Project: Command Shell

T1 Planning and Design based on Mathematical and Declarative Programming techniques:

In designing the Command Shell, mathematical structures and declarative programming techniques are crucial in creating a robust, flexible, and efficient shell environment. This shell incorporates various software engineering principles and mathematical concepts to ensure it operates effectively and provides a seamless user experience.

Mathematical Structures:

Following mathematical structures were used:

- **String Manipulation and Regular Expressions:** The shell frequently manipulates strings to handle user inputs, generate prompts, and manage command execution. The use of string replacement functions to dynamically generate the prompt format is an example of mathematical string operations.
- **Error Handling and Control Flow:** The shell uses conditional logic to handle various scenarios, such as directory changes, listing directory contents, and executing commands. This involves the application of mathematical logic principles to control the flow of execution based on user input and system state.

Declarative Programming Techniques:

Following declarative programming techniques were used:

- **State Representation:** The shell maintains the current working directory and prompt format as state variables. These are used declaratively to generate the prompt and determine the behavior of commands.
- **String Interpolation for Prompts:** The generate_prompt method declaratively constructs the shell prompt by replacing placeholders with their corresponding values.
- **Command Parsing and Handling:** The handle_command method declaratively maps user input to specific command handlers. The structure of this method makes it clear what commands are supported and what each command does without detailing the control flow.

Software Engineering Principles:

In the making of this project, Following software engineering principles were used:

- **Abstraction:** The shell uses abstraction to hide complex logic and provide simple interfaces for interacting with the operating system. For instance, the 'change_directory' and 'list_directory_contents' methods abstract away the details of changing directories and listing contents, allowing the main loop to interact with these functionalities in a straightforward manner.
- **Encapsulation:** Encapsulation is used to protect the internal state of the shell and provide controlled access through methods. For example, the 'prompt_format' attribute is modified only through the 'set_prompt_format' method, ensuring that prompt updates follow a specific format.
- **Modularity:** The shell is divided into distinct methods, each responsible for specific aspects of the shell's functionality. This modularity improves code organization, making it easier to understand, maintain, and extend. For example, different methods handle directory changes, command execution, and prompt generation.
- **Reusability:** The shell logic is designed to be reusable. The command handling, directory management, and prompt generation systems are implemented in a way that allows them to be reused across different parts of the shell without modification.

The Command Shell uses mathematical structures, such as string manipulation and control flow logic, to create efficient command handling. It also employs declarative programming techniques and software engineering principles like abstraction, encapsulation, modularity, and reusability to build a maintainable and scalable shell. By integrating these concepts, the project provides a great experience for users to interact with the shell.

T2 Program Implementation based on Declarative Programming tools and techniques.

Source Code:

```
'''
```

```
Command Shell:
Programming Language: Python
Interface: CLI based
Student Name: Aisha Abdi
Student ID: 22160571

'''

import os
import sys
import subprocess

class Shell:
    def __init__(self):
        # Initialize the shell with the home directory and default prompt
format
        self.home = os.path.expanduser("~")
        self.prompt_format = "PythonShell:$PWD$ "

    def main(self):
        # Print welcome message and start the shell loop from the home
directory
        print("Welcome to the Python Shell")
        os.chdir(self.home)
        self.shell_loop()

    def shell_loop(self):
        # Continuously prompt for user input and handle commands
        while True:
            cwd = os.getcwd()  # Get current working directory
            prompt = self.generate_prompt(cwd)
            try:
                user_input = input(prompt)  # Display prompt and get user
input
                self.handle_command(user_input)
            except EOFError:
                break  # Exit loop on EOF
            except KeyboardInterrupt:
                print()  # Handle keyboard interrupt (Ctrl+C) gracefully
                continue

    def generate_prompt(self, cwd):
        # Generate the shell prompt by replacing placeholders with actual
values
        return self.prompt_format.replace("$PWD$", cwd).replace("$HOME$",
self.home)

    def handle_command(self, user_input):
```

```
        # Parse the user input and execute the appropriate command
        commands = user_input.split()
        if not commands:
            return

        cmd, *args = commands
        if cmd == "cd":
            self.change_directory(args)
        elif cmd == "ls":
            self.list_directory_contents(args)
        elif cmd == "exit":
            self.exit_shell()
        elif cmd == "prompt":
            self.set_prompt_format(args)
        else:
            self.run_external_command(cmd, args)

    def change_directory(self, args):
        # Change the current working directory, handle errors if directory
does not exist
        dir = args[0] if args else os.path.expanduser("~")
        try:
            os.chdir(dir)
        except FileNotFoundError:
            print(f"cd: no such file or directory: {dir}")

    def list_directory_contents(self, args):
        # List the contents of the specified or current directory, handle
errors if directory does not exist
        dir = args[0] if args else "."
        try:
            for entry in os.listdir(dir):
                print(entry)
        except FileNotFoundError:
            print(f"ls: cannot access '{dir}': No such file or directory")

    def run_external_command(self, cmd, args):
        # Execute external commands using subprocess.run, handle errors if
command is not found
        try:
            subprocess.run([cmd] + args)
        except FileNotFoundError:
            print(f"{cmd}: command not found")
        except Exception as e:
            print(f"{cmd}: {e}")

    def exit_shell(self):
        # Exit the shell loop and terminate the program
```

```
        print("Exiting Python Shell")
        sys.exit(0)

    def set_prompt_format(self, args):
        # Update the shell's prompt format based on user input, provide usage
information if no arguments are given
        if args:
            self.prompt_format = ' '.join(args)
        else:
            print("Usage: prompt <format>")
            print("Available placeholders: $PWD$, $HOME$")

if __name__ == "__main__":
    shell = Shell()
    shell.main()
```

Output Screenshots:

```
Welcome to the Python Shell
PythonShell:C:\Users\saadG cd ..
PythonShell:C:\Users cd aishaA
PythonShell:C:\Users\aishaA ls
Desktop
PythonShell:C:\Users\aishaA cd Desktop
PythonShell:C:\Users\aishaA\Desktop ls -1
ls: cannot access '-1': No such file or directory
PythonShell:C:\Users\aishaA\Desktop ls
(20F-0318)(fyp_activity).fig
2D#4.zip
2D#4_RFC.txt
Task
Telegram bot
testproject
umrah app
x64
x86
PythonShell:C:\Users\aishaA\Desktop exit
Exiting Python Shell
```

Declarative Implementation of Command Shell in Python:

To implement a command-line shell, we define the Shell class that encapsulates the state and behavior of the shell. This class manages the current working directory, prompt format, and provides methods to handle user commands.

Data Structures and Type Definitions:

- The Shell class represents the command-line shell. It includes methods to change directories, list directory contents, execute external commands, and manage the shell prompt.

Behaviour Step Implementations:

- **Initialization and Main Loop:** The shell initialization sets up the initial state, including the home directory and prompt format. The shell_loop continuously prompts the user for input and handles commands.
- **Generating the Prompt:** The generate_prompt method replaces placeholders in the prompt format with actual values.
- **Handling Commands:** The handle_command method parses user input and dispatches to the appropriate command handler.
- **Changing Directory:** The change_directory method changes the current working directory, handling errors if the directory does not exist.
- **Listing Directory Contents:** The list_directory_contents method lists the contents of the specified or current directory, handling errors if the directory does not exist.
- **Running External Commands:** The run_external_command method executes external commands using subprocess.run, handling errors if the command is not found.
- **Exiting the Shell:** The exit_shell method exits the shell loop and terminates the program.
- **Setting the Prompt Format:** The set_prompt_format method updates the shell's prompt format based on user input.

Areas of Improvement:

- Implement a help command to provide usage information for built-in commands.
- Allow users to define and load custom commands or scripts to extend the functionality of the shell.
- Provide configuration options for customizing the shell's behavior and appearance.
- Add support for environment variables and shell scripting capabilities.

T3 Testing and Verification of Programs via appropriate tools and techniques

Test Case ID	Test Case Description	Test Steps	Test Data	Expected Results	Actual Results	Pass/Fail
TC1	Verify changing directory using 'cd' command	1. Start the shell. 2. Enter 'cd' followed by a valid directory path.	Valid directory path	The shell changes the current directory to the specified path.	As Expected	PASS
TC2	Verify changing directory to home using 'cd' command	1. Start the shell. 2. Enter 'cd' without any arguments.	None	The shell changes the current directory to the home directory.	As Expected	PASS
TC3	Verify changing directory to non-existent path	1. Start the shell. 2. Enter 'cd' followed by a non-existent directory path.	Non-existent directory path	The shell displays an error message indicating directory not found.	As Expected	PASS
TC4	Verify listing current directory contents using 'ls'	1. Start the shell. 2. Enter 'ls' command.	None	The shell lists the contents of the current directory.	As Expected	PASS
TC5	Verify listing contents of a specific directory using 'ls'	1. Start the shell. 2. Enter 'ls' followed by a valid directory path.	Valid directory path	The shell lists the contents of the specified directory.	As Expected	PASS

TC6	Verify running external command successfully	1. Start the shell. 2. Enter a valid external command (e.g., 'echo').	Valid external command	The shell executes the external command successfully.	As Expected	PASS
TC7	Verify running external command that does not exist	1. Start the shell. 2. Enter a non-existent external command.	Non-existent command	The shell displays an error message indicating command not found.	As Expected	PASS
TC8	Verify setting custom prompt format using 'prompt' command	1. Start the shell. 2. Enter 'prompt' followed by a custom format.	Custom prompt format	The shell updates the prompt format according to the input.	As Expected	PASS
TC9	Verify setting custom prompt format with no arguments	1. Start the shell. 2. Enter 'prompt' without any arguments.	None	The shell displays usage information for the 'prompt' command.	As Expected	PASS
TC10	Verify graceful termination of the shell using 'exit' command	1. Start the shell. 2. Enter 'exit' command.	None	The shell terminates gracefully and displays exit message.	As Expected	PASS