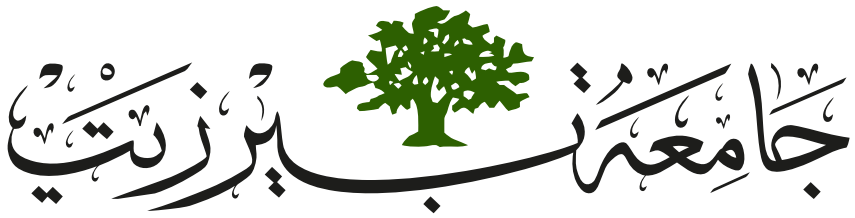

BIRZEIT UNIVERSITY

Department of Electrical & Computer Engineering
ENCS313 - Linux Lab

Python Project

Prepared by: Mumen Anbar 1212297
Instructor: Dr. Muhammad Jubran
TA: Eng. Tareq Zeidan
Date: January 31, 2024

Abstract

This project aims to develop a deep understanding of shell scripting and python programming language by combining both of them to generate UNIX command manuals having a Description, Version History, Examples on the command and some related commands.

The Command Manual Generator project is like a smart tool that uses Python to make working with command-line instructions easier. It takes simple commands you might use in a terminal, like 'ls' or 'echo', and turns them into organized manuals. This makes it simpler to understand what each command does and how to use it. The project uses Python's abilities to run shell commands, create graphical interfaces, and handle structured data in a neat way. It's a practical example of how Python can enhance and organize tasks we typically do in a shell, making things more user-friendly and efficient.

Contents

List of Figures

1 Theory

1.1 Python

Python is an interpreted, object-oriented, high-level programming language with dynamic semantics. Its high-level built in data structures, combined with dynamic typing and dynamic binding, make it very attractive for Rapid Application Development, as well as for use as a scripting or glue language to connect existing components together. Python's simple, easy to learn syntax emphasizes readability and therefore reduces the cost of program maintenance. Python supports modules and packages, which encourages program modularity and code reuse. The Python interpreter and the extensive standard library are available in source or binary form without charge for all major platforms, and can be freely distributed.[1]

1.2 Shell Scripting

A shell script is a text file that contains a sequence of commands for a UNIX-based operating system. It is called a shell script because it combines a sequence of commands, that would otherwise have to be typed into the keyboard one at a time, into a single script. The shell is the operating system's command-line interface (CLI) and interpreter for the set of commands that are used to communicate with the system.[2]

1.3 Xml File

An XML file contains XML code and ends with the file extension ".xml". It contains tags that define not only how the document should be structured but also how it should be stored and transported over the internet.

Let's look at a basic example of an XML file below. You can also click here to view the file directly in your browser.[3]

```
<?xml version="1.0"?>
- <birds>
  - <owl id="1201">
      <species>Bubo bubo</species>
      <name>Eagle Owl</name>
      <region>Eurasia</region>
    </owl>
  - <owl id="1202">
      <species>Strix occidentalis</species>
      <name>Spotted Owl</name>
      <region>North America</region>
    </owl>
</birds>
```

Figure 1.1: Xml File Example.

As you can see, this file consists of plain text and tags. The plain text is shown in black and the tags are shown in green.[3]

2 Procedure

2.1 Part 1: Generating Command Manual Content

In this part, each command manual content was generated through running some specific shell commands through python where each manual must contain the following:

2.1.1 Command Description

Command description was generated as follows:

```
def get_command_description(command_name):
    try:
        # Run 'man' command to get the manual page
        result = subprocess.run(['man', command_name], capture_output=True,
        text=True, check=True)

        # Extract and return the description (usually found in the NAME section)
        return delete_first_line(result.stdout.split('\n\n', 100)[3].strip())

    except subprocess.CalledProcessError as e:
        # If the command doesn't have a manual page, you can handle the exception
        return f"Error: {e}"
```

where the function, get_command_description(command_name) gets the description of a shell command by executing the 'man' command in a subprocess library. Then by splitting the output of the executed command according to new line character, taking the 4^{th} index of the splitted string.

2.1.2 Version History of the Command Generation

Version history of the command was generated as follows:

```
def get_command_version(command):
    try:
        # Run the command with the --version option
        result = subprocess.run([command, '--version'],
        stdout=subprocess.PIPE, stderr=subprocess.PIPE, text=True)

        # Check if the command executed successfully
        if result.returncode == 0:
            # Get the version information from the output
            version_info = result.stdout.split('\n', 2)[0].strip()
            # Delete the first word from the version information
            version_info = ' '.join(version_info.split()[1:])
            return version_info
        else:
            # Return the error message if the command failed
```

```
            return result.stderr.strip()
    except Exception as e:
        # Handle exceptions, if any
        return str(e)
```

Where it follows almost the steps in the previous generation where it runs the '–version' option. It runs the command in a subprocess, extracts and cleans the version details from the output (excluding the command name), and handles potential errors during execution, returning either the version information or an error message.

2.1.3 Command Example Generation

```
def get_command_example(command):
    try:
        # Run the command with the --version option
        orignial_command = command
        command = swtich(command)
        print(command)
        # Run the command and capture its output
        result = subprocess.Popen(command, shell=True,
        stdout=subprocess.PIPE, stderr=subprocess.PIPE, text=True)

        # Get the standard output and standard error
        stdout, stderr = result.communicate()

        # Check if the command executed successfully
        if result.returncode == 0:
            # Delete the first word from the version information
            return str(swtichOfExample(orignial_command)) + "\n" + str(stdout)
        else:
            # Return the error message if the command failed
            return result.stderr
    except Exception as e:
        # Handle exceptions, if any
        return str(e)
```

The get_command_example function is designed to retrieve examples for a shell command by running the command and capturing its output. The function incorporates a custom switch mechanism to modify the command if needed. It then executes the modified command in a subprocess, captures the standard output and standard error, and returns either the example output or an error message if the command execution fails. The function also considers potential exceptions during its execution.

Note that the external needed functions will be included in the submitted .py file.

2.1.4 Related Commands Generation

```
def get_command_releated_commands(command):
    command = f"bash -c 'compgen -c | grep {command}'"
```

```
    # Run the command and capture its output
    process = subprocess.Popen(command, shell=True,
    stdout=subprocess.PIPE, stderr=subprocess.PIPE, text=True)

    # Get the standard output and standard error
    stdout, stderr = process.communicate()
    stdout = stdout.replace('\n', ',')
    # Check if the command executed successfully
    if process.returncode == 0:
        # Print the output
        return stdout
    else:
        # Print the error
        print("Command failed. Error:")
        return
```

The get_command_related_commands function is designed to retrieve related commands for a given shell command by utilizing the (compgen -c) command in a subprocess. The function constructs a command that includes the input command for searching, executes it in a subprocess, and captures the standard output and standard error. The related commands, if found, are then processed and returned as a comma-separated string. If the command execution fails, an error message is printed, and the function returns without any specific output.

2.2 Part 2: Verification

```
def verify_xml_file(file_name):
tree = ET.parse(file_name)
root = tree.getroot()
Verification_message = ""
for xml_command in root:
    command_description = xml_command.find("Description").text
    original_description = generator.get_command_description(xml_command.tag)
    if (command_description == original_description):
        Verification_message += str("Same Description of "
        + xml_command.tag) + "\n\n"
    else:
         Verification_message += "Different in Description:\n"
         Verification_message += mark_intersection(command_description, original_desc

    command_version = xml_command.find("Version_History").text
    original_version = generator.get_command_version(xml_command.tag)
    if (command_version == original_version):
        Verification_message += str("Same Version_History of "
        + xml_command.tag) + "\n\n"
    else:
        Verification_message += "Different in Version_History:\n"
```

```
            Verification_message += mark_intersection(command_version,
            original_version, 2) + "\n\n"

        command_example = xml_command.find("Example").text
        original_example = generator.get_command_example(xml_command.tag)
        if (command_example == original_example):
            Verification_message += str("Same Example of "
            + xml_command.tag) + "\n\n"
        else:
            Verification_message += "Different in Example:\n"
            Verification_message += mark_intersection(command_example,
            original_example, 3) + "\n\n"

        return Verification_message
```

The verify_xml_file function is designed to verify the consistency of information stored in an XML file with the original information obtained from system commands. It parses the XML file, extracts information for each command (description, version history, and example), and compares it with the original information fetched using generator functions. The function generates a verification message detailing whether the information matches or differs.

For each command in the XML file, the function compares the stored information (description, version history, and example) with the information obtained using generator functions. If the stored and original information matches, it appends a message indicating the similarity. If there is a difference, it appends a message indicating the discrepancy and uses the mark_intersection function to mark the intersected and non-intersected parts between the stored and original information, providing a visual representation of the differences.

2.3 Part 3: Continuous Improvement and Extension

2.3.1 Command Recommendation System

```
def get_command_recommendation(command):
    # File and Directory Operations
    Directory_Operations = ['ls', 'touch', 'mv', 'cp', 'ln']
    if command in Directory_Operations:
        return sub_list(Directory_Operations, command)

    # Text Processing and Searching
    Text_Processing = ['echo', 'cut', 'grep', 'sed', 'sort', 'find', 'wc', 'tr']
    if command in Text_Processing:
        return sub_list(Text_Processing, command)

    # System Information
    System_Information = ['who', 'lspci', 'ps', 'whatis', 'man', 'which']
    if command in System_Information:
```

```
        return sub_list(System_Information, command)

    # Shell Built-in and Control Flow
    if command in ['test']:
        return sub_list(Text_Processing, command)

    return 'Command not recognized.'
```

The get_command_recommendation function provides recommendations for related commands based on the input command's category. It categorizes commands into different groups, such as File and Directory Operations, Text Processing and Searching, System Information, and Shell Built-in and Control Flow. If the input command is recognized in one of these categories, the function returns a sublist of related commands within that category. If the command is not recognized in any category, it returns a message indicating that the command is not recognized. The function uses the 'sub_list' helper function, which returns the target recommended group without the command itself, but it's implied to return a sublist of related commands from a given category.

2.3.2 Search Functionality

Will go through this topic in GUI section in details.

2.4 Part 4: OOP Integration

2.4.1 'CommandManualGenerator' Class

```
class CommandManualGenerator:
    def __init__(self, input_file):
        self.input_file = input_file
        self._manuals = []
        self._xml_files = []

    def get_manuals(self):
        return self._manuals
    def get_xml_files_objects(self):
        return self._xml_files

    def generate_manuals(self):
        with open(self.input_file, 'r') as file:
            commands = file.readlines()

        for command in commands:
            self._manuals.append(CommandManual(command.strip()))

        for i in self._manuals:
            cur_xml_file = XmlCreator.XmlSerializer(i)
            cur_xml_file.create_xml_files()
            self._xml_files.append(cur_xml_file)
```

The CommandManualGenerator class serves as a crucial component in the manual generation process. Upon initialization, it takes an input file containing a list of commands. The class maintains lists for both the generated manuals (_manuals) and their corresponding XML files (_xml_files). The generate_manuals method reads commands from the input file, creates a CommandManual object for each command, and appends it to the manuals list. Additionally, it utilizes the XmlSerializer class to generate XML files for each command manual, and the XML files' objects are stored in the XML files list. The class provides methods (get_manuals and get_xml_files_objects) to retrieve the generated manuals and XML files, respectively. Overall, this class plays a pivotal role in orchestrating the generation of manuals and their associated XML files.

2.4.2 'CommandManual' Class

```
class CommandManual:

    def __init__(self, command):
        self._command = command
        self._description = generator.get_command_description(command.strip())
        self._version_history = generator.get_command_version(command.strip())
        self._examples = generator.get_command_example(command.strip())
        self._related_commands = generator.get_command_releated_commands(command.stri

    def get_command(self):
        return self._command

    def get_description(self):
        return self._description

    def set_description(self, value):
        self._description = value

    def get_version_history(self):
        return self._version_history

    def set_version_history(self, value):
         self._version_history = value

    def get_examples(self):
        return self._examples

    def examples(self, value):
        self._examples = value

    def get_related_commands(self):
        return self._related_commands

    def set_related_commands(self, value):
        self._related_commands = value
```

The CommandManual class encapsulates information related to a specific command. Upon initialization, it retrieves the description, version history, examples, and related commands using various methods from the ‘generator‘ module. The class provides getter methods for accessing these attributes (‘get_command‘, ‘get_description‘, ‘get_version_history‘, ‘get_examples‘, ‘get_related_commands‘). Additionally, setter methods (‘set_description‘, ‘set_version_history‘, ‘examples‘, ‘set_related_commands‘) are defined for modifying these attributes. However, there’s a typo in the ‘examples‘ method, and it should be corrected to ‘set_examples‘. This class promotes encapsulation by managing command-related details within its instance variables and offers methods to retrieve and modify these details. It serves as an essential component for creating and managing individual command manuals.

2.4.3 ‘XmlSerializer’ Class

```
class XmlSerializer:
    def __init__(self, Command):
        self.Command = Command

    def add_Section(self, parent_element, curCommand):
        description_element = ET.SubElement(parent_element, "Description")
        version_element = ET.SubElement(parent_element, "Version_History")
        example_element = ET.SubElement(parent_element, "Example")
        related_commands_element = ET.SubElement(parent_element, "Related_Commands")

        description_element.text = curCommand.get_description()
        version_element.text = curCommand.get_version_history()
        example_element.text = curCommand.get_examples()
        related_commands_element.text =
        curCommand.get_related_commands()

    def create_xml_files(self):
        # Create the root element
        self.tree = ET.ElementTree(ET.Element("Manuals"))


        manual_Sections = ["Description",
        "Version History", "Example", "Related Commands"]

        # Create sub-elements for close friends, normal friends, and best friends
        command_Manual = ET.SubElement(self.tree.getroot(),
        self.Command.get_command())
        self.add_Section(command_Manual, self.Command)

        # Save the XML tree to a file
        self.tree.write(f"{self.Command.get_command()}.xml")
        print(f"{self.Command.get_command()}.xml Creation Is Done!!")
```

```
    def get_command_name(self):
        return self.Command.get_command()

    def get_Manual(self):
        return self.tree
```

The ‘XmlSerializer‘ class plays a crucial role in the process of generating XML files for command manuals. It is responsible for creating and managing XML structures based on the details of a specific command provided during instantiation. The class contains methods to add sections (‘add_Section‘) such as description, version history, examples, and related commands to the XML structure. The ‘create_xml_files‘ method orchestrates the creation of the XML file for the associated command, using the provided details. The resulting XML file is named after the command and stored in the current directory. The class also provides getter methods (‘get_command_name‘ and ‘get_Manual‘) for accessing the name of the associated command and the entire XML structure, respectively. Overall, the ‘XmlSerializer‘ class serves as a key component in the generation of command manuals in XML format, promoting separation of concerns by encapsulating XML creation logic.

2.5 Part 5: GUI Integration

Since the code is really large and not appropriate to include in the report, it will be attached with the submitted report while the results will be represented here.

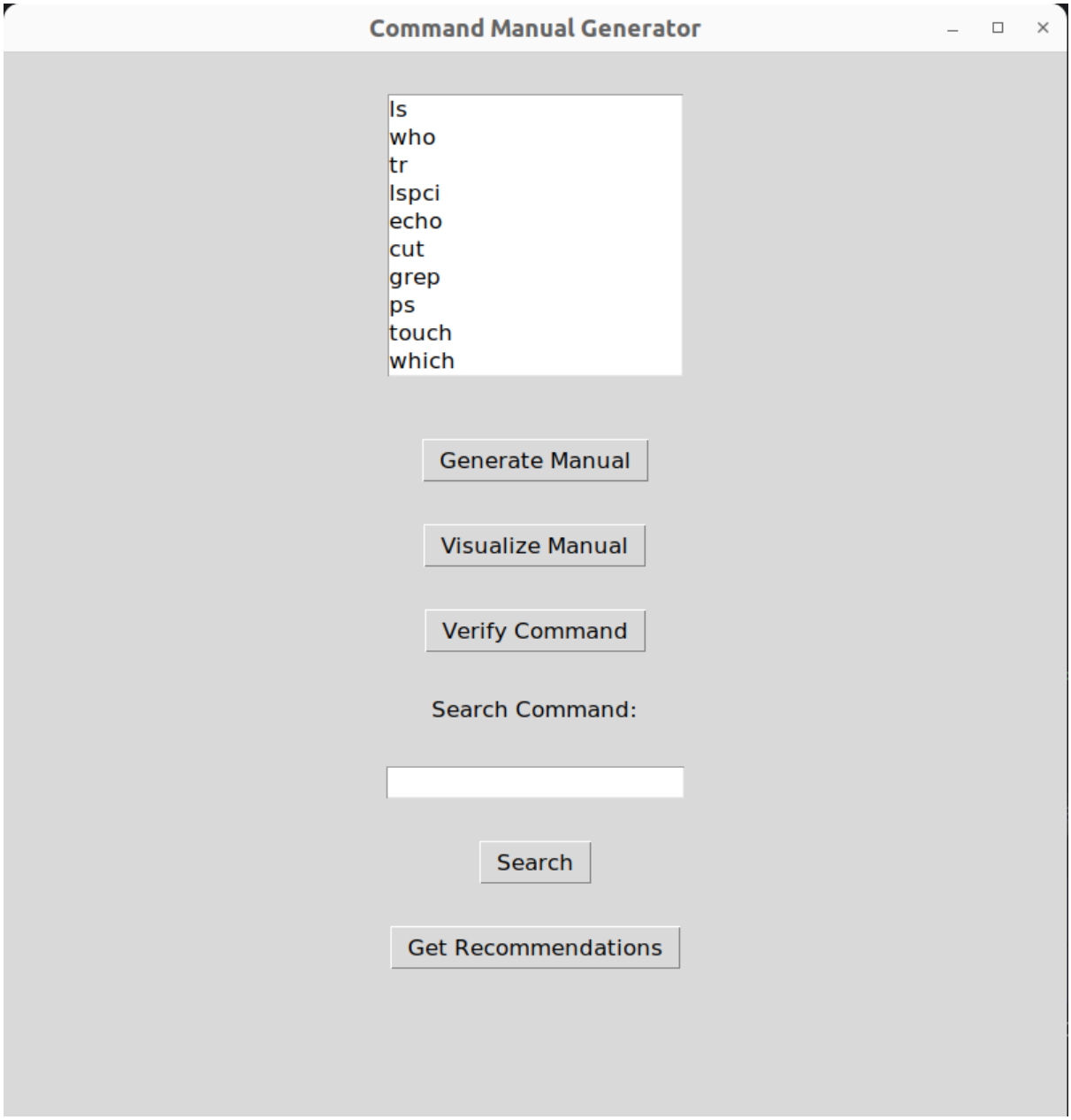

Figure 2.1: The Main Interface.

2.5.1 Command Selection

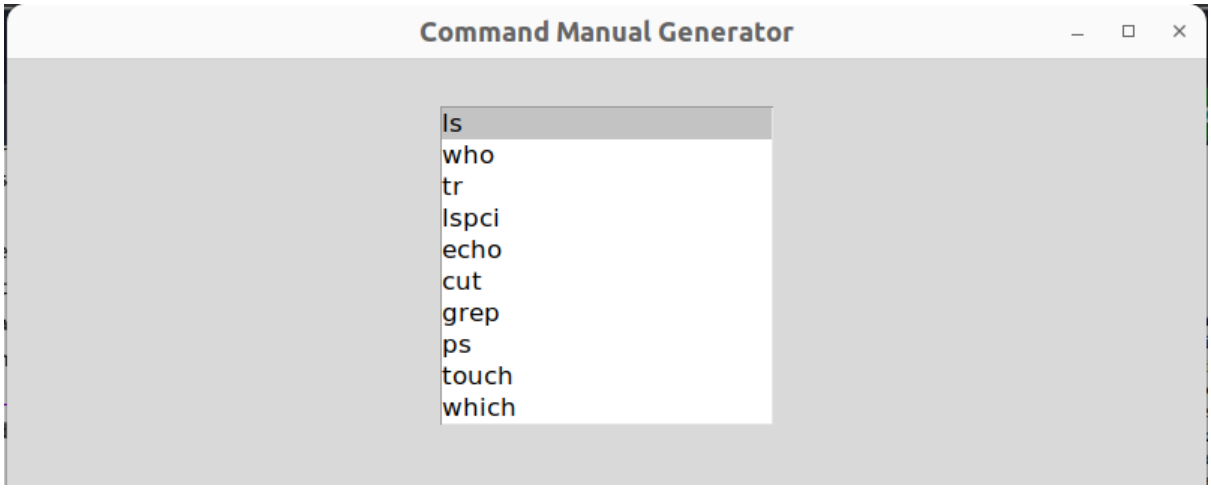

Figure 2.2: Command Selection.

2.5.2 Manual generation

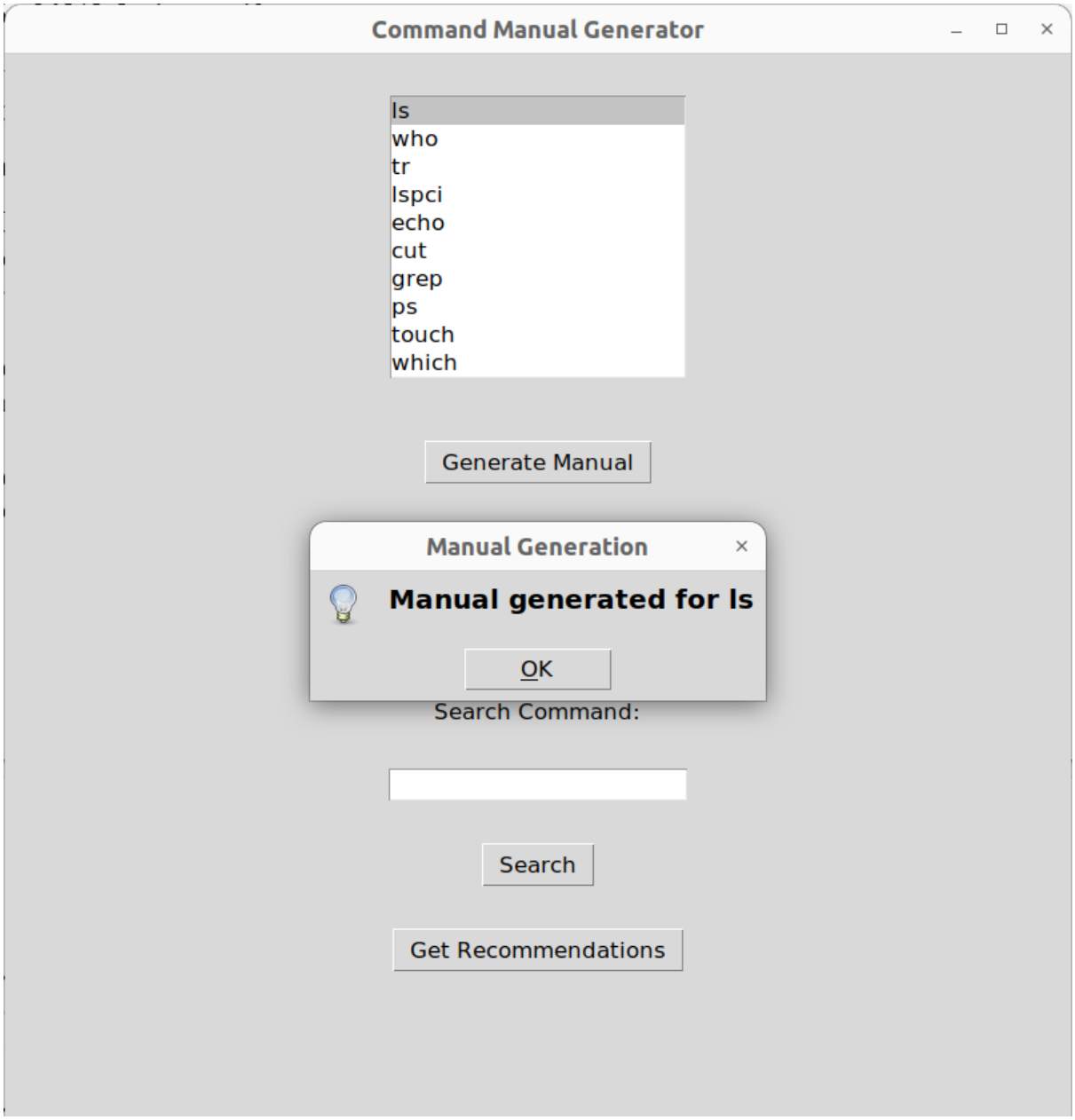

Figure 2.3: Manual generation.

2.5.3 Manual Visualization

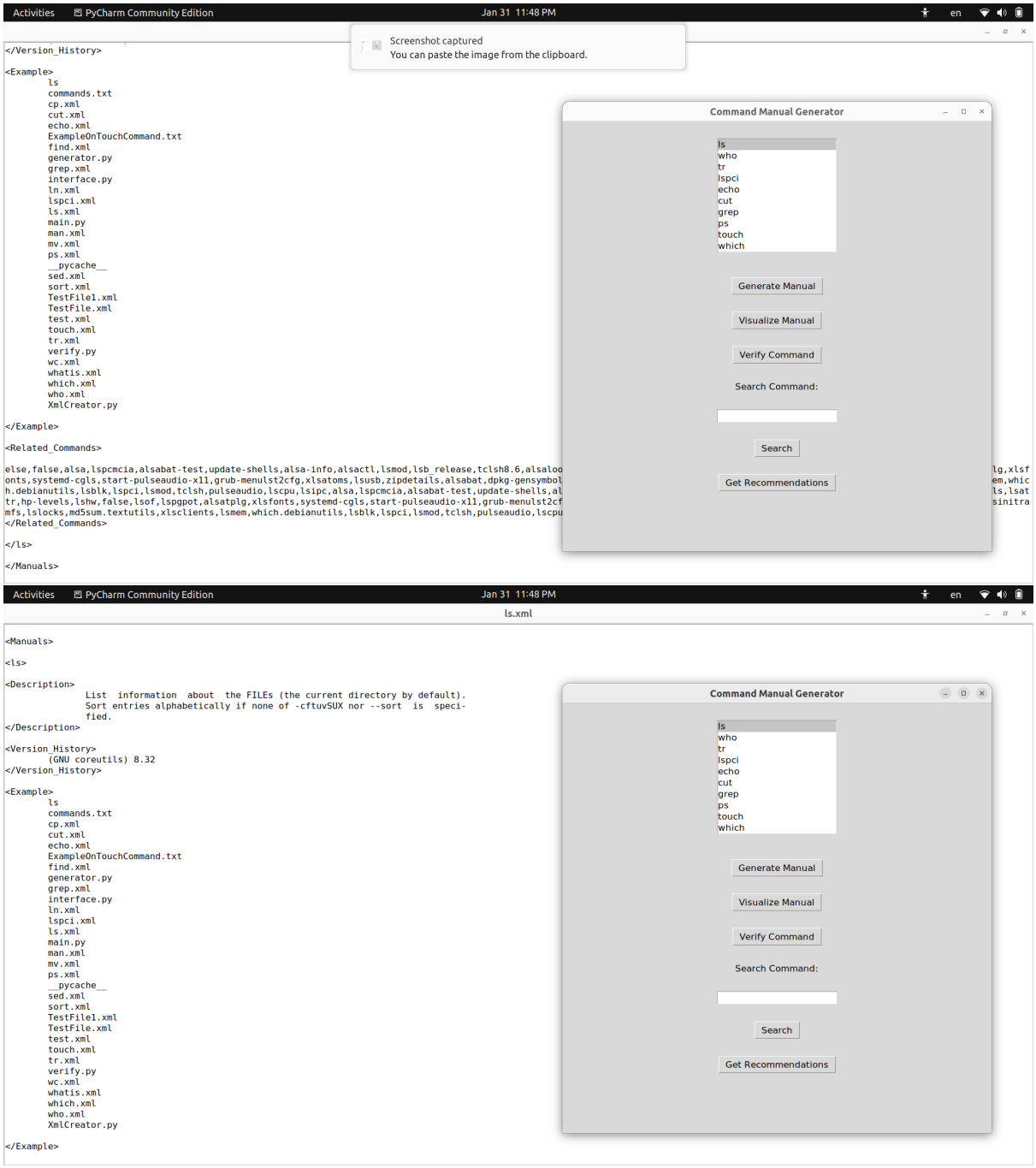

Figure 2.4: Visualization of "ls" Manual.

2.5.4 Verification Interface

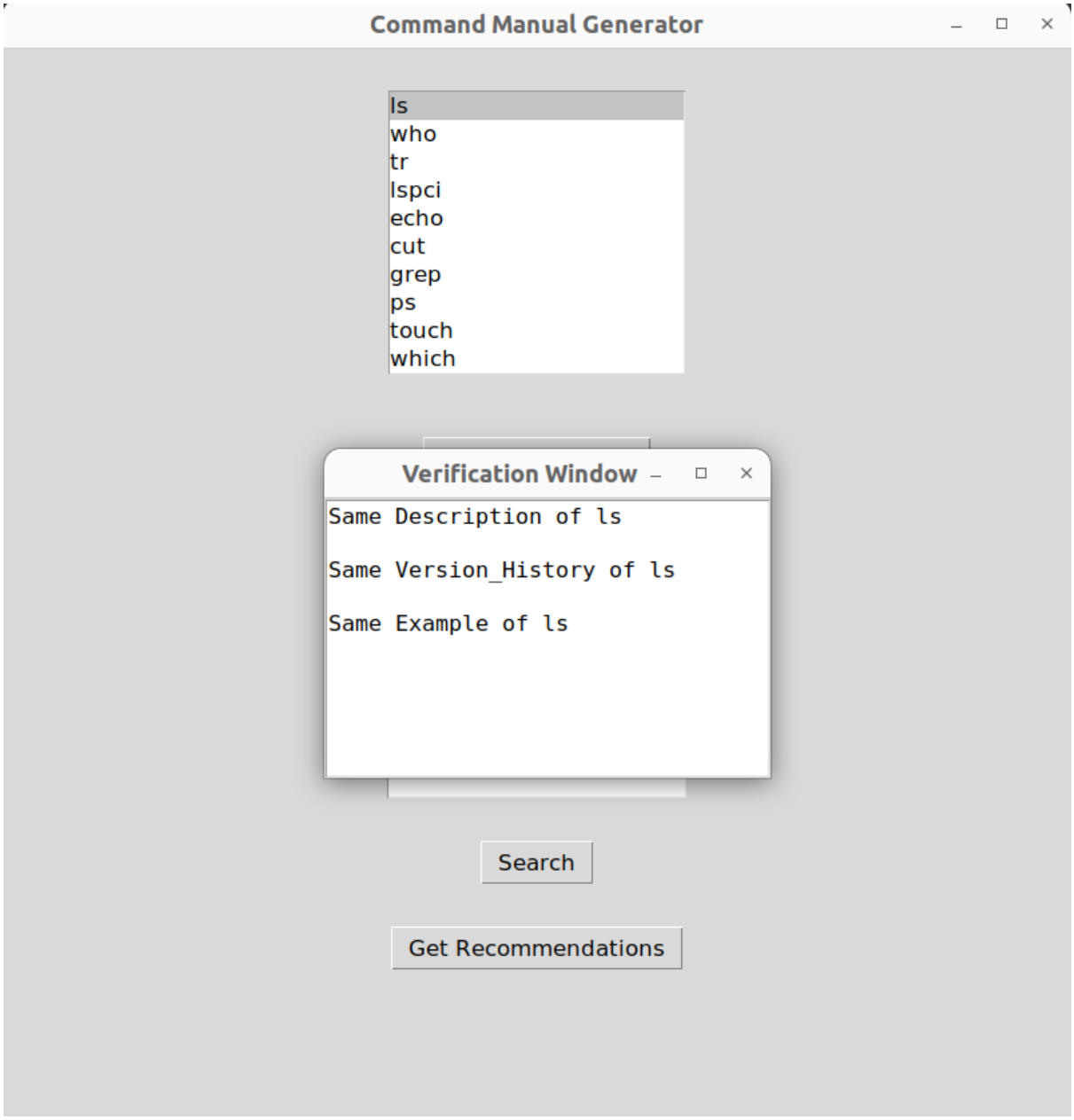

Figure 2.5: Verification Interface of ls Manual Content.

Figure 2.6: Verification Interface of ps Manual Content Showing The Differences.

2.5.5 Search Functionality

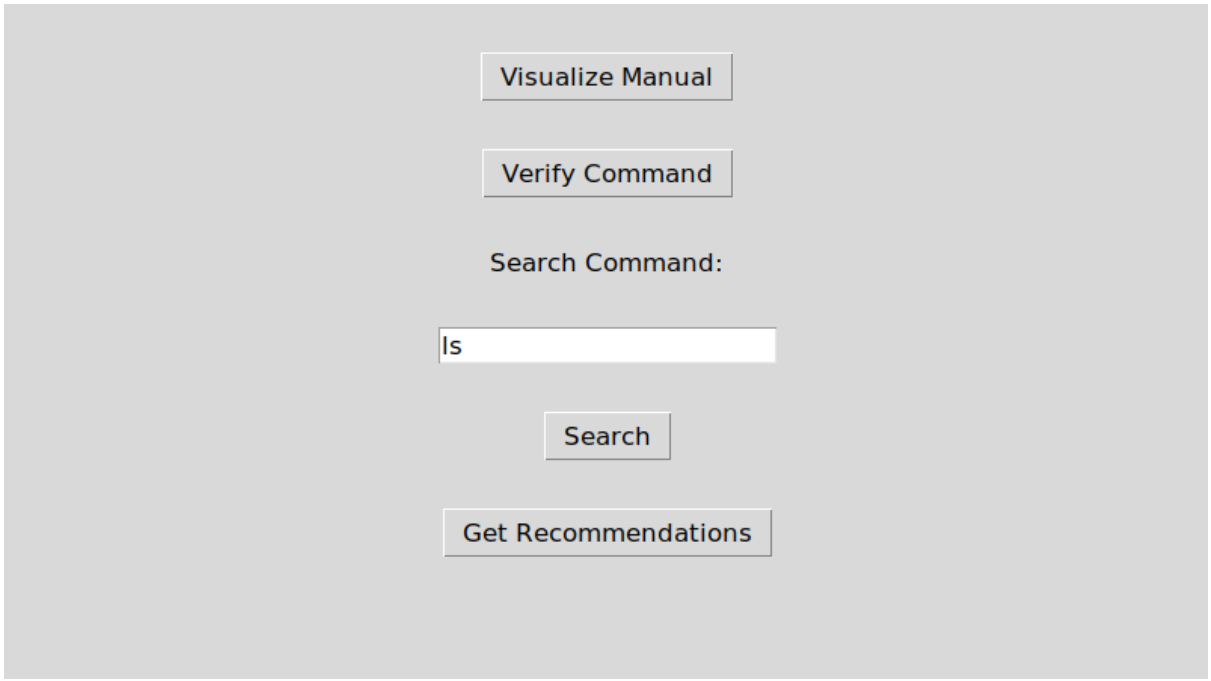

Figure 2.7: Search About ls Command.

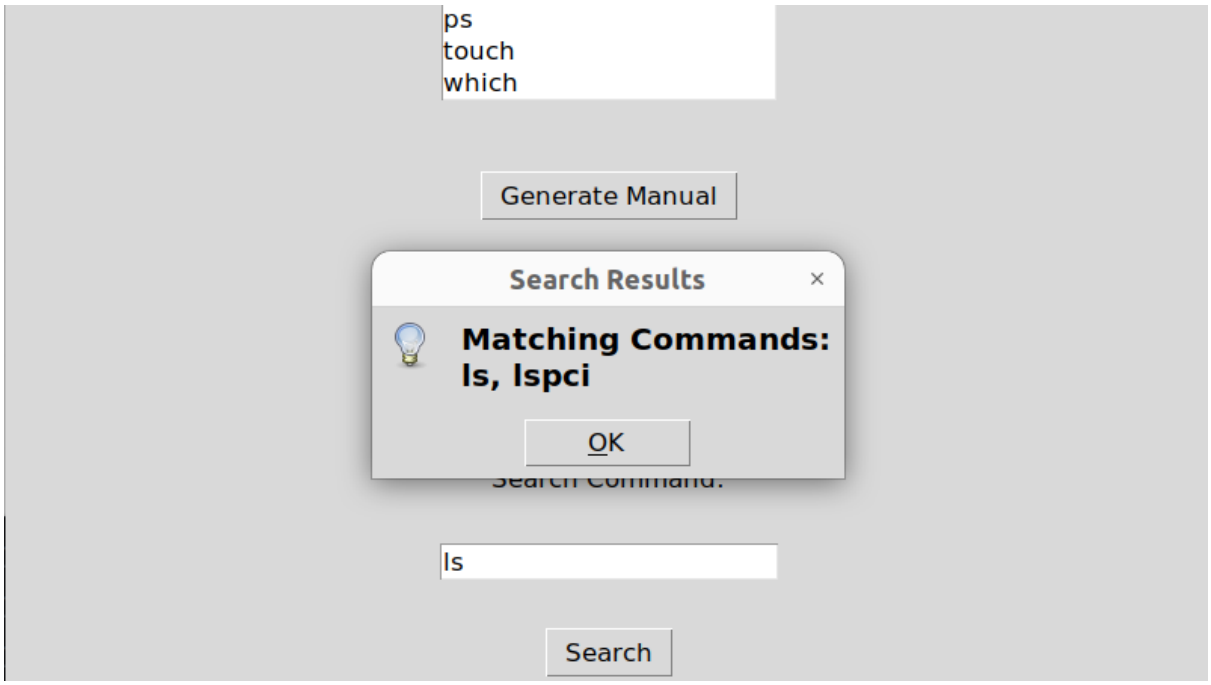

Figure 2.8: Results of The Searching.

2.5.6 Command Recommendation Panel

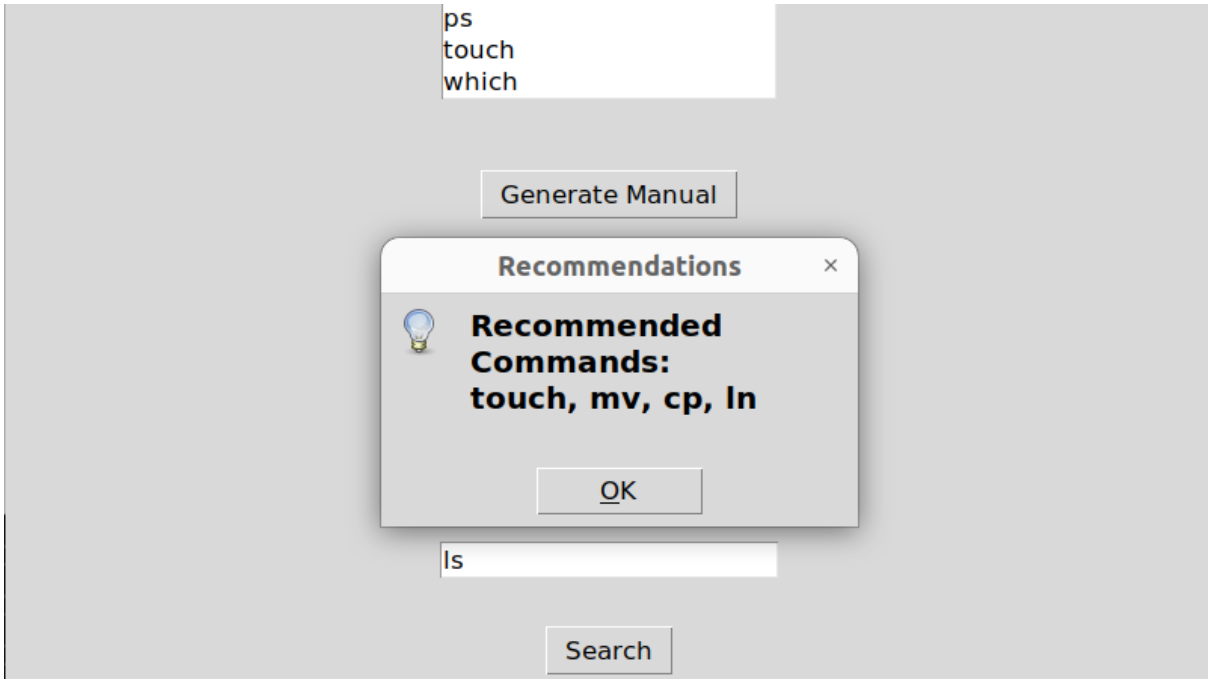

Figure 2.9: Recommendation Based on the Previous Command.

3 Conclusion

The project involving the creation of a Command Manual Generator in Python represents a significant achievement in bridging the gap between the principles learned in shell scripting and their application in a more dynamic and versatile programming language. By leveraging Python's capabilities, the project provides a comprehensive solution for generating, visualizing, and verifying command manuals. The integration of subprocess, tkinter for GUI, and XML manipulation showcases the versatility of Python in handling diverse tasks.

The Command Manual Generator seamlessly connects the principles of shell scripting, where commands are central, to the object-oriented paradigm of Python. The design encapsulates each command's details into Python objects, promoting modularity and readability. Subprocess usage facilitates communication with the underlying shell environment, aligning with shell scripting practices. The GUI functionality enriches user interaction, emphasizing the user-friendly nature of Python development.

Moreover, the XML serialization process demonstrates a transition from text-based representation in shell scripting to a structured and standardized format, enhancing readability and extensibility. The encapsulation of command details into Python objects aligns with the object-oriented paradigm, facilitating code organization and maintenance.

In conclusion, this project is a testament to the seamless integration of shell scripting principles into Python, showcasing the language's capabilities in handling system commands, creating graphical interfaces, and managing structured data. It not only builds upon the fundamentals learned in shell scripting but also highlights Python's strengths in terms of readability, modularity, and versatility. The Command Manual Generator serves as an excellent example of how Python can enhance and extend the principles learned in shell scripting to create powerful and user-friendly applications.

References

[1] Python. Accessed on 2023-01-30. [Online]. Available: https://www.python.org/doc/essays/blurb/

[2] S. Scripting. (2023) Accessed on 2023-01-30. [Online]. Available: https://www.techtarget.com/searchdatacenter/definition/shell-script

[3] (2022) Xml files. Accessed on 2023-01-30. [Online]. Available: https://blog.hubspot.com/website/what-is-xml-file