How to?

We will structure our project in a certain way, the Model-View-Controller (MVC)* architecture. It is very similar to the MVT architecture used by Django. This architecture separates the codebase into:

1. Model: This component represents the data structures and logic of the application. It is responsible for managing the data and performing operations on it. In our project, this component would handle the encryption and decryption of files. For our ease, we will be using Django ORM, ***very powerful ;-).***

2. View: This component represents the user interface of the application. It is responsible for displaying the data from the Model to the user and receiving user input. In our project, this component would handle the user interface of Crypt using Tkinter library.

3. Controller: This component is responsible for coordinating the Model and the View. It receives input from the user via the View, updates the Model accordingly and updates the View with new information from the Model. In our project, this component would handle the interaction between the UI and logic.

We will create separate folders for each component in the main directory. The 'data' folder will contain the database and encryption related codes. The 'logic' folder will contain the encryption and decryption functions. The 'ui' folder will contain the Tkinter related codes.

In order to integrate the UI and logic part, we will create classes in the logic folder, for example, 'FileVault' class and import it in the UI folder and use its methods to perform the operations. The data part can be connected to logic through the use of Django ORM and SQLite3.

We will also create a 'main.py' file in the main directory to run the application.

It is important to test the code regularly to ensure that it is working as expected, we can use python's unittest library to test the code or any other as per need.

*For video explanation on MVC: https://youtu.be/pAHVSpbftYY

By following this structure, it will be easy for us to work on different parts of the application simultaneously and also easy to maintain the codebase.

A Sample for a **To-Do Application** on how we will be following this architecture:

File Structure:

```
todo/
        data/
                db.py
                db.sqlite3
                _iniit_.py
        logic/
                logic.py
                _iniit_.py
        ui/
                ui.py
                _iniit_.py
        test/
        main.py
        env/
        .git/
```

The Logic module is the part where the algorithm of the project will be implemented.

File todo/logic/logic.py

```python
class ToDoList:


    def __init__(self):
        self.tasks = []

    def add_task(self, task):
        self.tasks.append(task)

    def delete_task(self, task):
        self.tasks.remove(task)

    def get_tasks(self):
        return self.tasks
```

In order to connect the logic and UI parts of your application, we will need to import the logic module in the UI module and use it to retrieve and display the necessary data.

File todo/ui/ui.py

```
import tkinter as tk

from logic import ToDoList

class ToDoListUI:
    def __init__(self):
        self.todo = ToDoList()
        self.root = tk.Tk()
        self.root.title("To-Do List")
        self.listbox = tk.Listbox(self.root)
        self.listbox.pack()
        self.task_entry = tk.Entry(self.root)
        self.task_entry.pack()
        self.add_button = tk.Button(self.root, text="Add", command=self.add_task)
        self.add_button.pack()
        self.delete_button = tk.Button(self.root, text="Delete",
command=self.delete_task)
        self.delete_button.pack()
        self.display_tasks()
        self.root.mainloop()

    def add_task(self):
        task = self.task_entry.get()
        self.todo.add_task(task)
        self.display_tasks()

    def delete_task(self):
        task = self.listbox.get(self.listbox.curselection())
        self.todo.delete_task(task)
        self.display_tasks()

    def display_tasks(self):
        self.listbox.delete(0, tk.END)
        for task in self.todo.get_tasks():
            self.listbox.insert(tk.END, task)
```

Here, in the ToDoListUI class imports the ToDoList class from the logic module and creates an instance of it. The UI class then uses the ToDoList instance to add and delete tasks, and to retrieve the list of tasks to display in the UI.
In the main.py, you can import the ui module and create an instance of the ToDoListUI class. This will start the application and display the to-do list UI.

File todo/main.py

```
from ui import ToDoListUI


if __name__ == "__main__":
    ui = ToDoListUI()
```

This is a very basic example, but it should give us an idea of how to connect the logic and UI parts of your application and use data structures from the logic part to display information in the UI.

Django ORM

We will use the Django ORM by creating a small Django project within the data module of the project, as "orm". It's already great the we have a **User** model in Django from the contributors, thanks to them. We will simply import the User model and initialize in the _init_.py file so that we could easily import as:

```
from data import User
```

instead of

```
from django.contrib.auth.models import User
```

With Django ORM, we will be using Object Relational Mapping (ORM) instead of SQL to query data from the database. Instead of:

```
SELECT * from crypt.users WHERE username = "bishnu_boko";
```

We will use:

```
User.objects.filter(username="bishnu_boko")
```