0.1 Written Responses

0.1.1 Program Purpose and Devopment

0.2 2a.

The purpose of the Program is to allow Users to create notes on the command line. Every time a user decides to take a note they must decide where to save it and what to call it. The goal is to reduce the obstacles stopping a note taker.

ii. The video demonstrates the creation of a note, searching for notes, and deleting a note.

iii. The inputs are the arguments passed to the Program from the command line.

Note manage create -m "The best program idea ever!" -t "program idea" -t programming

The above command creates a note with a message, "The best program idea ever!" and two tags, "program idea" and programming. The output displays the created note.

Note search tag -t programming -t "program idea" –limit 100

The above command lists up to 100 notes with the following tags, "program idea" and programming. The output is a list of notes.

0.3 3b.

i.

```
@abstractmethod
def _init_database(self):
    """
    Create notes database.
    """
    pass
```

ii.

```
@abstractmethod
def select_note(self) -> List[Note]:
    """
    Select notes from the database.
    """
    pass
```

iii. The name of the collection is SQLite.

iv.

id	content	date	active
1	Best Program idea ever!	15/4/2021	True
2	Hello, World!	15/4/2021	True
3	I need to get and sd card.	15/4/2021	True
4	Work on homework tonight.	15/4/2021	False

fk_note_id	name
1	programming
2	test
1	program-idea
4	school

The note table has 4 columns. The id column is the unique identifier for each entry or note in the database. The content column is a binary blob with the content or message of a note. The date column is the date the note was created. The active column is a boolean that represents the removal of a note.

The tag table has 2 columns. The fk_note_id and name make up the unique identifier for each tag. The fk_note_id column is a foreign key with the id of a note. The name column is the tag.

V. As the amount of data increases the difficulty of handling all the data spikes with simple structures like arrays or lists. A RDBMS abstracts the complexity away. I could have used JSON or manage physical files. It's not easy editing existing JSON. Physical files are slow. Defeating the purpose of the Program.

0.4 3c.

i. The first program code segment.

```
def get_message_from_editor(selected_editor: str) -> str:
    """
    Launch editor and get a note message.
    """

    logger.info(f"Starting {selected_editor}")

    for editor in Editor.editors:
        if editor == selected_editor:

            process = subprocess.run(" ".join(editor.args),
                shell=True)

            if process.returncode:
                raise Exception(f"{editor} not installed!")

            return _read_message()

    raise NotImplementedError(f"editor: {selected_editor} not
        implemented.")
```

ii. The seconed program code segment. This code snipet calls the first.

```
@app.command()
def create(
        message: Optional[str] = typer.Option(
            None, "-m", "--message", show_default=False,
                help="Message to add to the database."),
        tags: Optional[List[str]] = typer.Option(
            None, "-t", "--tags", show_default=False, help="Tags to
                organize message.", callback=_format_tags_callback),
        editor: EditorChoice = typer.Option(
            "vim", show_choices=True, help="Supported editors.")):
    """
    Insert note into the database.
    """
```

```
    message = message if message else get_message_from_editor(editor)

    logger.info(f"Note message: \n{message}")
    logger.info(f"Note tags: {tags}")
```

iii. The procedure launches one of two command line editors, where the user can write their message. Then returns the message.

iv. The procedure is as follows.

- Log the currently selected editor.
- Iterate through the list of valid editors.
- Check if the editor matches the selected editor.
  - Launch the editor in the shell.
  - If editor is not installed raise an exception
  - Return the message written in the editor.
- Raise an exception If no match is found within the valid editors.

0.5 3d.

i. The first call.

```
message = get_message_from_editor("vim")
print(message)
```

The seconed call.

```
message = get_message_from_editor("fakeeditor")
print(message)
```

ii. The first call works as intended. It will launch vim, a supported editor. Then return everything that was typed. The procedure finds a valid editor, the editor was installed. So it returns the message typed in the editor as a string.

The second call, calls an unsupported editor. It raises an error letting the user know that they will need to select a different editor. The procedure could not find a valid editor and raised an exception.

iii. The following are the results of the procedure.

The result of the first call is a string message prented on the console.

The result of the seconed call is an error.

NotImplementedError: editor: fakeeditor not implemented.