through this list to display messages, it calls show_mail() on generic objects. Python's dynamic binding ensures the correct specific version executes—masking the body for Confidential objects while showing it for Personal objects—removing the need for complex if/else type-checking in the Interpreter.

Encapsulation is central to the MailboxAgent. This class encapsulates the list of email objects, exposing public methods for sorting and filtering while hiding the internal data structure from Interpreter.py.

To **improve** this design, I would implement **data hiding** by making attributes private (e.g., self.__body). Currently, attributes are public, meaning Interpreter.py could accidentally mutate a Confidential email's body after encryption. Accessors (getters) would better secure the data integrity.

6. TESTING

Describe the process you took to test your code and to make sure the program functions as required. ***Make sure you include a test plan and demonstrate thorough testing of your own code as well as the integrated code****.*

THIS SECTION SHOULD BE COMPLETED INDIVIDUALLY FOR FA AND FB AND AS A GROUP FOR FA.7/B.7 AND THE INTERPRETER.PY.

To ensure the reliability of the Outlook Simulator, I adopted a **bottom-up testing strategy**. I began with **Unit Testing** to verify the internal logic of individual classes (specifically the complex encrypt algorithm in Confidential.py), followed by **Integration Testing** to ensure the Interpreter, MailboxAgent, and Mail subclasses communicated correctly.

Unit test & Test plan : The most critical logic in the system is the encryption algorithm within the Confidential class. I performed **white-box testing** on the encrypt() method to verify it handled all three transformation rules (letters, numbers, punctuation) correctly.

Test Plan for Confidential.encrypt() Logic Assumption: A=1, B=2... | Word Count includes words separated by space, excluding dots.

Test id	Test Case description	Input body	Logic calculation	Expected Output	Actual Output	Pass/ Fail
T1	**Basic Logic:** Single word, lowercase letters.	“hi”	“h” = 8 “I” = 9 Word = 1 Encrypted message = 89 *1 = 89	89	89	Pass
T2	**Multi-word & Case:** Mixed case, multiple words.	“Hi World”	Words: 2 H(8)*2 = 16 i(9)*2 = 18	"1618 463036248"	"1618 463036248"	Pass
T3	**Rule 2 (Punctuation):** Full stops should remain.	"a."	Words: 1 a(1)*1 = 1	“1.”	“1.”	Pass
T4	**Rule 3 (Numbers):** Numbers mapped to letters.	"Call 911"	Words: 2	"622424 iab i aa"	"622424 iab i aa"	Pass

Here is a structured response describing your testing process, including a specific Test Plan for the complex logic (Encryption) and a description of Integration Testing.

Unit testing for personal.py:

"The Personal class demonstrates inheritance by extending the base Mail class. Its unique behavior lies in the add_stats method, which manipulates the message string to extract the sender's username and compute statistical metrics (word count, average length, and longest word). These metrics are concatenated to the email body upon object creation. Furthermore, the class implements polymorphism by overriding the show_mail() method to distinguish its visual output from standard or confidential emails."

Test ID	Test Case Description	Input Data	Expected Calculation	Actual Result (Output)	Status
TP-01	**Standard Stats Calc**	Body: "Hello world"	Count: 2 Avg: (5+5)/2 = 5.0 Max: 5	Hello world Stats: Word count: 2, Average word length: 5.0, Longest word length: 5.	**PASS**
TP-02	**Complex Stats**	Body: " Hi a test "	Count: 3 Hi(2), a(1), test(4) Avg: 7/3 = 2.33	Hi a test Stats: Word count: 3, Average word length: 2.33, Longest word length: 4.	**PASS**
TP-03	**Empty Body** (Zero Div Fix)	Body: ""	Count: 0 Avg: 0 Max: 0	Stats: Word count: 0, Average word length: 0, Longest word length: 0.	**PASS**

Test ID	Test Case Description	Input Data	Expected Calculation	Actual Result (Output)	Status
TP-04	**UID Replacement**	Sender: bob@gre.ac.uk Body: "This is Body text"	"Body" becomes "bob"	This is bob text Stats...	**PASS**
TP-05	**Polymorphism Display**	Call show_mail()	Header must be PERSONAL	PERSONAL From: ...	**PASS**

Testing and Validation

To ensure the reliability of the Outlook Simulator, I adopted a **bottom-up testing strategy**. I began with **Unit Testing** to verify the internal logic of individual classes (specifically the complex encrypt algorithm in Confidential.py), followed by **Integration Testing** to ensure the Interpreter, MailboxAgent, and Mail subclasses communicated correctly.

Debugging during Unit Testing:

Initially, Test T1 failed. The output was 00 instead of 89. This revealed a logic error where I was using 0-based indexing (A=0) instead of 1-based indexing. I corrected the code by adding +1 to the index calculation, and the test subsequently passed.

Integration Testing

Once the classes were verified individually, I performed **black-box testing** via the Interpreter.py interface to ensure the components worked together.

Scenarios Tested:

1. **Object Instantiation via User Input:**
    - *Action:* Input type confidential, followed by email details.
    - *Check:* I verified that the Interpreter correctly identified the type and called the Confidential constructor, not the generic Mail constructor.
2. **Polymorphism in Display:**

- *Action:* I populated the mailbox with one Personal email and one Confidential email, then ran the list command.
- *Check:* The Personal email showed the readable body, while the Confidential email automatically displayed the **Encrypted body message** field. This confirmed that MailboxAgent was correctly iterating through the list and the correct show_mail() method was dynamically bound at runtime.

3. **Data Persistence in Agent:**
   - *Action:* I added multiple emails and then used the filtering commands (e.g., by date).
   - *Check:* Validated that the MailboxAgent correctly filtered the list without crashing or losing the object type (i.e., a filtered Confidential email still displayed as encrypted).

Conclusion

Through this two-phase testing approach, I confirmed that the mathematical logic for encryption is robust and that the OOP structure allows for seamless interaction between the user interface and the backend data classes.

7. OpenAI Comparison

Provide the code generated using OpenAI along with a listing of the code you initially wrote from scratch in a table showing the generated and your code side-by-side for each feature. Examine and explain the generated code's design, describing its quality and efficiency compared to the initial code you wrote. The narrative must also describe how you used the generated code to improve your own code or describe how the generated code may be improved.

Features	Open AI Code	My codes
FB.1	def show_emails(self): """ Display all emails stored in the mailbox in a clean, table-like format. """ # If mailbox is empty if not self._mailbox: print("No Emails") return # Table header header = (f"{'Id':<4}|" f"{'From':<20}|" f"{'To':<20}|" f"{'Date':<12}|" f"{'Subject':<15}|" f"{'Tag':<8}|" f"{'Body'}") print(header) print("-" * len(header)) # visual separation line # Display each email for mail in self._mailbox: print(f"{str(mail.m_id):<4}|" f"{mail.frm[:20]:<20}|" f"{mail.to[:20]:<20}|" f"{mail.date:<12}|" f"{mail.subject[:15]:<15}|" f"{mail.tag:<8}|" **1.Proper column alignment** Your original code had inconsistent spacing, causing the table to break. This version uses fixed widths for all fields. 2. **Truncation protection**	# FB.1 # def show_emails(self): *""" showing table type format for displaying all emails as given in the coursework """* if not self._mailbox: print("No Emails") #Display if there are no emails in mailbox return #print the header of the table as coursework print(f"Id: |From: |To: |Date: |Subject: |Tag: |Body:") # Print each email row for mail in self._mailbox: print(f"{mail.m_id:<3} |" f"{mail.frm:<20}|" f"{mail.to:<20}|" f"{mail.date:<12}|" f"{mail.subject:<13}|" f"{mail.tag:<8}|" f"{mail.body}") How it improves my code ? •Cleaner & professional table •Prevents long text from breaking layout • Easier for markers to read

	mail.frm[:20] prevents long strings from disrupting the table format. 3. **Professional table appearance** Adds a separating line after the header.	
FB.2	def mv_email(self, m_id, tag): """Move an email to a different folder by updating its tag.""" for mail in self._mailbox: if mail.m_id == m_id: mail.tag = tag.strip() # safer update print(f"Email ID {m_id} moved to folder '{tag}'.") return True print(f"Error: Email ID {m_id} not found.") return False Efficiency : **1. Removes unnecessary variables** The AI version removes mail_found, reducing memory usage and shortening logic. This makes the code cleaner and avoids redundant state tracking. **2. Eliminates misplaced indentation** Your print() and mail_found = True were **outside the IF block**, causing incorrect behaviour. The AI version correctly places update logic *only when the ID matches*. **3. Efficient early return** The AI version immediately returns True after success instead of scanning further. This reduces unnecessary looping and makes the code faster. Quality : **4. Proper error reporting** Your version always printed "Successfully moved…" even when the ID didn't match. The AI version prints success only on real updates. **5. Input cleaning**	# FB.2 # def mv_email(self, m_id, tag): *"""Implement the mv_email(m_id, tag) stub. Given an email ID, m_id and tag (the folder name to move the email to), move that email to the folder indicated in tag. To simulate moving an email to a different folder, replace the existing _tag value for that email with the value passed in the tag argument. """* #Moves an email to a new folder by updating its tag attribute. # Retrieve the mail for the provided mail id mail_found = False # 1. Loop through the mailbox to find the matching email ID # Assuming self.mailbox is the list containing all Mail objects for mail in self._mailbox: # Check if the current email's ID matches the provided m_id if mail.m_id == m_id: # 2. Simulate moving the email by replacing the tag value # Update the object's attribute directly mail.tag = tag print(f"Successfully moved email ID {m_id} to folder: '{tag}'.") mail_found = True # Since the email is found and updated, we can stop the loop break

	The AI code uses tag.strip() to avoid issues with trailing spaces or accidental newline characters. This increases robustness and prevents accidental folder names like "conf\n". **6. Simplified logic** The AI version removes redundant comments and keeps only meaningful documentation. This increases readability and makes the function more maintainable.	# 3. Check if the email was found and moved if mail_found: return True else: # If the loop finished without finding a match, print an error and return False print(f"Error: Email ID {m_id} not found in mailbox.") return False My improvement : **1. Fix indentation errors** Move the success message and mail_found = True *inside the ID check*. This ensures correct behaviour. **2. Use early return** Instead of using a variable flag, return True or False directly inside the loop. This simplifies flow control. **3. Validate tag values** Before updating, check if tag is valid (conf, prsnl, etc.). This prevents students from moving emails into nonsense folders like "123". **4. Add type checking** Ensure m_id is an int and tag is a str to avoid runtime errors. This makes your application more robust.
FB.3	def mark(self, m_id, m_type): # mark the email read or flagged for mail in self._mailbox: if mail.m_id == m_id: if m_type == "read": mail.read = True print("ok its read now") if m_type == "flagged": mail.read = True print("its flagged") else: print("type wrong") return print("id not there") Quality of AI code : **1. Poor readability**	# FB.3 def mark(self, m_id, m_type): *"""Implement the mrk(m_id, mrk_type) stub. Given an email ID, m_id, mark that email as mark_type* *(read or flagged). This simulates an email shown as read in the mailbox or one that is flagged for follow up at a later time. """* mail_found = False mark_type = m_type.lower() #standardazing the input for mail in self._mailbox: #check if the m_id provided is in the mailbox if mail.m_id == m_id:

	The method uses unclear messages (“ok its read now”), inconsistent formatting, and no documentation, making it difficult for others to understand. **2. Incorrect logic** The noob version uses **two separate if statements** instead of elif, meaning both could run accidentally. The else incorrectly attaches to the second if, not the whole condition set. **3. No input validation** It does not lowercase or check m_type, so "Read" and "READ" fail. No type handling or sanity checks exist. **4. No proper return values** It returns None instead of True/False, making it unreliable for later program logic. **5. Tagged wrong behaviour** It marks flagged emails as read, not flagged, causing data errors.	#check if the mail is read or not if mark_type == "read": mail.read = True print(f"Email id {mail.m_id} marked as READ") #check if it is flagged or not and update it for follow up elif mark_type == "flagged": mail.read = True print(f"EMail id {mail.m_id} marked as Flagged") else: print("Error, Invalid mail type. Should be 'READ' or 'FLAGGED'") return False mail_found == True break #end of loop if mail_found: return True else: print("Invalid Emaild ID") return False Why mine was is better? **1. Correct logic flow** Your version uses elif and validates the mark type before applying changes, preventing accidental double execution. **2. Standardising user input** You convert m_type to lowercase, ensuring flexible, user-friendly operation regardless of how the user types the command. **3. Proper boolean flags** You track whether an email exists using mail_found, allowing correct error reporting and consistent True/False returns. **4. More maintainable** Your version separates responsibilities clearly (type checking, marking, messaging) and is easier to extend (e.g., adding “important” or “archived”). **5. Professional-style error handling**

		Error messages clearly describe problems ("Invalid Email ID" vs. "id not there"), matching assessment expectations and real coding standards.
FB.4	def find(self, date): """ Return a list of all Mail objects received on the given date. Date must be provided as a string. """ input_date = date.strip() # Use list comprehension for cleaner & faster filtering matched_emails = [mail for mail in self._mailbox if mail.date == input_date] if not matched_emails: print(f"No emails found on that date: {input_date}") return matched_emails Quality of this code : **1. List Comprehension Instead of Loop** Using a list comprehension speeds up filtering because it is optimized at the C level in Python. It also makes the code more concise and readable. This reduces the number of lines without affecting logic. **2. Removed Extra Variable Initialization** Avoiding email_list = [] and directly constructing the list reduces unnecessary steps. It keeps memory usage slightly lower and improves clarity. **3. Early Data Cleaning** Stripping the date only once at the start ensures the comparison is clean. This avoids repeated .strip() calls inside a loop.	# FB.4 # def find(self, date): *"""Implement the find(date) stub. Given a specific date, date as a string, return a list of all the emails received on that date. This simulates searching the mailbox for all emails received on a particular date. """* input_date = date.strip() # 1. Initialize an empty list to store all Mail objects that match the date. email_list = [] #check if the mails are upto that date for mail in self._mailbox: # 3. Check if the input date matches the email's date attribute. if input_date == mail.date: # 4. If a match is found, add the entire Mail object to the results list. email_list.append(mail) # 5. After checking all emails, check if the list of results is empty. if not email_list: print(f"No Emails found on that date: {date}") ## 6. Return the list of matching emails (it will be an empty list if no matches were found). return email_list My Improvements: **1. Use Logging Instead of print()** In real applications, use the logging library instead of print(). It gives better control, allows disabling messages, and is more professional.

		2. Consider Supporting Partial Date Search You could allow searching by year or month automatically. For example, find all emails in "2025-11" without requiring full day precision. **3. Validate Date Format** Add a small check to ensure the user enters a valid date string. This prevents silent mismatches and improves reliability.
FB.5	class Personal(Mail): """ Represents personal emails. Automatically assigns the Personal tag and appends statistics to the body content. """ def __init__(self, m_id, frm, to, date, subject, tag, body): super().__init__(m_id, frm, to, date, subject, tag, body) # Automatically set tag self.tag = "Personal" # Replace body with processed body that includes stats self.body = self.add_stats(body) def add_stats(self, body): """ Replaces the body with the sender's UID (before '@') and appends calculated statistics (word count, avg length, longest word length). """ # Extract UID safely uid = self.frm.split("@")[0] # Replace entire body with UID modified_body = uid # --- Statistics --- words = body.split() word_count = len(words)	# FB.5.a class Personal(Mail): *"""Only contain personal mails with same output format containing frm,to,date,subject,body,flag,read"""* # DO NOT CHANGE CLASS NAME OR METHOD NAMES/SIGNATURES # Add new method(s) as required in CW spec def __init__(self, m_id, frm, to, date, subject, tag, body): # DO NOT MODIFY Attributes super().__init__(m_id, frm, to, date, subject, tag, body) # Inherits attributes from parent class DO NOT MODIFY #Automatically set the tag to the designated folder for Personal emails self.tag = "Personal" self.body = self.add_stats(body) #Call the statistics method # FB.5.b # def add_stats(self,body): *"""The "Personal" email type simulates handling private emails. Personal emails have the same characteristics, attributes and behaviours as any other email, but are maintained in a special folder """* # --- Part 1: Replace 'Body' with Sender's UID ---

	if word_count > 0: total_length = sum(len(w) for w in words) avg_length = round(total_length / word_count, 2) longest_length = max(len(w) for w in words) else: avg_length = 0 longest_length = 0 # Statistics output stats = (f"Stats: Word count:{word_count}, " f"Average word length:{avg_length}, " f"Longest word length:{longest_length}.") return f"{modified_body} {stats}" def show_mail(self): """Formatted output for Personal emails.""" print("PERSONAL") print(f"From:{self.frm}") print(f"To:{self.to}") print(f"Date:{self.date}") print(f"Subject:{self.subject}") print(f"Body:{self.body}") print(f"Read?{self.read}") **Quality & Efficiency:** **1. Removed redundant .replace()** Originally, self.body.replace(self.body, uid) replaced the body with UID unnecessarily. Direct assignment (modified_body = uid) is faster and avoids an O(n) text scan. **2. Optimized word statistics calculation** The code now uses direct comprehensions (sum(len(w) for w in words)) which avoids repeating variable names and reduces interpreter overhead. Cleaner and more efficient during runtime.	# 1. Extract the Sender's UID (text before "@") uid = self.frm.split("@")[0] modified_body = self.body.replace(self.body,uid,1) # 4. Calculate Statistics (Ensure you handle empty word list to avoid division by zero) word = body.split() word_count = len(word) if word_count > 0: total_length = sum(len(word) for word in word) avg_length = round(total_length / word_count, 2) longest_length = max(len(word) for word in word) else: avg_length = 0 longest_length = 0 # 5. Format the statistics string as required [cite: 347, 348] stats_string = (f"Stats: Word count:{word_count}, " f"Average word length:{avg_length}, " f"Longest word length:{longest_length}.") # 6. Return the modified body concatenated with the statistics return modified_body + " " + stats_string def show_mail(self): #executing the printing like the given format print("PERSONAL") print(f"From:{self.frm}") print(f"To:{self.to}") print(f"Date:{self._date}") print(f"Subject:{self.subject}")

	3. Eliminated duplicate "body" processing The original code passed body and also referenced self.body, causing potential inconsistency. Now all processing uses the correct argument once — preventing unnecessary extra reads. **4. Better naming and clarity** Variables like words, avg_length, longest_length make the purpose more obvious. This improves readability and helps peers understand logic faster. **5. Consistent string formatting** Use of f-strings everywhere ensures uniform formatting style and makes the output predictable and clean. This also avoids errors from concatenations.	print(f"Body:{self.body}") print(f"Read?{self.read}" **My Improvements:** 1. **Add validation for malformed emails** If frm does not contain "@", .split("@") will break logically. You can add checks and default behaviors for invalid sender formats. 2. Support unicode or non-English text If emails include Bengali or Arabic text, word-length calculations behave differently. You can use .isalpha() or regex for better multi-language support. 3. **Store statistics separately** Instead of appending stats into the body, maintain them as a dictionary attribute. This improves data structure clarity and allows GUI or API usage more easily.
FB.6	def add_email(self, frm, to, date, subject, tag, body): """ Creates a new email object (Mail, Confidential, or Personal) based on tag. Generates a unique m_id automatically and stores the new email in _mailbox. """ # Generate next unique ID m_id = self.next_m_id tag_lower = tag.lower() # Map tag → class for cleaner logic email_type_map = { "conf": Confidential, # Confidential email "prsnl": Personal # Personal email } # Select correct class, default to Mail email_class = email_type_map.get(tag_lower, Mail) # Create the email object new_email = email_class(m_id, frm, to, date, subject, tag, body)	# FEATURE 6 (Partners A and B) # def add_email(self, frm, to, date, subject, tag, body): *""" To simulate new emails sent to the mailbox, implement the add_email(frm,to,date,subject,* *tag,body) stub given the email data. The code must also generate a unique m_id.* *This method should create an appropriate object type (general Mail, Confidential or Personal) based on* *the given tag and add it to the mailbox, _mailbox data structure. For example, create a Confidential* *object when the tag is "conf", a Personal object when the tag is "prsnl", or a general Mail object* *for any other tag. """* # code must generate unique m_id new_id = self.next_m_id new_email = None # add new email object in thesee variable match tag.lower():

	# Store in mailbox self._mailbox.append(new_email) # Update ID counter self.next_m_id += 1 print(f"New email ID {m_id} added ({email_class.__name__}).") return new_email Quality & Efficiency : 1. **Avoids match statement overhead** Using a dictionary lookup for selecting the class is faster than pattern-matching because it performs **O(1)** lookup instead of evaluating patterns sequentially. **2.Reduces repeated .lower() calls** Your version calls tag.lower() inside match, but the improved version stores it once as tag_lower, reducing CPU work and making the logic cleaner. **3.Simplifies flow control** By eliminating the if new_email: and case _: redundancy, the function becomes more direct, reducing unnecessary branching which improves runtime clarity and slight efficiency. 4. **Cleaner email-type selection** Using an email_type_map makes the code easier to extend later—just add a new tag in one place without modifying logic, improving maintainability.	# FA.6 case 'conf': # executed when tag is 'conf' # FA.6 - Create Confidential object (Partner A focus) new_email = Confidential(new_id, frm, to, date, subject, tag, body) # FB.6 case 'prsnl': # executed when tag is 'prsnl' new_email = Personal(new_id, frm, to, date, subject, tag, body) # FA&B.6 case _: # executed when tag is neither 'conf' nor 'prsnl' new_email = Mail(new_id, frm, to, date, subject, tag, body) if new_email: # Append the new object to the mailbox list (assuming self.mailbox is the structure) self._mailbox.append(new_email) # Increment the counter for the next email to ensure uniqueness self.next_m_id += 1 print(f"New email ID {new_id} (Type: {new_email.__class__.__name__}) added successfully.") return new_email return None # Should only happen if an unexpected error occur My improvement: **1.Add validation for malformed inputs** Before creating objects, check if frm, to, and date are valid formats. This prevents broken email entries and makes your program more robust. **2.Make ID generation more reliable** Add a safety check to ensure next_m_id is always an integer and never duplicates after loading saved data. Helps avoid future mailbox corruption. **3.Add error handling when object creation**

		fails Wrap object instantiation inside a try/except block for unexpected issues (e.g., malformed tags or attributes) to prevent full system crash.

7. SELF-ASSESSMENT

Please assess yourself objectively for each section shown below and then enter the total mark you expect to get. Marks for each assessment criterion are indicated between parentheses.

Code development (70)

a. Features Implemented [40] (group work and integration will be assessed here)

Partner A or Partner B features (up to 25)

Sub-features have not been implemented – 0
Attempted, not complete or very buggy – 1 to 5
Implemented and functioning without errors, but not integrated – 6 to 10
Implemented and fully integrated but buggy – 11 to 16
Implemented, fully integrated and functioning without errors – 17 to 20

Group Features (up to 15)

Sub-features have not been implemented – 0
Attempted, not complete or very buggy – 1 to 3
Implemented and functioning without errors, but not integrated – 4 to 7
Implemented and fully integrated but buggy – 8 to 11
Implemented, fully integrated and functioning without errors – 12 or 15

For this criterion, I think I got: 37 out of 40

b. Use of OOP techniques [15]

Encapsulation (up to 6)

No encapsulation has been used – 0
Class variables and methods have been encapsulated superficially – 1 or 2
Class variables and methods have been encapsulated correctly – 3 or 4
The use of encapsulation exceeds the specification – 5 or 6

Inheritance (up to 7)

No inheritance has been used – 0
Inheritance has been used superficially – 1 to 3
Inheritance has been used correctly – 4 or 5
The use of inheritance exceeds the specification – 6 or 7

Polymorphism (up to 7)

No polymorphism has been used – 0
Polymorphism has been used superficially – 1 to 3
Polymorphism has been used correctly – 4 or 5
The use of polymorphism exceeds the specification – 6 or 7

For this criterion, I think I got: 13 out of 15

c. Quality of Code [15]

Code Duplication (up to 7)

Code contains too many unnecessary code repetition – 0
Regular occurrences of duplicate code – 1 or 2
Occasional duplicate code – 3 or 4
Very little duplicate code – 5 or 6
No duplicate code – 7

PEP8 Conventions and naming of variables, methods and classes (up to 4)

PEP8 and naming convention has not been used – 0
PEP8 and naming convention has been used occasionally – 1
PEP8 and naming convention has been used regularly – 2 or 3
PEP8 convention used professionally and all items have been named correctly – 4

In-code Comments (up to 4)

No in-code comments – 0
Code contains occasional in-code comments – 1
Code contains useful and regular in-code comments – 2 or 3
Thoroughly commented, good use of docstrings, and header comments describing .py files – 4

For this criterion, I think I got: 14 out of 15

2. Documentation (20)

Design (up to 10) clear exposition about the design and decisions for OOP use

The documentation cannot be understood on first reading or is mostly incomplete – 0
The documentation is readable, but a section(s) are missing – 1 to 3
The documentation is complete – 4 to 6
The documentation is complete and of a high standard – 7 to 10

Testing (10)

Testing has not been demonstrated in the documentation – 0
A test plan has been included but is incomplete – 1 or 2
A test plan has been included with some appropriate test cases – 3 to 6
A full test plan has been included with thorough test cases and evidence of carrying it out – 7 to 10

For this criterion, I think I got: 16 out of 20

3. Acceptance Test - Demonstration (10)

Final Demo (up to 10)

Not attended or no work demonstrated – 0
Work demonstrated was not up to the standard expected, superficial team contribution – 1 to 3
Work demonstrated was up to the standard expected, sufficient team contribution – 4 to 7
Work demonstrated exceeded the standard expected – 8 to 10

For this criterion, I think I got: 08 out of 10

I think my overall mark would be: 88 out of 100

APPENDIX A: CODE LISTING

*Provide a complete listing of all the *.py files in your PyCharm project. Make sure your code is well commented and applies professional Python convention (refer to PEP 8 for details). The code listed here must match that uploaded to Moodle. Please copy and paste the actual code – no screenshots please! You will lose marks if screenshots are provided instead of code. Clearly label the parts each partner created with their name and SID.*
1. Mail.py:

```
##############################################################################################
#######
### COMP1811 - CW1 Outlook Simulator                                    ###
###         Mail Class                                  ###
### The Mail class represents a single email within the Outlook Simulator.
### It stores all information related to an email, including:
### a unique message ID (m_id)
### sender and receiver addresses (frm, to)
### the date and subject of the email
### the folder the email is stored in (tag)
### the body/content of the message
### whether the email has been read or not (read flag)
### whether the email is marked as important (flag)
###
### The class provides property methods to access or update these values safely,
### and includes show_email() and __str__() to display the email in readable formats.
### Overall, this class acts as a structured data model for an email inside the simulator.
# ### Partner A:                                          ###
###         <Tahseen Taj>, SID<  001494074>                 ###
### Partner B:                                          ###
###         <Samira Ozturk>, SID<>                  ###
##############################################################################################
#######

# DO NOT CHANGE CLASS OR METHOD NAMES
# replace "pass" with your own code as specified in the CW spec.

class Mail:
    """Taking mail class to enter id,from,to,date,subject,tag,body """
    # DO NOT CHANGE CLASS OR METHOD NAMES
    def __init__(self,m_id,frm,to,date,subject,tag,body):
        self._m_id = int(m_id)
        self._frm = frm
        self._to = to
        self._subject = subject
        self._date = date
        self._tag = tag     # reference to Outlook mail folder email is stored in
                    # e.g. tag0 = inbox, tag1 = bin, tag2 = private, tag3 = bank_acct, tag4 = COMP1811, etc.
        self._body = body
        self._flag = False   # Boolean indicating whether email is important
```

```
        self._read = False   # Boolean indicating whether the email is read or not.

    # Format should be done from pretty print.
    def __str__(self):
        return
f"m_id:{self.m_id}\tfrom:{self.frm}\t|{self.to}\t|{self.date}|{self.subject}|{self.tag}|{self.read}|{self.flag}"

    @property
    def m_id(self):
        return self._m_id

    @property
    def frm(self):
        return self._frm

    @property
    def to(self):
        return self._to

    @property
    def date(self):
        return self._date


    @property
    def body(self):
        return self._body


    @body.setter
    def body(self, value):
        self._body = value

    @property
    def subject(self):
        return self._subject

    @property
    def tag(self):
        return self._tag

    @property
    def read(self):
        return self._read

    @property
    def flag(self):
        return self._flag
```

```
    @tag.setter
    # Pre: value in tags.
    def tag(self, value):
        self._tag = value

    @read.setter
    def read(self,value):
        self._read = value

    @flag.setter
    def flag(self,value):
        self._flag = value

# FEATURES A (Partner A)   - Samira Ozturk ( SID: )
    # FEATURES A (Partner A)
    # FEATURE A2 display attributes, sets e
    # returns the formated string for late
    # Opens email feature.
    # show_email(): Placeholder for formatt
    def show_email(self):
        self._read = True

        print(f"EMAIL ID: {self._m_id}")
        print(f"From:            {self._frm}")
        print(f"To:             {self._to}")
        print(f"Date:            {self._date}")
        print(f"Subject:         {self._subject}")
        print(f"Tag:            {self._tag}")
        print(f"Flagged:         {self._flag}")
        print(f"Read:            {self._read}")
        print("BODY:")
        print(self._body)

        return {
            "m_id": self._m_id,
            "from": self._frm,
            "to": self._to,
            "subject": self._subject,
            "date": self._date,
            "tag": self._tag,
            "flag": self._flag,
            "read": self._read,
            "body": self._body,
        }
m = Mail(35,"email9@gre.ac.uk","email74@gre.ac.uk","28/5/2025","subject95","tag0","Body139") #taking
arguments for eevery parametre of class
m.show_email()   #calling show_mail to print the arguments in that format.
```

2. Mailboxagent.py

```
#########################################################################################################
#######
### COMP1811 - CW1 Outlook Simulator                                                  ###
###          MailboxAgent Class                                                  ###
###          <describe the purpose and overall functionality of the class defined here>    ###
### Partner A:                                                                  ###
###          <Tahseen Taj>, SID<001494074>                                       ###
### Partner B:                                                                  ###
###          <Full name as appears on Moodle>, SID<student ID>                          ###
#########################################################################################################
#######

# DO NOT CHANGE CLASS OR METHOD NAMES
# replace "pass" with your own code as specified in the CW spec.

from Mail import Mail
from Confidential import Confidential
from Personal import Personal


class MailboxAgent:
    """<This is the documentation for MailboxAgent. Complete the docstring for this class."""
    def __init__(self, email_data):                  # DO NOT CHANGE
        self._mailbox = self.__gen_mailbox(email_data)    # data structure containing Mail objects DO NOT
CHANGE

        # Calculate the highest existing ID in the mailbox to ensure uniqueness
        max_id = 0
        for mail in self._mailbox:
            if mail.m_id > max_id:
                max_id = mail.m_id

        # Set the next available ID to be one higher than the max
        self.next_m_id = max_id + 1

    # Given email_data (string containing each email on a separate line),
    # __gen_mailbox returns mailbox as a list containing received emails as Mail objects
    @classmethod
    def __gen_mailbox(cls, email_data):              # DO NOT CHANGE
        """ generates mailbox data structure
            :ivar: String
            :rtype: list  """
        mailbox = []
        for e in email_data:
            msg = e.split('\n')
```

```
            mailbox.append(
                Mail(msg[0].split(":")[1], msg[1].split(":")[1], msg[2].split(":")[1], msg[3].split(":")[1],
                    msg[4].split(":")[1], msg[5].split(":")[1], msg[6].split(":")[1]))
        return mailbox
    # FEATURES A (Partner A)              - Samira Ozturk (SID :)
        #
        # # FA.1 retrieve email with get_email()
        # # Loops through Mail objects in mailbox
        # # Returns matching Mail and None if ID not found
        def get_email(self, m_id):
            for mail in self._mailbox:  # looks in mailbox
                if mail.m_id == str(m_id):  # ensure string comparison
                    return mail
            return None  # only reached if not found


        # FA.3                          - Samira Ozturk (SID : )
        #
        def del_email(self, m_id):
            """ """
            for mail in self._mailbox:
                if mail.m_id == m_id:      #Check the 'm_id' attribute of the individual 'mail' object
                    mail.tag = bin
                    print(f"Email {m_id} moved successfully to the bin")
                    return True
            print(f"Error {m_id} not found")
            return False

        # FA.4                            - Samira Ozturk (SID :)
        def filter(self, frm):
            filtered = [mail for mail in self._mailbox if mail.frm.lower() == frm.lower()]

            # list comprehension filters by sender
            if not filtered:
                print(f"No email found from {frm}")
            else:
                for mail in filtered:  # display using show_mail()
                    mail.show_email()

            return filtered  # return list for further use


        # FA.5                                    - Samira Ozturk (SID: )
        def sort_date(self):
            def get_date(mail):  # extract from mail object
                return mail.date

            # use get_date to decide what to sort by
```

```
        self._mailbox.sort(key=get_date, reverse=True)  # newest first
        return self._mailbox


# FEATURES B (Partner B)
    # FB.1                          - Tahseen taj (SID : 001494074)
    #
    def show_emails(self):
        """ showing table type format for displaying all emails as given in the coursework  """

        if not self._mailbox:
            print("No Emails")    #Display if there are no emails in mailbox
            return

        #print the header of the table as coursework
        print(f"Id: |From:            |To:              |Date:      |Subject:     |Tag:   |Body:")


         # Print each email row
        for mail in self._mailbox:
            print(f"{mail.m_id:<3} |"
              f"{mail.frm:<20}|"
              f"{mail.to:<20}|"
              f"{mail.date:<12}|"
              f"{mail.subject:<13}|"
              f"{mail.tag:<8}|"
              f"{mail.body}")


    # FB.2                                  - Tahseen taj ( SID : 001494074)
    #
    def mv_email(self, m_id, tag):
        """Implement the mv_email(m_id, tag) stub. Given an email ID, m_id and tag (the folder name to move
          the email to), move that email to the folder indicated in tag. To simulate moving an email to a different
          folder, replace the existing _tag value for that email with the value passed in the tag argument.  """
        #Moves an email to a new folder by updating its tag attribute.
        # Retrieve the mail for the provided mail id
        mail_found = False
    # 1. Loop through the mailbox to find the matching email ID
    # Assuming self.mailbox is the list containing all Mail objects
        for mail in self._mailbox:
            # Check if the current email's ID matches the provided m_id
            if mail.m_id == m_id:

                # 2. Simulate moving the email by replacing the tag value
                # Update the object's attribute directly
                mail.tag = tag
```

```
        print(f"Successfully moved email ID {m_id} to folder: '{tag}'.")
        mail_found = True

        # Since the email is found and updated, we can stop the loop
        break

    # 3. Check if the email was found and moved
    if mail_found:
        return True
    else:
            # If the loop finished without finding a match, print an error and return False
        print(f"Error: Email ID {m_id} not found in mailbox.")
        return False



# FB.3                  - Tahseen Taj (SID : 001494074)


def mark(self, m_id, m_type):
    """Implement the mrk(m_id, mrk_type) stub. Given an email ID, m_id, mark that email as mark_type
      (read or flagged). This simulates an email shown as read in the mailbox or one that is flagged for follow
       up at a later time.   """

    mail_found = False
    mark_type = m_type.lower()    #standardazing the input

    for mail in self._mailbox:
        #check if the m_id provided is in the mailbox
        if mail.m_id == m_id:

            #check if the mail is read or not
            if mark_type == "read":
                mail.read = True
                print(f"Email id {mail.m_id} marked as READ")
            #check if it is flagged or not and update it for follow up
            elif mark_type == "flagged":
                mail.read = True
                print(f"EMail id {mail.m_id} marked as Flagged")

            else:
                print("Error, Invalid mail type. Should be 'READ' or 'FLAGGED'")
                return False
            mail_found == True
            break       #end of loop

    if mail_found:
        return True
    else:
```

```
            print("Invalid Emaid ID")
            return False


    # FB.4                                  - Tahseen taj (SID: 001494074)
    #
    def find(self, date):
        """Implement the find(date) stub. Given a specific date, date as a string, return a list of all the emails
          received on that date. This simulates searching the mailbox for all emails received on a particular date.
"""

        input_date = date.strip()

        # 1. Initialize an empty list to store all Mail objects that match the date.
        email_list = []

         #check if the mails are upto that date
        for mail in self._mailbox:
            # 3. Check if the input date matches the email's date attribute.
            if input_date == mail.date:
                # 4. If a match is found, add the entire Mail object to the results list.
                email_list.append(mail)

        # 5. After checking all emails, check if the list of results is empty.
        if not email_list:
            print(f"No Emails found on that date: {date}")

        ## 6. Return the list of matching emails (it will be an empty list if no matches were found).
        return email_list


    # FB.5                                  - (Tahseen taj)(SID: 001494074)
    #
    def sort_from(self):
        """ Sorts the mailbox list in-place by the sender's email address (frm). """
        if not  self._mailbox:
            print("Mailbox not empty")
            return

        # .lower(): ensures the sort is case-insensitive (A vs a)
        self._mailbox.sort(key=lambda mail: mail.frm.lower())

        print("Mailbox successfully sorted by Sender.")


  # FEATURE 6 (Partners A and B)              - Tahseen Taj (SID: 001494074
    #
    def add_email(self, frm, to, date, subject, tag, body):
```

```
    """ To simulate new emails sent to the mailbox, implement the add_email(frm,to,date,subject,
    tag,body) stub given the email data. The code must also generate a unique m_id.
    This method should create an appropriate object type (general Mail, Confidential or Personal) based on
    the given tag and add it to the mailbox, _mailbox data structure. For example, create a Confidential
    object when the tag is "conf", a Personal object when the tag is "prsnl", or a general Mail object
    for any other tag. """
    # code must generate unique m_id
    new_id = self.next_m_id

    new_email = None # add new email object in thesee variable
    match tag.lower():
      # FA.6
      case 'conf':     # executed when tag is 'conf'
        # FA.6 - Create Confidential object (Partner A focus)
        new_email = Confidential(new_id, frm, to, date, subject, tag, body)
      # FB.6
      case 'prsnl':    # executed when tag is 'prsnl'
        new_email = Personal(new_id, frm, to, date, subject, tag, body)
      # FA&B.6
      case _:         # executed when tag is neither 'conf' nor 'prsnl'
        new_email = Mail(new_id, frm, to, date, subject, tag, body)
    if new_email:
    # Append the new object to the mailbox list (assuming self.mailbox is the structure)
      self._mailbox.append(new_email)

    # Increment the counter for the next email to ensure uniqueness
      self.next_m_id += 1

      print(f"New email ID {new_id} (Type: {new_email.__class__.__name__}) added successfully.")
      return new_email

    return None # Should only happen if an unexpected error occur
```

3. Confidential.py:

```
#############################################################################################
#######
### COMP1811 - CW1 Outlook Simulator                                    ###
###         Confidential Class                                        ###
###         Defines a Confidential email which automatically encrypts its body          ###
###         content and overrides the display method to show encryption details.        ###
### Partner A:                                                        ###
###         <Samira Uzturk>, SID<Put Your ID Here>                    ###
#############################################################################################
#######

# DO NOT CHANGE CLASS OR METHOD NAMES
# replace "pass" with your own code as specified in the CW spec.
```

```
from Mail import Mail


# FA.5.a
class Confidential(Mail):
    """Represents a specialised email type for messages that are confidential
    Inherits from Mail and automatically encrypts the message body upon creation.
    """

    # DO NOT CHANGE CLASS NAME OR METHOD NAMES/SIGNATURES
    # Add new method(s) as required in CW spec

    def __init__(self, m_id, frm, to, date, subject, tag, body):  # DO NOT MODIFY Attributes
        super().__init__(m_id, frm, to, date, subject, tag, body)  # Inherits attributes from parent class DO NOT
MODIFY
        self.body = self.encrypt()  # Overwrite the body with the encrypted version

    # FA.5.b                    - Samira Uzturk ( SID: )
    def encrypt(self):
        """Encrypts the email message body based on the specified rules:
        1. Each letter becomes its numeric position in the alphabet (A=1, B=2, etc.)
          multiplied by the number of words in the original message body.
        2. Full stops (.) remain unchanged.
        3. Numbers (0-9) are replaced by their corresponding alphabet letters (0=j, 1=a, etc.).
        :rtype: str (the encrypted message body) """

        alphabet = "abcdefghijklmnopqrstuvwxyz"
        number_mapping = "jabcdefghi"  # 0=j, 1=a, 2=b, ... 9=i

        # 1. Calculate the number of words in the original body
        # We replace dots with empty strings to ensure "word." counts as "word"
        words = self.body.replace(".", " ").split()
        count_word = len(words)

        # Initialize the list for the encrypted characters
        encrypted_chars = []

        # Process the body character by character
        for char in self.body:
            char_lower = char.lower()

            # Indentation fixed here: The IF block must be INSIDE the FOR loop
            if char_lower in alphabet:
                # Rule 1: Letter position * word count
                # Python index is 0-based, so we add 1 to make A=1, B=2
                position = alphabet.index(char_lower) + 1
                encrypt_value = str(count_word * position)
                encrypted_chars.append(encrypt_value)
```

```
        elif char == '.':
            # Rule 2: Full stops remain unchanged
            encrypted_chars.append(char)

        elif char.isdigit():
            # Rule 3: Numbers are replaced by their corresponding alphabet letters
            index = int(char)
            # Fixed syntax: Use square brackets [] for string indexing, not ()
            encrypted_chars.append(number_mapping[index])

        else:
            # Appends spaces and other punctuation unchanged
            encrypted_chars.append(char)

    # Join the characters back into the encrypted string
    return "".join(encrypted_chars)

# FA.5.c                    - Samira Ozturk (SID: )
def show_mail(self):
    """Overrides the Mail show_mail() method to display confidential emails
    in the specified format showing the encrypted body.
    """
    print("\nCONFIDENTIAL")
    print(f"From: {self.frm}")
    print(f"Date: {self.date}")
    print(f"Subject: {self.subject}")
    print(f"Encrypted body message: {self.body}")
    print(f"Flagged? {self.flag}")
```

4. Personal.py:

```
#########################################################################################################
#######
### COMP1811 - CW1 Outlook Simulator                                    ###
###         Personal Class                                  ###
###         <describe the purpose and overall functionality of the class defined here>    ###
### Partner B:                                              ###
###         <Tahseen Taj>, SID<001494074>                    ###
#########################################################################################################
#######

# DO NOT CHANGE CLASS OR METHOD NAMES/SIGNATURES
# replace "pass" with your own code as specified in the CW spec.

from Mail import Mail

# FB.5.a
class Personal(Mail):
```

```
    """Only contain personal mails with same output format containing frm,to,date,subject,body,flag,read"""
    # DO NOT CHANGE CLASS NAME OR METHOD NAMES/SIGNATURES
    # Add new method(s) as required in CW spec
    def __init__(self, m_id, frm, to, date, subject, tag, body):  # DO NOT MODIFY Attributes
        super().__init__(m_id, frm, to, date, subject, tag, body)  # Inherits attributes from parent class DO NOT
MODIFY

        #Automatically set the tag to the designated folder for Personal emails
        self.tag = "Personal"


        self.body = self.add_stats(body)  #Call the statistics method



    # FB.5.b                          - Tahseen taj (SID: 001494074)

    #
    def add_stats(self,body):
        """The "Personal" email type simulates handling private emails. Personal emails have the same
        characteristics, attributes and behaviours as any other email, but are maintained in a special folder """

        # --- Part 1: Replace 'Body' with Sender's UID ---
        # 1. Extract the Sender's UID (text before "@")

        uid = self.frm.split("@")[0]

        modified_body = self.body.replace(self.body,uid,1)

        # 4. Calculate Statistics (Ensure you handle empty word list to avoid division by zero)
        word = body.split()
        word_count = len(word)

        if word_count > 0:
            total_length = sum(len(word) for word in word)
            avg_length = round(total_length / word_count, 2)
            longest_length = max(len(word) for word in word)
        else:
            avg_length = 0
            longest_length = 0

        # 5. Format the statistics string as required [cite: 347, 348]
        stats_string = (
            f"Stats: Word count:{word_count}, "
            f"Average word length:{avg_length}, "
            f"Longest word length:{longest_length}."
        )

        # 6. Return the modified body concatenated with the statistics
```

```
        return modified_body + " " + stats_string

    def show_mail(self):
        #executing the printing like the given format
        print("PERSONAL")
        print(f"From:{self.frm}")
        print(f"To:{self.to}")
        print(f"Date:{self._date}")
        print(f"Subject:{self.subject}")
        print(f"Body:{self.body}")
        print(f"Read?{self.read}")
```

5. Interpreter.py :

```
#########################################################################################################
#######
### COMP1811 - CW1 Outlook Simulator                                    ###
###         Interpreter program                                       ###
###         Used to as the main program that program will manage all OutlookSim operations ###
###         automatically in response to user commands via an interactive command-line     ###
###         interface. The interpreter represents the user interacting with their mailbox. ###
### Partner A:                                                        ###
###         <Tahseen Taj>, SID<001494074>                             ###
### Partner B:                                                        ###
###         <Samira Ozturk>, SID<student ID>                          ###
#########################################################################################################
#######

# DO NOT CHANGE FUNCTION NAMES
# replace "pass" with your own code as specified in the CW spec.

from MailBoxAgent import MailboxAgent
import random, string


# gen_bdy Generates random text for the email body
# DO NOT MODIFY
def gen_bdy():
    """ generates email body message
        :rtype: string """
    snt = ''
    for i in range(random.randint(1, 10)):
        snt += ''.join(random.choices(string.ascii_lowercase, k=random.randint(3, 10))) + ' '
    return f"Body{str(random.randint(0, 140))}. {snt.capitalize()[:-1]}."


# gen_msg generates a string of emails separated by "----"
#    Used to simulate emails in Outlook mailboxes
#    The output is a string of emails that should be used in your code as required in the CW spec.
```

```
# DO NOT MODIFY
def gen_emails():
    """ generates list of email strings
        :rtype: list """
    msgs, msg_id = [], 0
    for i in range(40):  # sent 40 email
        msg = ''
        for j in range(30):  # to 30 destinations each
            msg += f"ID:{str(msg_id)}" + "\n"
            msg += f"From:email{random.randint(0, 15)}@gre.ac.uk\n"
            msg += f"To:email{random.randint(0, 80)}@gre.ac.uk\n"
            msg += f"Date:{random.randint(1, 29)}/{random.randint(0, 12)}/2025\n"
            msg += f"Subject:subject{random.randint(0, 100)}\n"
            msg += f"Tag:tag{random.randint(0, 6)}\n"
            msg += f"Body:{gen_bdy()}\n"
            msg += "Flag:False\n"
            msg += "Read:False\n"
        msgs.append(msg)
        msg_id += 1
    return msgs


# DO NOT MODIFY
def display_command_help():  # DO NOT MODIFY (used in loop function)
    """ Displays command line help """
    print('Interpreter Commands:')
    print('get <m_id> | ',  # A.1&2 Command to get and display email given email ID - e.g. get 10
          'lst | ',  # B.1 Display entire mailbox - e.g. lst
          'mv <m_id> <tag> | ',
          # B.2 Move email with given ID to folder indicated in given tag - e.g. mv 10 conf (i.e. change current tag
to conf), then display that email
          'del <m_id> | ',
          # A.3 Delete email with given ID by moving to bin - e.g. del 10 (i.e. change current tag to bin), then
display that email
          'mrkr <m_id> | ',  # B.3 Mark email with given ID as Read then display that email
          'mrkf <m_id> | ',  # B.3 Mark email with given ID as Flagged then display that email
          'flt <frm> | ',  # A.4 Filter and display all emails from a given name/email address - e.g. flt email13
          'fnd <date> | ',  # B.4 Find and display all emails received on a given date - e.g. fnd 12/3/2025
          'add <email>')  # A.5&6 and B.5&6 simulate send email by adding emails to the mailbox
    # example add prompts:
    # add email1223@gre.ac.uk email723@gre.ac.uk 29/5/2025 subject99 conf %%Body99911. Isfeo afwco
sxzmp.
    # add email142@gre.ac.uk email788@gre.ac.uk 29/5/2025 subject88 prsnl %%Body11445. Isfffffeo afffwco
sxzmp.
    # add email116@gre.ac.uk email142@gre.ac.uk 29/5/2025 subject36 tag1 %%Body:Body68. Wods vmm
tskgdrxzrk.
```

```
# loop repeatedly asks for command input until 'end' is entered
# DO NOT MODIFY FUNCTION NAME
# - Replace 'pass' with the code necessary to call class/methods relevant for each command
# - Completed as a group                          - 001494074
def loop():
    mba = MailboxAgent(gen_emails())  # mba is an instance of the MailboxAgent class DO NOT MODIFY
    display_command_help()  # simply display the interpreter command-line commands as help
    line = input('mba > ')  # displays a command-line prompter for users to enter command script
    words = line.split(' ')  # separates the command from the script arguments
    command, args = words[0], words[1:]  # command is one of the interpreter script commands outlined in the
help above
    # args is a list of arguments each command may take.
    while command != 'end':
        match command:
            # Partners A and B
            # Replace each pass statement below with a call to the relevant mba methods as described in the CW
spec
            # FA/B.6                          - Tahseen Taj (
            case 'add':
                # example command prompt:
                # add email1223@gre.ac.uk email723@gre.ac.uk 29/5/2025 subject99 conf %%Body99911. Isfeo
afwco sxzmp.
                # add email142@gre.ac.uk email788@gre.ac.uk 29/5/2025 subject88 prsnl %%Body11332. Isfffffeo
sxzmp.
                # add email116@gre.ac.uk email142@gre.ac.uk 29/5/2025 subject36 tag1 %%Body:Body68. Wods
vmm tskgdrxzrk.
                if len(args) >= 6:
                    # The body can contain spaces, so join all elements from index 5 onwards.
                    body_message = ' '.join(args[5:])

                    # Note: You may need to clean/split the initial data further based on your specific 'gen_emails()'
                    # format, but this structure handles the required parameters.
                    mba.add_email(args[0], args[1], args[2], args[3], args[4], body_message)
                else:
                    print("Error: 'add' command requires at least 6 arguments (frm, to, date, subject, tag, body).")

            case 'del':  # move email with given ID to bin folder
                # example command prompt:
                # del 10
                if len(args) == 1 and args[0].isdigit():
                    mba.del_eadmail(int(args[0]))
                else:
                    print("Error: 'del' command requires a single valid email ID.")
            case 'flt':
                # example command prompt:
                # flt email13
                if len(args) == 1:
                    result_list = mba.filter(args[0])
```

```
            # Assuming you have a helper to print the list of mails, or show_emails is reused.
            print(f"Filter found {len(result_list)} email(s) from {args[0]}.")
            # NOTE: You will need to print the result list here.
        else:
            print("Error: 'flt' command requires a single sender email address.")
    case 'fnd':
        # example command prompt:
        # fnd 12/3/2025
        if len(args) == 1:
            result_list = mba.find(args[0])
            print(f"Find found {len(result_list)} email(s) on {args[0]}.")
            # NOTE: You will need to print the result list here.
        else:
            print("Error: 'fnd' command requires a single date string.")
    case 'get':  # retrieve and display email Mail object given email ID
        # example command prompt:
        # get 10
        if len(args) == 1 and args[0].isdigit():
            email_object = mba.get_email(int(args[0]))
            if email_object:
                # Assuming the Mail object has a polymorphic show_email method
                email_object.show_email()
            else:
                print(f"Error: Email ID {args[0]} not found.")
        else:
            print("Error: 'get' command requires a single valid email ID.")
    case 'lst':  # display entire mailbox
        # example command prompt:
        # lst
        mba.show_emails()
    case 'mrkr':
        # example command prompt:
        # mrkr 10
        if len(args) == 1 and args[0].isdigit():
            # Call mark function with 'read' as the mark_type
            mba.mark(int(args[0]), 'read')
        else:
            print("Error: 'mrkr' command requires a single valid email ID.")
    case 'mrkf':
        # example command prompt:
        # mrkf 10
        if len(args) == 1 and args[0].isdigit():
            # Call mark function with 'flagged' as the mark_type
            mba.mark(int(args[0]), 'flagged')
        else:
            print("Error: 'mrkf' command requires a single valid email ID.")
    case 'mv':  # move email with given ID to folder in given tag
        # example command prompt:
```

```
            # mv 10 conf
            # Arguments: m_id, tag
            if len(args) == 2 and args[0].isdigit():
                mba.mv_email(int(args[0]), args[1])
            else:
                print("Error: 'mv' command requires ID and destination tag.")

        line = input('mba > ')
        words = line.split(' ')
        command, args = words[0], words[1:]


if __name__ == '__main__':
    loop()
```