If the below README is difficult to navigate because of its length, please find "README Navigate with Headings.docx" in this directory.

# EXECUTEABLE SETUP

Please note

The executable only works for macOS. I can confirm that it works with M1 chip;

however, it has not been tested on any other chip architectures.

1, Download the zip file "Executable for Mac - Products Tracker".

2, Double-click on app.exe.

3, Access will likely be denied.

4, Within Security and Privacy in Mac settings, please allow third-party developers by selecting "Open Anyway".

5, If the app opens, please see the README.txt within the source code folder.

6, If the app does not open, please see "Manual Installation" within the "source code" folder.

7, If manual installation fails, please see the "Screenshots" folder within the "source code" folder.

# MANUAL SETUP

Hopefully, the zipped executable worked for you. If not, the good news is that this project can be manually set up within three steps:

1. Install Python and the package manager pip.
2. Run pip install -r requirements.txt or pip3 install -r requirements.txt
3. From the products directory, run **python view/app.py**.

# FUTURE EXTENDABILITY

Every design decision focused on how this application can be "extended in the future."

After implementing the below design decisions, it made sense to implement the sorting features because the project was optimized for this feature and its implementation required a small amount of code. Below is a summary of how I built the project so it can be extended in the future both in terms of new features and database integration and scalability. Design decision summary; please read below for details on each of the following:

1. ARCHITECTURE CHOICE
2. FUTURE EXTENDABILITY BY LAYER SUMMARY
3. SQL DATABASE AGNOSTIC
4. SORTING IMPLEMENTATION
5. IDENTITY MANAGEMENT
6. VALIDATION
7. OOP

# 1 ARCHITECTURE CHOICE

I chose not to use a four-layer architecture because I believed it would unnecessarily complicate the application. However, I also avoided using a two-layer architecture because I didn't think it would adequately separate future business logic from the rest of the application. Under the presumption that future extendibility would likely utilize business logic.

Instead, I opted for the 3 layer architecture because I strongly believed that separating the database, business logic, and GUI concerns would result in maximizing loose coupling. This would make the application more modular and easier to maintain and extend in the future.

## Model-View-Controller (MVC) architecture:

## Requirements of each layer:

### View:

* Renders the user interface.
* Includes client-side validation.
* Handles user input.

### Controller:

* Handles user input and business logic.
* Coordinates interactions between the view and model layers.
* Includes server-side validation.

Model:

* Manages the application's data and business logic.
* Includes the database layer for data storage and retrieval.
* Handles database interactions and data management.

## File structure implementing each MVC layer:

### View:

* app
* client\_side\_validation
* GUI/ui\_mainwindow.ui used when manually installing package.
* GUI/ui\_mainwindow.py used in executable.

### Controller: handles user input and business logic and the model:

* read\_controller.py
* write\_controller.py
* server\_side\_validation.py

### Database: data management logic:

* db\_connection.py
* query\_db.py
* write\_to\_db.py
* products.db
* create\_database.py void because database already created.

# 2 FUTURE EXTENDABILITY BY LAYER SUMMARY

Explanation of how each layer can easily be extended in a meaningful way in the future. This is because each layer was designed with loose coupling in mind. The below examples go beyond the standard benefits of using a layered system and are specific to future extensions in mind.

VIEW**:**

**app.py** renders a UI file which means different GUIs can easily be plugged into the MainWindow in anticipation of future UI changes. For example:

uic.loadUi('view/GUI/mainwindow.ui', self)

uic.loadUi('view/GUI/ui\_mainwindow.py', self)

 Are different UI files and different file types, which means entirely different UI file formats can easily be plugged in. Furthermore, a simple if statement could change which UI file is loaded depending on the user's system or other variables.

CONTROLLER**:**

read\_controller.py and wrtie\_controller.py act as the intermediary between the view and model layers. The primary reason I created them is to anticipate future extensions that would require new business logic. This way, business logic can be loosely coupled from the view and model layers.

## MODEL

Beyond the basic separation of model from view and controller, within the model there is additional loose coupling in which only 1 file needs to be changed to connect to a different database externally or internally. This means not only is each layer loosely coupled but even within the model layer there is loose coupling within.

For more details, please see below 3 SQL DATABASE AGNOSTIC.

# 3 SQL DATABASE AGNOSTIC

Only the db\_connection.py within the model layer needs refactoring to achieve this. For example, currently the project is connected to an internal database however the model layer can be changed to easily connect to an external database. Changes are independent of other layers thus the controller and view layers maintain high cohesion. Furthermore, within the model layer concerns are also loosely coupled only the db\_connection.py would need to be changed to connect to an external database because query\_db.py and write\_to\_db.py are designed to be SQL platform-agnostic. Although query\_db.py and write\_to\_db.py provide SQL code queries they are executed within db\_connection.py. And a change to db\_connection.py does not affect these files from delivering SQL Strings to be executed within db\_connection.py. One caveat being if db\_connection.py is refactored to connect to a NoSQL database then query\_db.py and write\_to\_db.py would need refactoring to provide no SQL commands however such refactoring would still be isolated to model layer. My primary goal in building this project was to optimize for future extendibility.

# 4 SORTING IMPLEMENATON

The following sorting features have been built in to the GUI and utilises all MVC layers.

Features

Use by date (ASC),

Date created (ASC),

Name(ASC),

Use by date (DESC),

Date created (DESC),

Name(DESC),

The below code walkthrough shows how the MVC architecture achieves the implementation of new features can easily be added.

## sorting 1,

The below code block implements sorting used by dates in the view layer in app.py method loadTableView

if self.sortByThis == "Use by date (ASC)":

allColumns = self.readController.sortByUseByDate("asc")

elif self.sortByThis == "Date created (ASC)":

allColumns = self.readController.sortByDateCreated("asc")

elif self.sortByThis == "Name(ASC)":

allColumns = self.readController.sortByName("asc")

elif self.sortByThis == "Use by date (DESC)":

allColumns = self.readController.sortByUseByDate("desc")

elif self.sortByThis == "Date created (DESC)":

allColumns = self.readController.sortByDateCreated("desc")

elif self.sortByThis == "Name(DESC)":

allColumns = self.readController.sortByName("desc")

else:

# makes sure table view has values even if sorting failed

allColumns = self.readController.getAllColumns()

print("Sorting Failed")

## sorting 2,

the controller layer, read\_controller.py, class ReadController

handles the communication between the view and model layers for example sortByUseByDate is passed on to the model layer as follows. This would be the best place for future business logic for existing and new methods.

def sortByUseByDate(self, order):

"""

Returns a tuple of all records sorted by use-by date.

Business logic could also be placed here, which sets it apart from the queryDb in the model layer.

Args:

order (str): A string representing the order in which to sort the records. Must be either 'asc' or 'desc'.

Returns:

A tuple of dictionaries representing the sorted records by use by date, with each dictionary containing the following keys:

- id (int): The unique identifier for the record.

- name (str): The name of the product.

- use\_by\_date (str): The use-by date of the product.

- date\_created (str): The date the record was created.

"""

return self.queryDb.sortByUseByDate(order)

## sorting 3,

model layer, query\_db, query\_db.py class queryDb

Next the SQL code is locates the corresponding sort feature.

The write\_to\_db.py and query\_db.py are the ideal place for new SQL methods for the controllers business logic to use for future extensions. This creates a separation of concerns and loose coupling for known and unknown future features.

However no SQL code is run here in order to allow the destination of the SQL database to later be changed without affecting query\_db.py or write\_to\_db.py please see - SQL database agnostic in this read me for more information.

def sortByUseByDate(self, order):

"""

Sorts the products in the database by use-by date in ascending or descending order.

Args:

order (str): The order in which to sort the products. Must be "asc" or "desc".

Returns:

tuple: A tuple of dictionaries representing the sorted products.

Raises:

ValueError: If order is not "asc" or "desc".

"""

try:

if order.lower() == "desc":

return self.db.get("SELECT \* FROM product ORDER BY use\_by\_date DESC")

elif order.lower() == "asc":

return self.db.get("SELECT \* FROM product ORDER BY use\_by\_date;")

else:

raise ValueError("Invalid order value. Must be 'asc' or 'desc'.")

except ValueError as ve:

print("Error: please only provide desc or asc for order", ve)

## sorting 4,

model layer, db\_connection.py, class dbConnection

finally the SQL code in string format from query\_db.py or write\_to\_db.py

arrives at db\_connection.py, class dbConnection

the get methods provides a safe environment for the SQL to be run

for the sorting SQL statements the get methods are used but for

adding and deleting a set method is used because there is no SQL data that

needs to reach the view from the controller layer.

def get(self, query):

"""

Executes a SELECT statement on the database and returns the result.

Args:

query (str): The SELECT statement to execute on the database.

Returns:

The result of the SELECT statement as a tuple.

"""

try:

self.connect()

self.cursor.execute(query)

result = self.cursor.fetchall()

self.close()

return result

except sqlite3.Error as e:

print(f"Error executing query: {e}")

return None

def set(self, sql, data):

"""

Executes an INSERT statement on the database.

Args:

sql (str): The INSERT statement to execute on the database.

data (tuple): The data to be inserted into the database.

Returns:

None.

"""

try:

self.connect()

self.cursor.execute(sql, data)

self.closeCommit()

except sqlite3.Error as e:

print(f"Could not be added to database {e}")

# 5, IDENTITY MANAGEMENT

For each product added a unique record identifier (RID) in the form of a primary key is created

and associated with that product. The SQL database automatically creates and keeps track.

This way items can be duplicates with the same use by dates yet the users can specify which one

to delete.

# 6, VALIDATION

With in view client-side validation and within controller server-side validation files and classes are used. This could be important for future extendibility to ensure entries in the database are consistent and valid for future features.

# 7, OOP

Examples of what I built into my project and how it relates to OOP

1, MVC architecture: Encapsulation, Abstraction, Polymorphism, Separation of concerns.

2, Extendibility by layer: Abstraction, Polymorphism, Loose coupling.

3, SQL database agnostic: Abstraction, Polymorphism, Loose coupling.

4, Sorting implementation: Abstraction, Polymorphism, Loose coupling.

5, Identity management: represent unique objects and manage their relationships

6, Validation: Abstraction, Polymorphism, Data encapsulation.