TkinterWiki

Release 0.0.1

JeffCube

Feb 12, 2023

CONTENTS

1 Databases **3**
1.1 sqlite3 3

2 Application Structure **5**
2.1 Model View Controller (MVC) 5

3 Connecting Apps Through the Internet **13**

TkinterWiki is a wiki containing information on various topics concerning Tkinter, Python's standard GUI.

Note: This project is under active development.

CHAPTER

ONE

DATABASES

1.1 sqlite3

1.1.1 Example Database Class

Note: This class does not contain functions for creating, saving, updating information in the database. It only outlines a way to have a class that safely initializes and closes a connection to an sqlite3 database when the class is instantiated and deleted.

```
class Sqlite3DatabaseExample:
    def __init__(self, database_file_path: str):
        """
        Initializes connection to a database file. If the database file does not exist
        it will be created at the given database_file_path location.

        **Note** In general you should not expose the database location for security␣
↪purposes.
        :param database_file_path: Path to a (.db) database file
        """
        self._database_connection: sqlite3.Connection = sqlite3.connect(database_file_
↪path)

    def __del__(self):
        """
        Closes the connection to the database. This gives us peace of mind
        as the database connection will be terminated whenever the class
        is deleted or garbage collected.

        **Note** This function does not call the sqlite3 commit() command
        so any uncommitted changes to the database will be lost.
        """
        self._database_connection.close()
```

CHAPTER

TWO

APPLICATION STRUCTURE

2.1 Model View Controller (MVC)

- *Useful Links*
- *Overview*
  - *Model*
  - *View*
  - *Controller*
- *MVC Structures*
  - *No Connection Between Model and View*
  - *Server Client MVC*
- *How To Handle Increased Complexity*
  - *Overview*
  - *Class Structure For Multiple Windows*

2.1.1 Useful Links

- MVC Explained in 4 Minutes

2.1.2 Overview

MVC is a way to sturcture your applications into components that handle specific aspects of an application. This is similar to clean code principles where a function should do one thing. It would be a mess designing a class that handles both the database of your application as well as the visuals.

Model

- Handles data logic: validating, updating, saving, deleting data.

View

- Handles visualization of the application to the user.

Controller

- Handles user requests.

2.1.3 MVC Structures

No Connection Between Model and View

In this version of MVC, there exists no communication between the model and the view. The controller is the middleman between the model and the view. The controller can retrieve data from the model and deliver it to the view for it to be rendered. The controller also handles events from the view and updates the model accordingly.

Below is a simple MVC appplication showcasing the controller as a middleman:

Example Application: Light Bulb

```
import tkinter as tk
from typing import Callable, Any


class LightBulbModel:
    """Keeps track of the state of the bulb."""
    def __init__(self):
        self.is_on: bool = False


class LightBulbView(tk.Tk):
    """
    Displays the state of the bulb and contains a button that the user can press
    to turn the bulb on and off.
    """

    ON_COLOR = "#FFFF00"
    OFF_COLOR = "#808080"

    def __init__(self):
        super().__init__()

        # Set size of window
        self.geometry("500x500")

        # Creates a Label with text. The background color is used to show the bulb being␣
↪on/off
        self.bulb = tk.Label(self, text="(*---*)", bg=self.OFF_COLOR)
```

(continues on next page)

(continued from previous page)

```
        self.bulb.grid(row=0, column=0)

        # A button for the user to toggle the bulb on and off
        self.switch = tk.Button(self, text="ON / OFF")
        self.switch.grid(row=1, column=0)

    def bind_command_to_switch_button_press(self, command: Callable[[], Any]) -> None:
        """Sets a command that is invoked whenever the user clicks on the switch button""
↪"
        self.switch.config(command=command)

    def set_bulb_state(self, on: bool) -> None:
        """Sets the background color of the label depending on the state of the bulb"""
        if on:
            self.bulb.config(bg=self.ON_COLOR)
        else:
            self.bulb.config(bg=self.OFF_COLOR)


class LightBulbController:
    """
    Handles user input to the application. The controller is the middleman between the
    model and the view. It can access the state of the light bulb from model.is_on and
    updates the view by calling the view.set_bulb_state function.
    """
    def __init__(self, model: LightBulbModel, view: LightBulbView):
        self.model = model
        self.view = view
        # For the controller to handle user input, we link user input callbacks from the␣
↪view
        # to functions in the controller. In this case whenever the user clicks on the␣
↪switch
        # we invokes the controller's flip_switch function.
        self.view.bind_command_to_switch_button_press(self.flip_switch)

    def flip_switch(self) -> None:
        """Updates the model to switch the bulb state and then updates the view"""
        self.model.is_on = not self.model.is_on
        self.view.set_bulb_state(self.model.is_on)

    def run_application(self) -> None:
        self.view.mainloop()


def main():
    light_bulb_model = LightBulbModel()
    light_bulb_view = LightBulbView()
    light_bulb_controller = LightBulbController(light_bulb_model, light_bulb_view)
    light_bulb_controller.run_application()


if __name__ == '__main__':
```

(continues on next page)

(continued from previous page)

```
    main()
```

Server Client MVC

When you start to deal with applications in a client-server setting. You may opt for keeping the model and controller on the server side while the view is on the client side. See Below for an example

2.1.4 How To Handle Increased Complexity

Overview

Lets say that your view creates windows and those windows can create even more windows. You then have to answer questions like: Should I put all of this view functionality in a single class? How do events in child windows connect to the controller?

Hopefully some of these questions can be answered here.

Class Structure For Multiple Windows

In a simple application, we may be able to get away with placing the model, view, and controller into only 3 classes. However for more complex applications placing all of this functionality whitin only 3 classes makes the classes bloated and difficult to develop / debug.

To keep our classes clean and simple, we look to the concept of Feature Envy. Feature envy is when methods of a class have nothing to do with variables or functions of the class they belong to. If we have the prospect of multiple windows where each window has a different purpose, it may be wise to create a separate controller class and view class for each window (see Stack Overflow discussion).

In a multi window application one view may be responsible for displaying the model state while another view changes the state of the model. To keep up to date with the changes to the model, we could use the observer pattern. Using this pattern, the controller subscribes to the model and the model sends out a notification whenever it is updated.

Note: If keeping a view up to date with the model not a high priority you could instead update the view at regular intervals.

Below is a MVC application that handles multiple controllers, multiple views, and a model that handles update events:

Example Application: Shopping List

```
import tkinter as tk
from tkinter import messagebox
from typing import List, Callable, Any


def show_error(title: str, message: str):
    messagebox.showerror(title=title, message=message)


class ShoppingListModel:
    """
```

(continues on next page)

(continued from previous page)

```
    Keeps track of a list of items.
    The class can store and remove functions inside _update_commands.
    When the list is modified the model triggers an event that calls each function␣
↪stored in _update_commands.
    This is useful when a function needs to be invoked whenever the shopping lists␣
↪updates.
    """
    _update_commands: List[Callable[[], Any]] = []

    def __init__(self):
        self.list: List[str] = []

    def add_item(self, item: str) -> None:
        self.list.append(item)
        self.list_has_been_updated()

    def remove_item(self, item_index: int) -> None:
        self.list.pop(item_index)
        self.list_has_been_updated()

    def list_has_been_updated(self) -> None:
        """
        Calls all commands stored in self._update_commands
        This function should be called whenever the shopping list is updated.
        """
        for command in self._update_commands:
            command()

    def bind_command_to_update_event(self, command: Callable[[], Any]):
        self._update_commands.append(command)

    def remove_command_from_update_event(self, command: Callable[[], Any]):
        self._update_commands.remove(command)


class AddItemWindow(tk.Toplevel):
    """
    A window the user uses to add items to the shopping list. It contains a text field␣
↪for the user
    to input an item name and a button they use to submit the name to the shopping list.
    """
    def __init__(self, *args, **kwargs):
        super().__init__(*args, **kwargs)
        self.item_entry = tk.Entry(self)
        self.item_entry.grid(row=0, column=0)
        self.submit_button = tk.Button(self, text="Submit")
        self.submit_button.grid(row=0, column=1)
        # Force the user to only interact with this window. No other windows can be␣
↪interacted with
        # until this window is closed (window is closed when user presses the "Submit"␣
↪button.
        self.grab_set()
```

(continues on next page)

(continued from previous page)

```
    def get_item(self) -> str:
        return self.item_entry.get()

    def bind_command_to_submit_button_press(self, command: Callable[[], Any]) -> None:
        self.submit_button.config(command=command)


class AddItemWindowController:
    """
    Handles events from the AddItemWindow.
    Can add items to the shopping list model. Once an item is added this controller␣
↪closes the AddItemWindow.
    """
    def __init__(self, model: ShoppingListModel, view: AddItemWindow):
        self.model = model
        self.view = view
        self.view.bind_command_to_submit_button_press(self.add_item)

    def add_item(self) -> None:
        """
        If the user as entered a non empty item string, add it to the shopping list.␣
↪Otherwise
        show a error message to the user.
        """
        item = self.view.get_item()
        if item:
            self.model.add_item(item)
            self.view.destroy()
        else:
            show_error(title="Invalid Item", message="Please add a item to the text box.
↪")


class ShoppingListView(tk.Tk):
    """
    Allows the user to see the list of items in the shopping list. Includes buttons for␣
↪adding
    items to the list and removing selected items.
    """
    def __init__(self):
        super().__init__()
        self.shopping_listbox = tk.Listbox(self)
        self.shopping_listbox.grid(row=0, column=0, columnspan=2)
        self.add_item_button = tk.Button(self, text="Add Item")
        self.add_item_button.grid(row=1, column=0)
        self.remove_item_button = tk.Button(self, text="Remove Item")
        self.remove_item_button.grid(row=1, column=1)

    def bind_commands_to_view_events(self,
                                     add_item_command: Callable[[], Any],
                                     remove_item_command: Callable[[], Any],
```

(continues on next page)

(continued from previous page)

```
                                       close_view_command: Callable[[], Any]) -> None:
        self.add_item_button.config(command=add_item_command)
        self.remove_item_button.config(command=remove_item_command)
        # This command is called when the user closes the window
        self.protocol("WM_DELETE_WINDOW", close_view_command)

    def get_selected_item_index(self) -> int:
        """
        Gets the index of the shopping item selected by the user. The user selects items␣
↪when
        clicking on them inside the shopping_listbox.
        """
        selection = self.shopping_listbox.curselection()
        if selection:
            return selection[0]
        else:
            return -1

    def open_add_item_window(self) -> AddItemWindow:
        return AddItemWindow(self)

    def refresh_list(self, shopping_list: List[str]) -> None:
        """Clears all items from the shopping_listbox and repopulates it with the␣
↪shopping_list"""
        self.shopping_listbox.delete(0, tk.END)
        for item in shopping_list:
            self.shopping_listbox.insert(tk.END, item)


class ShoppingListController:
    """
    Handles events from the ShoppingListView.
    Can remove items from the shopping list model and spawns a AddItemWindow with its␣
↪AddItemWindowController when
    the user wants to add an item to the list. The contoller also updates the list␣
↪displayed to the user whenever
    the model is updated.
    """
    def __init__(self, model: ShoppingListModel, view: ShoppingListView):
        self.model = model
        # Sets self.update_list to be called whenever the model is updated
        self.model.bind_command_to_update_event(self.update_list)

        self.view = view
        self.view.bind_commands_to_view_events(add_item_command=self.open_add_item_
↪window,
                                               remove_item_command=self.remove_item,
                                               close_view_command=self.close_application)

    def open_add_item_window(self) -> None:
        """
        Instructs the view to create an add item window and creates a controller for the␣
↪window.
```

(continues on next page)

(continued from previous page)

```
        """
        add_item_window = self.view.open_add_item_window()
        AddItemWindowController(self.model, add_item_window)

    def remove_item(self) -> None:
        """
        Gets the selected item index from the view and removes the item from the␣
↪shopping list model.
        If no item is selected show an error message to the user.
        """
        index = self.view.get_selected_item_index()
        if index == -1:
            show_error(title="Invalid Selection", message="Select an item from the list␣
↪to remove")
        else:
            self.model.remove_item(index)

    def update_list(self) -> None:
        """
        This method should be called whenever the shopping list is updated. We supply␣
↪the list
        of items to the view so that it can refresh the shopping list displayed to the␣
↪user
        """
        self.view.refresh_list(self.model.list)

    def run_application(self) -> None:
        self.view.mainloop()

    def close_application(self) -> None:
        """
        When the user closes the window we must remove our command reference from the␣
↪model.
        Afterwards we destroy the application window.
        """
        self.model.remove_command_from_update_event(self.update_list)
        self.view.destroy()


def main():
    shopping_list_model = ShoppingListModel()
    shopping_list_view = ShoppingListView()
    shopping_list_controller = ShoppingListController(shopping_list_model, shopping_list_
↪view)
    shopping_list_controller.run_application()


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

CHAPTER

THREE

CONNECTING APPS THROUGH THE INTERNET