Basic Shell Implementation
OS Project:

Explanation: building a small command-line shell (like bash or sh) that accepts a line of text, parses it, and runs commands. The shell must support running external programs, a few built-in commands (cd, exit), background jobs (&), input/output redirection (<, >), pipes (|), and basic signal handling so the shell behaves correctly with Ctrl-C and when children terminate.

What you're building:

You are creating a small program that:

- Shows a prompt (like $ or >).
- Lets the user type a command (like ls, cd, cat file.txt).
- Runs that command by creating a new process.
- Shows the output.
- Keeps running until the user types exit.

Shell must include:

- A command prompt.
- Command parsing.
- Built-in commands:
    - Cd.
    - exit.
- External command execution using:
    - fork().

    - exec().
    - wait().
- Background processes using &.
- Basic job cleanup using SIGCHLD.
- Input/output redirection (<, >).
- Piping using |.
- Signal handling for:
    - SIGINT.
    - SIGCHLD.

🟦 **Optional (NOT required, but allowed)**

These are extensions — not mandatory:

- Job control (fg, bg, process groups).
- Environment variables like $PATH.
- Command history.
- Arrow-key navigation (readline).

Project Structure:

```
python-shell-project/
 ├── main.py                # All members contribute
 ├── myshell/
 │   ├── __init__.py
 │   ├── input_handler.py      # MEMBER 1
 │   ├── parser.py             # MEMBER 1
 │   ├── builtins.py           # MEMBER 2
 │   ├── executor.py           # MEMBER 2
 │   ├── signal_handler.py     # MEMBER 3
 │   ├── redirector.py         # MEMBER 3
 │   └── piper.py              # MEMBER 3
```

Shell Project Build Instructions (3-Member Division)

MEMBER 1: INPUT & PARSING LAYER

Module 1: Input Handler (input_handler.py)

What to build: Function that displays prompt and reads user input

Instructions:

1. Import os module to get current directory
2. Create function read_input()
3. Get current working directory using os module
4. Format prompt string to show directory path

5. Use Python's input() to read user's typed command
6. Remove leading/trailing whitespace
7. Handle EOF error (Ctrl+D) by returning None
8. Handle other exceptions by printing error and returning empty string
9. Return the cleaned input string

Testing: Call function and verify prompt appears, input is returned correctly, Ctrl+D doesn't crash

Module 2: Command Parser (parser.py)

What to build: Class and function to break commands into structured data

Part A - Command Class:

1. Create class named Command
2. Initialize these attributes in constructor:
    - program (command name)
    - args (list of arguments)
    - background (boolean for & operator)
    - input_file (for < redirection)
    - output_file (for > redirection)
    - append_output (boolean for >> vs >)
    - pipe_to (reference to next command in pipeline)

Part B - Main Parser Function:

1. Create function parse_command(raw_input) that takes user input string
2. Return None if input is empty
3. Check if input ends with & - set background flag and remove it

4. Split input by | character to separate piped commands
5. For each segment between pipes, parse it individually
6. Create Command objects for each segment
7. Link Command objects together by setting pipe_to references
8. Set background flag on the last command if applicable
9. Return the first Command object

Part C - Single Command Parser:

1. Create helper function _parse_single_command(segment)
2. Create new Command object
3. Split segment into individual tokens (words)
4. Loop through tokens and identify operators:
    - If token is <: next token is input filename
    - If token is >: next token is output filename, append_output=False
    - If token is >>: next token is output filename, append_output=True
    - Otherwise: add to program or args list
5. First non-operator token becomes program name
6. Remaining tokens become arguments
7. Insert program name at start of args list (required for subprocess)
8. Return Command object

Testing: Parse various commands and verify all fields are set correctly

MEMBER 2: EXECUTION & BUILT-INS LAYER

Module 1: Built-in Commands (builtins.py)

What to build: Internal shell commands that modify shell state

Instructions:

1. Import os and sys modules
2. Create empty dictionary to store built-in commands
3. Create decorator function to register built-ins in dictionary
4. Create function is_builtin(command) to check if command exists in dictionary
5. Create function execute_builtin(command, args) to run built-in from dictionary

For CD command:

1. Create function decorated with built-in name "cd"
2. Check if directory argument provided
3. If not provided, get HOME directory from environment
4. Try to change directory using os.chdir()
5. Catch FileNotFoundError and print error message
6. Catch PermissionError and print error message
7. Return True if successful

For PWD command:

1. Create function decorated with built-in name "pwd"
2. Get current directory using os.getcwd()
3. Print the directory path
4. Return True

For EXIT command:

1. Create function decorated with built-in name "exit"
2. Check if exit code argument provided
3. Try to convert argument to integer
4. If conversion fails, print error
5. Call sys.exit() with exit code

For ECHO command:

1. Create function decorated with built-in name "echo"
2. Take all arguments after command name
3. Join them with spaces
4. Print the result
5. Return True

Testing: Execute each built-in individually and verify behavior

Module 2: Command Executor (executor.py)

What to build: Function that runs external programs

Main Executor:

1. Import subprocess, sys modules
2. Import built-in functions from previous module
3. Create function execute_command(command) that takes Command object
4. Check if command is empty - return if so
5. Check if command is built-in - execute and return if so
6. Check if command has pipe_to set - call Member 3's piper and return

7. Check if command has redirects - call Member 3's redirector and return
8. Otherwise call external executor helper

External Executor Helper:

1. Create function _execute_external(command)
2. Use subprocess.Popen to start new process
3. Pass command args list to Popen
4. Set stdin, stdout, stderr to PIPE
5. Check if background flag is set:
    - If yes: print process PID and return immediately
    - If no: wait for process to complete
6. Read stdout and stderr from process
7. Print stdout to console
8. Print stderr to error stream
9. Catch FileNotFoundError if command doesn't exist
10. Catch other exceptions and print error messages

Testing: Run built-in commands, external commands, background processes

MEMBER 3: ADVANCED FEATURES LAYER

Module 1: Signal Handler (signal_handler.py)

What to build: Handler to prevent Ctrl+C from killing shell

Instructions:

1. Import signal module
2. Create function setup_signals()
3. Register handler for SIGINT (Ctrl+C signal)
4. Create handler function sigint_handler(signum, frame)
5. In handler, print newline character
6. Don't call exit - just return to allow shell loop to continue

Note for Windows: Windows has limited signal support, SIGCHLD not available

Testing: Run shell, press Ctrl+C, verify shell doesn't exit

Module 2: I/O Redirector (redirector.py)

What to build: Function to redirect input/output to files

Instructions:

1. Import subprocess and sys modules
2. Create function execute_with_redirect(command)
3. Set default stdin to PIPE
4. Set default stdout to PIPE
5. Set default stderr to PIPE
6. Check if command has input_file:
    - Try to open file in read mode

- Set as stdin source
- Catch file not found error
- Catch permission error

7. Check if command has output_file:
    - Determine mode: 'a' for append, 'w' for write
    - Try to open file in appropriate mode
    - Set as stdout destination
    - Catch permission error
8. Create subprocess with Popen using file handles
9. If not background process, wait for completion
10. Read stdout and stderr
11. If output not redirected to file, print stdout
12. Always print stderr to console
13. Close any opened file handles in finally block
14. Catch and print any exceptions

Testing: Redirect input from file, redirect output to file, test append mode

Module 3: Pipeline Handler (piper.py)

What to build: Function to connect multiple commands with pipes

Instructions:

1. Import subprocess and sys modules
2. Create function execute_pipeline(command)
3. Create empty list to store process objects
4. Start with first command, loop through all piped commands

5. For each command in pipeline:
    - Set stdin default to PIPE
    - Set stdout default to PIPE
    - If not first command: connect stdin to previous process stdout
    - If last command and has output file: open file for stdout
    - Create subprocess with Popen
    - Add process to list
    - Move to next command via pipe_to reference
6. Wait for all processes to complete
7. Read stdout and stderr from each process
8. Only print output from last process
9. Print stdout if not redirected to file
10. Always print stderr
11. Close any opened file handles

Testing: Test simple two-command pipe, test three-command pipe, test pipe with redirect

INTEGRATION: MAIN FILE (main.py)

What to build: Main shell loop that ties everything together

Instructions:

1. Import sys module
2. Import read_input from Member 1's input_handler
3. Import parse_command from Member 1's parser
4. Import execute_command from Member 2's executor

5. Import setup_signals from Member 3's signal_handler
6. Create function main()
7. Call setup_signals at start
8. Print welcome message
9. Create infinite while loop
10. Inside loop:
    - Call read_input to get user command
    - Check if result is None (EOF) - break loop
    - Skip if input is empty
    - Call parse_command to convert to Command object
    - Skip if parsing returns None
    - Call execute_command to run the command
11. Wrap loop in try-except blocks:
    - Catch KeyboardInterrupt - print newline and continue
    - Catch EOFError - break loop
    - Catch general exceptions - print error and continue
12. Print goodbye message after loop ends
13. Add main guard: if __name__ == "__main__":
14. Call main() function

Testing: Run shell and test all features together