Faculty of Engineering and Technology
Electrical and Computer Engineering Department

Lunix Laboratory
ENCS3130

Project II
Automative programming

Prepared by:
Abdel Rahman Shahen 1211753

Instructor: Aziz Qaroush
Assistants: Eng. Ahed Mafarjeh

Section: 4
Date: 1/27/2024

Libraries

subprocess: Execute and interact with external processes, commands, and programs.

xml.etree.ElementTree (ET): Parse and manipulate XML documents.

re: Perform regular expression operations for pattern matching.

xml.dom.minidom: Process XML documents using a minimal DOM implementation.

os: Interface with the operating system for file and directory operations.

collections.Counter: Count and tally elements in a collection.

pathlib.Path: Object-oriented filesystem path handling.

tkinter (Tk): Create graphical user interfaces (GUIs) in Python.

tkinter.Canvas: Draw graphics and place objects in a Tkinter window.

tkinter.Entry: Input field widget for Tkinter GUIs.

tkinter.Text: Multiline text widget for Tkinter GUIs.

tkinter.Button: Clickable button widget for Tkinter GUIs.

tkinter.PhotoImage: Display images in Tkinter applications.

tkinter: Python interface for creating GUIs using the Tk toolkit.

tkinter.font: Manage fonts in Tkinter applications.

Importlib: is a Python module for importing and managing Python packages and modules.

threading: threads to make the program faster. And prevent the program from crashing.

Classes

`CommandManual:`

This CommandManual class is like a guidebook creator for Python commands. Imagine you have a command, like a magic spell, and you want to make a detailed guide about it. This class helps you gather all the important details and write them down in a special computer-readable format called XML, which can be used by other programs or displayed on websites.

Here's a simple explanation of what each part does:

Class functions:

- When you create a new CommandManual, you tell it which Python command you're making a guide for, like 'print' or 'len'.

- The fetch_command_description function is like asking a Python expert (using a program called pydoc) to explain what the command does, and then writing down that explanation. This fetching operation is done by just taking the whole pydoc using subprocess.Popen executing. Python -m pydoc command and passing the output to the result.

- The fetch_version_history function finds which version of Python command. If the command is the name of a top-level package it brings the version using importlib.metadata.version(self.command). if the package/command isn't found use subprocess.run to get the version of the current python.

- With fetch_examples_from_file, it's like having an online documentation that includes commands, their description and their examples. This makes the program more reliable as the online documentation can vary along the time which will cause the change on the structure of the documentation.

- fetch_related_commands is like finding friends for your command. It looks through Python's documentation to find other commands that are similar or related to the one you're making a guide for. The program has a set of known commands, it looks for them in the pydoc of the command we need related commands for. and if any commands found. They will be the related commands.

- The set_syntax_and_usage function figures out the correct way to write and use the command by examining the instruction manual (pydoc) closely. The examining process is like this: the code goes through pydoc using subprocess.Popen(["python", "-m", "pydoc", self.command], stdout=subprocess.PIPE), then it uses regular expression to extract lines matching the specific patterns. Some constant texts will be in pydoc. These will be used by re.compile to find. Then when ever found in the pydoc they will be fetched and saved.

- set_documentation_link is for when you want to include a link to the online manual in your guide, so people can click to get more information. It classifies each command with a valid link to find information their.

- Finally, generate_xml_manual takes all the information you've collected and writes it into an XML file. This file is like a neat, organized booklet about your command, ready to be used by others.

CommandManualGenerator

The CommandManualGenerator class is made to generate documentation manuals for Python commands. Here's a simple explanation of this class and its functions:

`read_commands` function opens the file and reads all the Python commands listed in it. And insert all of the commands in the list “commands” so it can be used later in the code.

`generate_all_manuals` is where the main action happens. The function sets up a separate mini-worker (a thread) to do the job. This is like hiring someone to cook the recipes you've listed, while you continue doing other things. This way, your kitchen (the program) doesn't get crowded, and you (the user interface) can still move around freely and do other tasks.

Inside theses threads, for each command from the list, it creates a detailed manual. This includes gathering information about what the command does, how to use it, examples, etc.

Main Functions

compile_manuals_to_xml() is like creating a big album from several smaller photo albums. Each small album (_manual.xml file) has pictures (information) about a specific python command. This function takes all those pictures and puts them into one big album (all_manuals.xml), so you have a complete collection in one place. This process create a root and take all files and insert their data into that root

normalize_xml_element(element) is similar to having a box with different shapes inside, and you want to organize them neatly. This function sorts those shapes (sub-elements in an XML element) and cleans them up (removes extra spaces and line breaks) so everything looks tidy and is easy to compare.

normalize_text(text) is like taking a sentence and breaking it down into individual words, ignoring all punctuation and making all words lowercase. It's a way to simplify text for easy comparison. This uses regular excerptions.

compare_texts(text1, text2) compares two files to see what items are missing in one file compared to the other. Its very simple. It just subtracts the texts and the difference is the result.

extract_related_commands(xml_file): function goes through the XML file and for each command, it lists related commands.

get_command_recommendations(command_name, related_commands_data) just gets the related commands generated before.

print_command_file(command) looks for a specific command's manual file and shows it to you. If the command isn't there, it tells you that it couldn't find it. It uses os library to find the file then print it.

print_lines_containing_command(command, all_manuals_file) is like searching through a big book for any mention of a specific person's name (command). Whenever the name appears, it shows you the whole sentence where it was found.

perform_search(command) combines the last two functions. It first shows you the specific command manual and then searches through the all manuals file for any mentions of that command.

read_commands(): reads the commands that will be used later.

view_manual(): print the manual of the manual entered

verify_command() is a bit like double-checking two versions of a document but doing it in small parts. It looks at small sections of each document at a time and updates you on the progress, telling you all the differences it found or if it didn’t find any. This done by splitting the files as chunks for faster verification.

search_commands() allows you to type in what command you're looking for and then searches through all the manuals using perform_search(command).

fetch_and_show_recommendations(): If you tell it a specific command you're interested in, this function acts like an assistant, giving you recommendations on other commands that might be useful or related to your interest just by fetching the related commands in the manuals.

UI/UX

Firstly, the code finds out where the script is located on the file system. Then constructs a path to a specific directory or resource that is part of my project. This is to access the assets files.

Her we have relative_to_assets:

This function is used to get the full path of a file that is in assets file.

window = Tk(): Initializes the main window for the UI.

custom_font = font.Font(size=18): Creates a custom font style to be used in the UI.

window.geometry("605x771"): Sets the size of the window.

window.configure(bg = "#FAEAF0"): Sets the background color of the window.

A Canvas widget is created as the primary area within the window where other elements (like buttons, text entries) are placed. It is set to the same size as the window and has a custom background color.

Multiple Button elements are created, each associated with different images and placed at specific coordinates within the canvas. Each button is linked to a specific function (command=lambda:) that defines what it does when clicked. For example, button_1 is linked to the function CommandManualGenerator.generate_all_manuals(None).

Input textboxes are used for user input. They are placed on the canvas. These input fields are used to receive input from the user, such as command names for searching or recommendations.

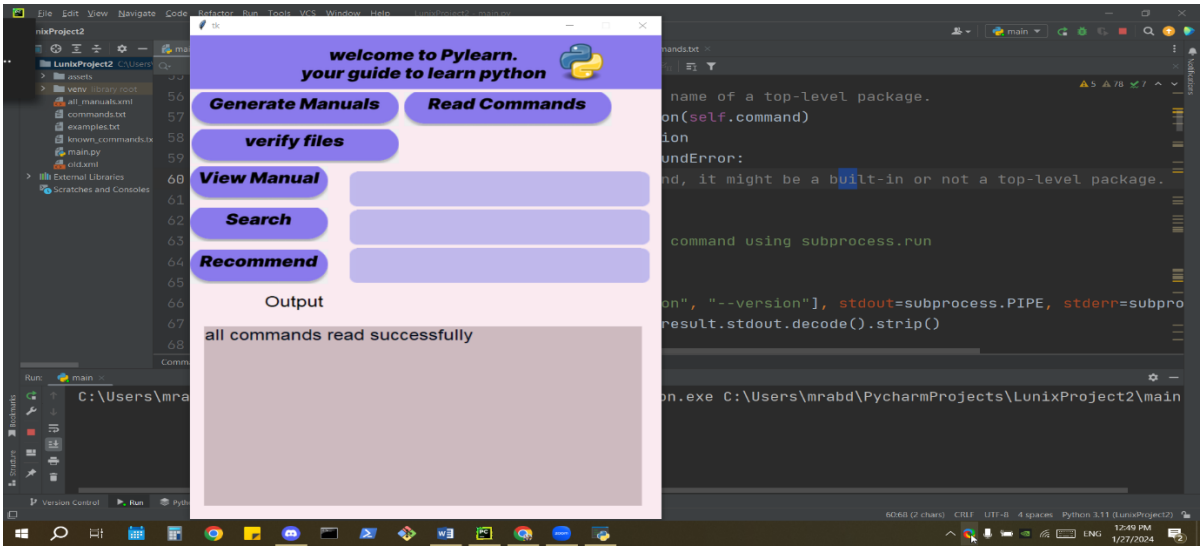
A TextArea is created as an area to display output messages or results.

labels and images are placed on the canvas to make better user experience.

The window is set to be non-resizable.

window.mainloop() starts the Tkinter loop, which waits for user interaction and updates the UI.

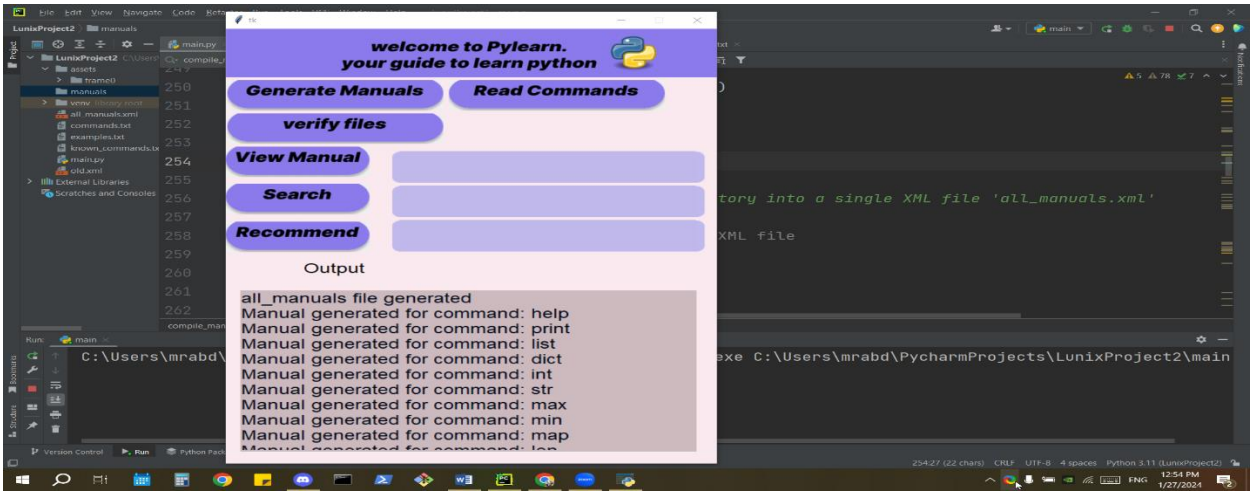
Output results and screenshots



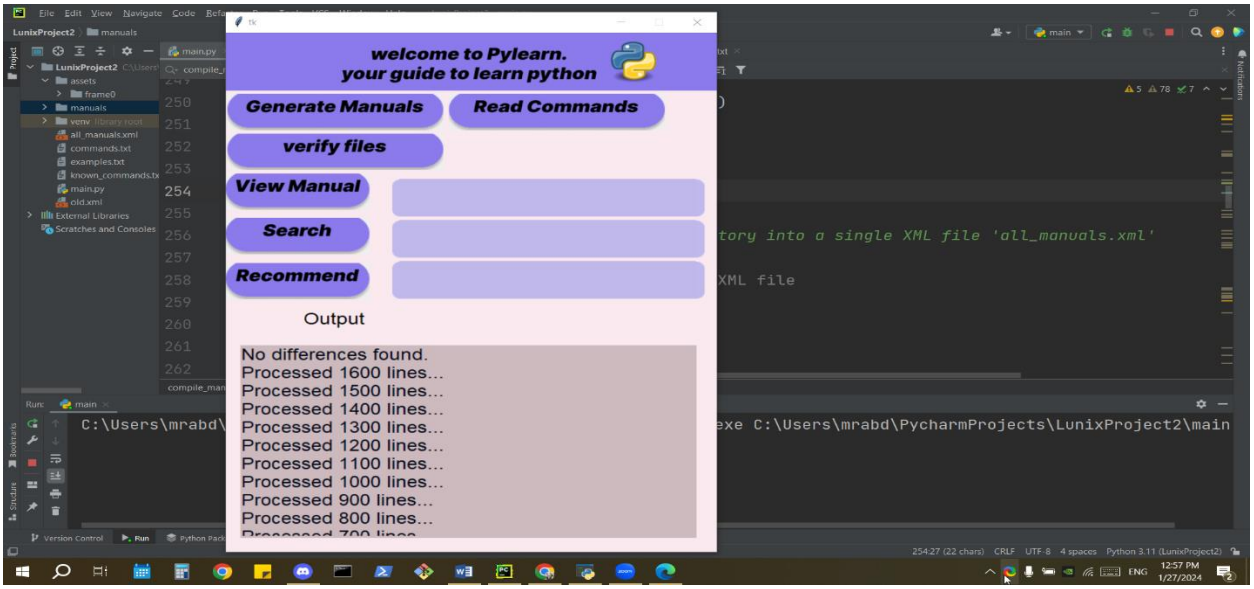
When read commands button clicked the program read the commands from the commands.txt files

And we have the output as all commands are read successfully as an indicator that the read worked

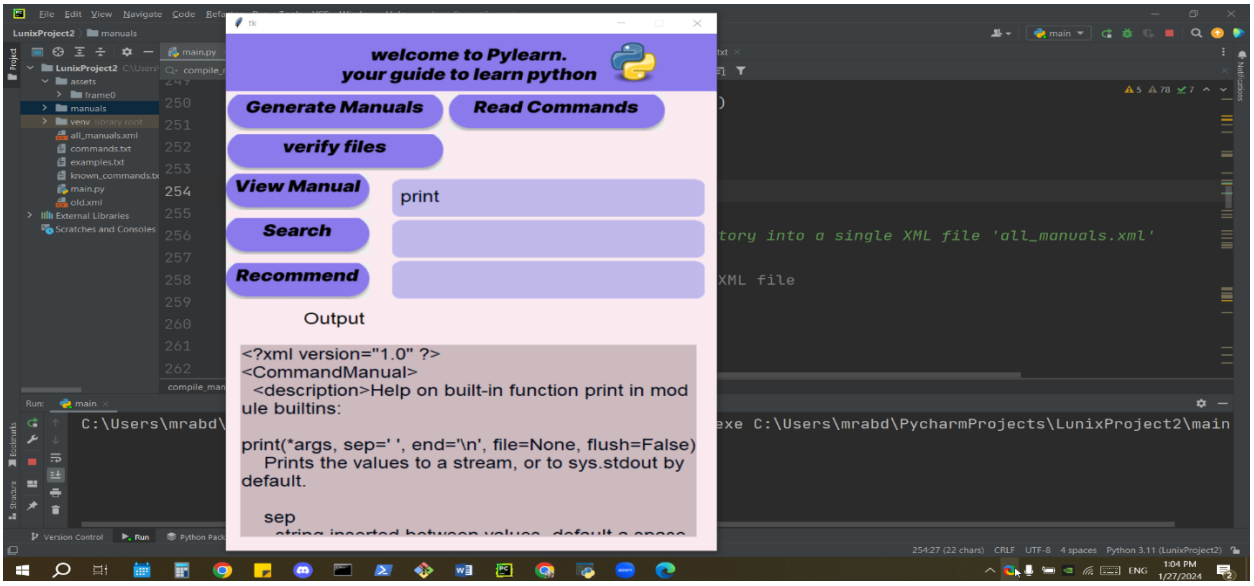
Else if the file was not found or something like that an error will occur as the output.



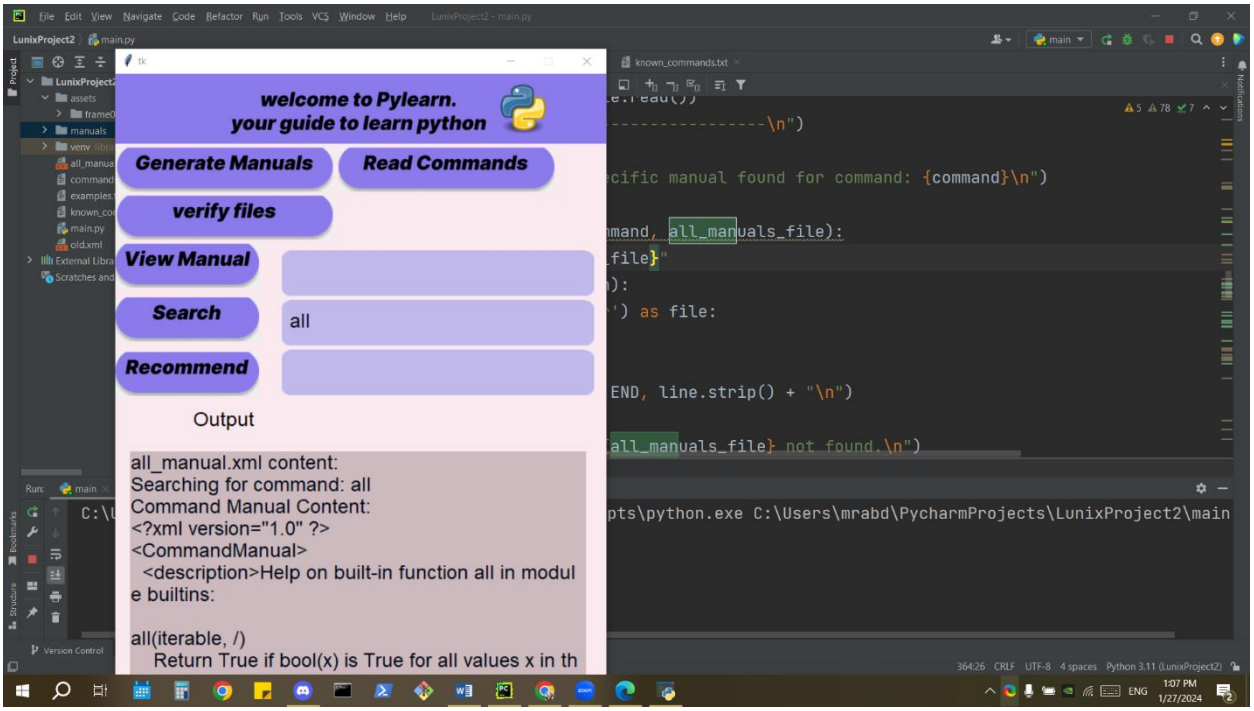
When generate manuals is clicked the system generate all the required manuals for the user



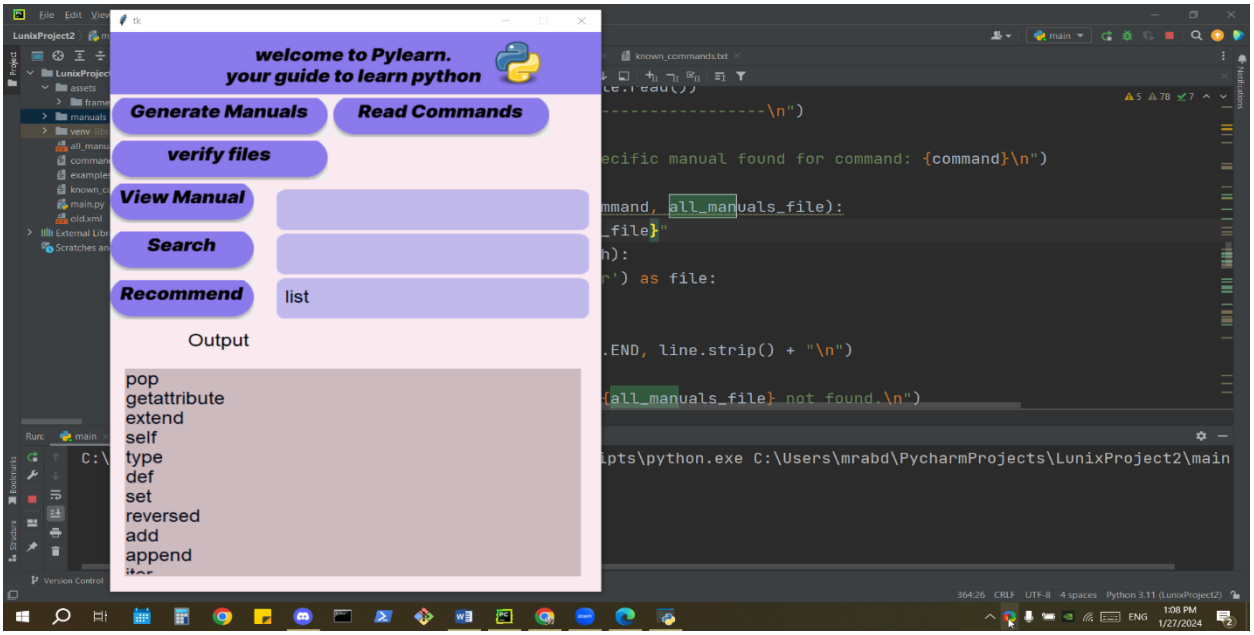
When the user tries to verify files, the system checks for the differences in the files and if there was It will be shown on the output.



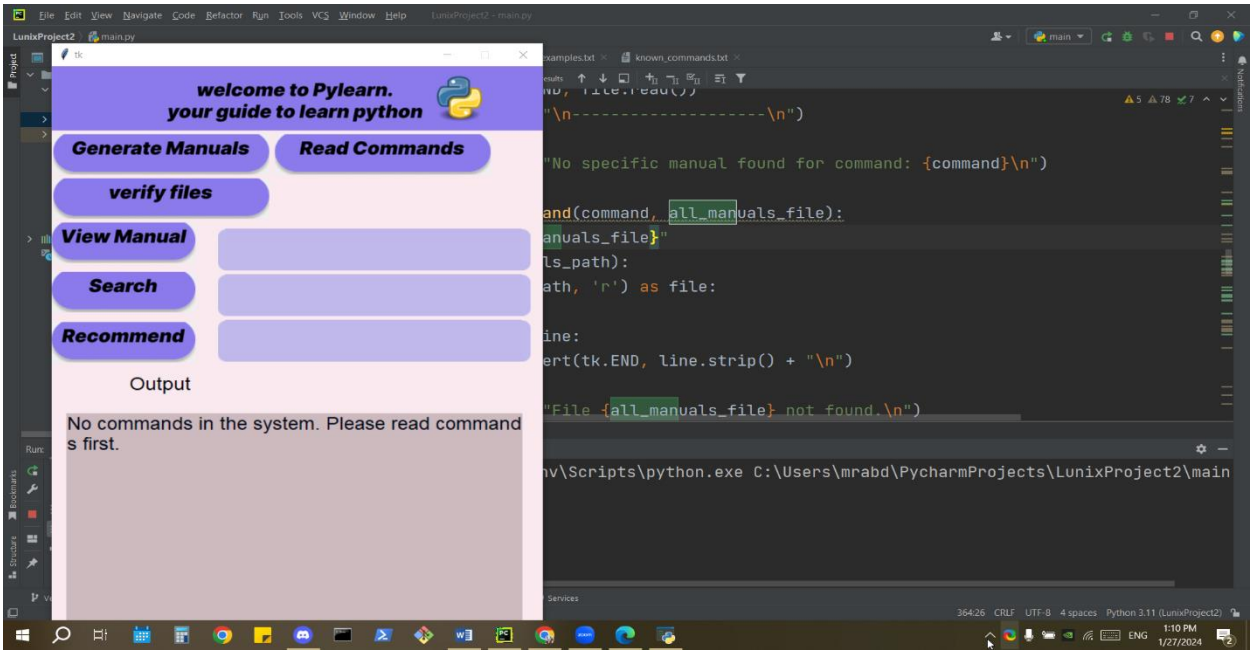
When the user enters the commands and click view manual the code print the corresponding manual



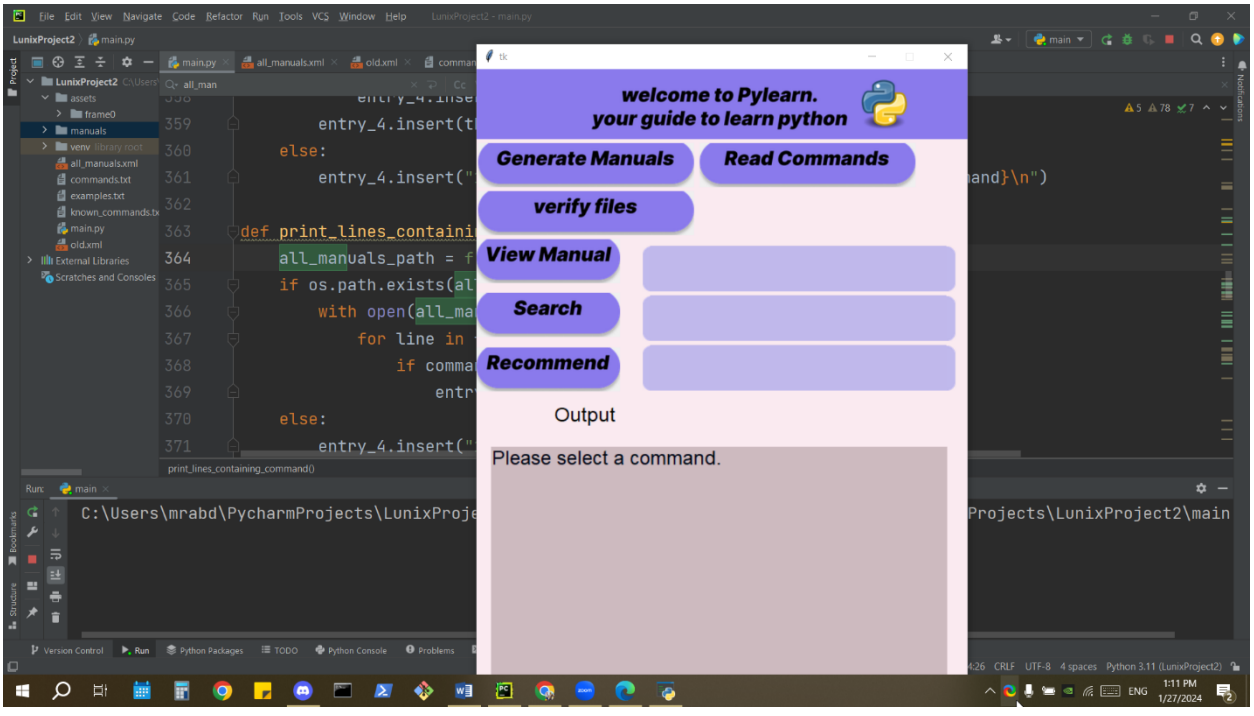
When the user enters the commands and click search the code print the corresponding manual and prints all occurrences of that word in all_manuals.xml which contains all info about all the manuals.



When the user enters the commands and click recommend the code fetch the related commands and print them. As they look the best choice for me to choose the related commands as recommendations for the currently used commands.

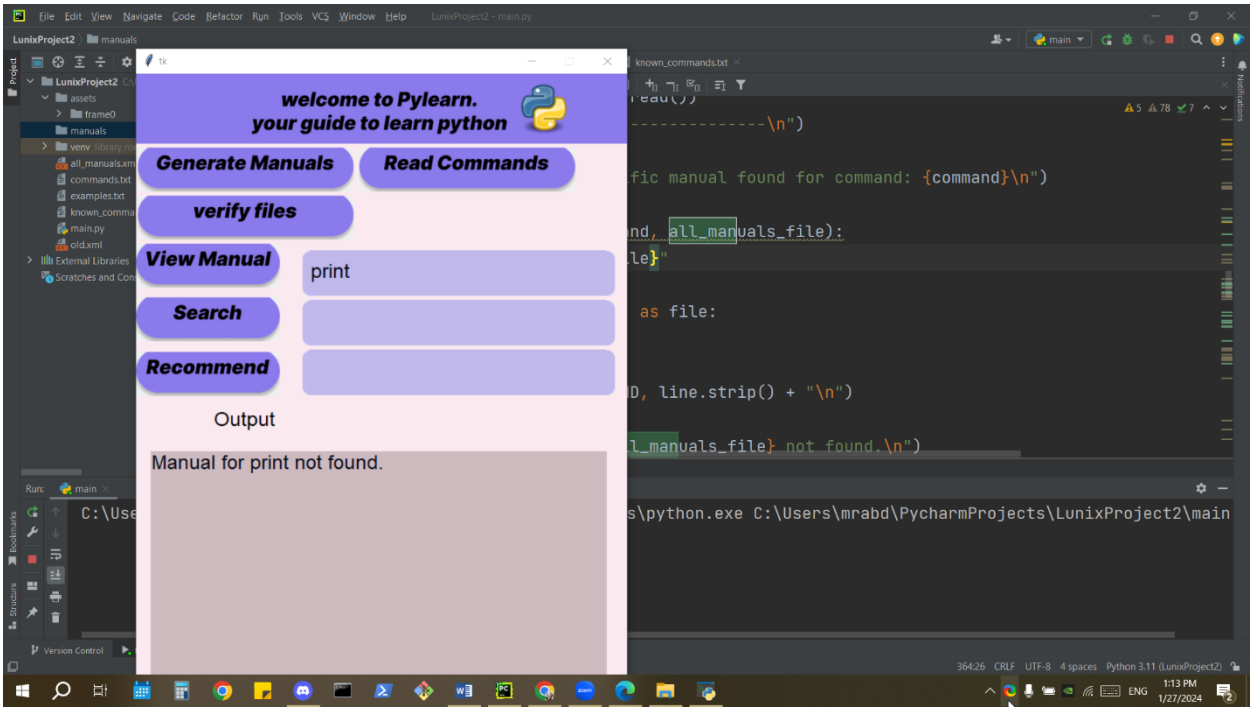


Assume user tried to do an operation without reading the commands. This message will occur.



Assume a user tried to do one of the three operations on commands and he/she forgot to fill the text.

Then this message will occur.



If an entered command is not found or has no manual this message then will occur.

Required considerations

Please install all packages that are not in your environment using pip install package_name.

Make sure to delete all manuals before running the code to make sure that everything is running perfectly.

Make sure to have the same order of the files.

Name	Date modified	Type	Size
.idea	1/27/2024 11:19 AM	File folder	
assets	1/27/2024 12:54 PM	File folder	
manuals	1/27/2024 1:19 PM	File folder	
venv	1/21/2024 3:59 AM	File folder	
commands.txt	1/27/2024 3:57 AM	Text Document	1 KB
examples.txt	1/27/2024 11:47 AM	Text Document	6 KB
known_commands.txt	1/27/2024 11:48 AM	Text Document	1 KB
main.py	1/27/2024 1:07 PM	Python source file	23 KB
old.xml	1/27/2024 4:24 AM	Microsoft Edge HT...	53 KB

Assets have all sources. Manuals should be initially empty, old.xml contains the latest version.

Commands.txt,examples.txt, known_commands.txt have the needed information for the code to run