System Requirements Specification Index

For

File System Explorer

Version 1.0

IIHT Pvt. Ltd.

fullstack@iiht.com

TABLE OF CONTENTS

File System Explorer

System Requirements Specification

1 Project Abstract

TechSolutions Inc. needs a simple file system explorer tool that can analyze directory structures and search for specific files. This assignment focuses on implementing recursive functions to traverse a simulated file system structure, where directories are represented as dictionaries and files as integers (representing file sizes in bytes).

2 Business Requirements:

Screen Name	Console input screen
Problem Statement	1. System must navigate dictionary-based directory structures using recursive traversal. 2. Tool must locate files by name or extension. 3. System must generate basic file distribution reports by type and size.

3 CONSTRAINTS

3.1 INPUT REQUIREMENTS

1. Directory Structure:
    - Directories must be represented as nested dictionaries.
    - Files must be represented as key-value pairs (filename: size in bytes).
2. File Naming:
    - File extensions must be standard (.txt, .jpg, .png, etc.).
    - - File names must be case-insensitive during search operations.

3.2 FUNCTION DEFINITION REQUIREMENTS

1. Recursive Structure:
    - Each function must include proper base case handling.
    - Each function must implement recursive traversal of the simulated file system.
    - No external libraries should be used (only built-in Python functions).

1. Docstrings:
    - Each function must include a docstring describing:
        - Purpose of the function
        - Parameters and return values
        - Base and recursive case descriptions

3. Parameter Types:
    - Functions must accept simulated directory structures (dictionaries).
    - Search functions must accept search criteria as strings.

3.3 OPERATIONS CONSTRAINTS

1. **Directory Traversal:**

- `list_all_files` must recursively find all files in a directory structure.
- Function must handle nested directories of arbitrary depth.

2. File Search:
   - `find_by_extension` must locate all files with specific extensions.
   - `find_by_name` must locate files matching a name pattern.
3. File Analysis:
   - `calculate_directory_size` must sum file sizes recursively.
   - `count_files_by_type` must count files by extension.
   - `find_largest_files` must identify the N largest files in the structure.
4. Function Call Composition:
   - At least one function must call another function and use its return value.
   - Example: `total_size = calculate_directory_size(list_all_files(directory))`

3.4 Output Constraints

1. Display Format:
   - File paths should be displayed as relative paths.
   - File sizes must be formatted in human-readable format (KB, MB, GB).

2. Output Format:
   - "== FILE SYSTEM EXPLORER =="
   - "Directory Summary" showing total files and size
   - "File Type Distribution" showing count by extension
   - "Search Results" showing files matching criteria
   - "Largest Files" showing top N files by size

4. TEMPLATE CODE STRUCTURE:

1. Directory Traversal Functions:

- `list_all_files(directory, file_system=None, path_prefix="")` - returns all files in the directory tree.
- `calculate_directory_size(directory, file_system=None)` - computes total size of all files.

2. File Search Functions:

- `find_by_extension(directory, extension, file_system=None, path_prefix="")` - finds files with specific extension.
- `find_by_name(directory, pattern, file_system=None, path_prefix="")` - finds files matching name pattern.

3. File Analysis Functions:

- `count_files_by_type(directory, file_system=None)` - counts files by extension.
- `find_largest_files(directory, n, file_system=None)` - finds N largest files.

4. Helper Functions:

- `create_sample_file_system()` - creates a sample file system structure for testing.
- `format_file_size(size_bytes)` - converts bytes to human-readable format.

5. Main Program Function:

- `main()` - demonstrates all functions and produces formatted output.

5. DETAILED FUNCTION IMPLEMENTATION STRUCTURE

5.1 Core Directory Traversal Functions

1. Write a Python function to recursively list all files in a directory structure. Define: `list_all_files(directory, file_system=None, path_prefix="")`

The function should:

- Accept three parameters: `directory` (string for target directory), `file_system` (dict representing file system), `path_prefix` (string for path construction)
- Use sample file system if none provided: `if file_system is None: file_system = create_sample_file_system()`
- Validate input is dictionary: `if not isinstance(file_system, dict): raise TypeError("File system must be a dictionary")`
- Initialize result list: `files = []`
- Navigate to target directory if specified:
    - Split directory path: `parts = directory.split("/") if directory else []`
    - Navigate through path: `current = file_system`
    - Use loop: `for part in parts: if part and part in current: current = current[part]`
- **Base case**: When value is integer (file): add to results with full path
- **Recursive case**: When value is dictionary (directory): call `list_all_files` recursively
- Iterate through current directory: `for name, value in current.items():`
- Check if item is file: `if isinstance(value, int):`
- Construct file path: `file_path = f"{path_prefix}{name}" if not path_prefix else f"{path_prefix}/{name}"`
- Add to results: `files.append(file_path)`
- Check if item is directory: `elif isinstance(value, dict):`
- Construct directory path: `new_prefix = f"{path_prefix}{name}" if not path_prefix else f"{path_prefix}/{name}"`
- Recursive call: `files.extend(list_all_files("", value, new_prefix))`
- Return complete file list: `return files`
- Example: list_all_files("Documents") should return all files in Documents directory and subdirectories

2. Write a Python function to recursively calculate directory size. Define: `calculate_directory_size(directory, file_system=None)`

The function should:

- Accept two parameters: `directory` (string for target directory), `file_system` (dict representing file system)
- Use sample file system if none provided: `if file_system is None: file_system = create_sample_file_system()`
- Validate input is dictionary: `if not isinstance(file_system, dict): raise TypeError("File system must be a dictionary")`
- Navigate to target directory if specified:
  - Split directory path: `parts = directory.split("/") if directory else []`
  - Navigate through path: `current = file_system`
  - Use loop: `for part in parts: if part and part in current: current = current[part]`
  - Return 0 if path not found: `else: return 0`
- Initialize total size: `total_size = 0`
- **Base case**: When value is integer (file): add its size to total
- **Recursive case**: When value is dictionary (directory): recursively calculate size of contents
- Iterate through current directory: `for name, value in current.items():`
- Check if item is file: `if isinstance(value, int):`
- Add file size: `total_size += value`
- Check if item is directory: `elif isinstance(value, dict):`
- Recursive call: `total_size += calculate_directory_size("", value)`
- Return total size: `return total_size`
- Example: calculate_directory_size("Documents/Projects") should return sum of all file sizes in Projects directory

5.2 File Search Functions

3. Write a Python function to recursively find files by extension. Define: `find_by_extension(directory, extension, file_system=None, path_prefix="")`

The function should:

- Accept four parameters: `directory` (string), `extension` (string), `file_system` (dict), `path_prefix` (string)
- Use sample file system if none provided: `if file_system is None: file_system = create_sample_file_system()`
- Validate inputs: `if not isinstance(file_system, dict): raise TypeError("File system must be a dictionary")`

- Convert extension to string and lowercase: `extension = str(extension).lower()`
- Initialize result list: `matching_files = []`
- Navigate to target directory if specified:
    - Split directory path: `parts = directory.split("/") if directory else []`
    - Navigate through path: `current = file_system`
    - Use loop: `for part in parts: if part and part in current: current = current[part]`
- **Base case**: When value is integer (file): check if filename ends with extension
- **Recursive case**: When value is dictionary (directory): search recursively in subdirectories
- Iterate through current directory: `for name, value in current.items():`
- Check if item is file: `if isinstance(value, int):`
- Extract file extension: `file_ext = name.split('.')[-1].lower() if '.' in name else ""`
- Check extension match: `if file_ext == extension:`
- Construct file path: `file_path = f"{path_prefix}{name}" if not path_prefix else f"{path_prefix}/{name}"`
- Add to results: `matching_files.append(file_path)`
- Check if item is directory: `elif isinstance(value, dict):`
- Construct directory path: `new_prefix = f"{path_prefix}{name}" if not path_prefix else f"{path_prefix}/{name}"`
- Recursive call: `matching_files.extend(find_by_extension("", extension, value, new_prefix))`
- Return matching files: `return matching_files`
- Example: find_by_extension("", "pdf") should return all PDF files in the file system

4. Write a Python function to recursively find files by name pattern. Define: `find_by_name(directory, pattern, file_system=None, path_prefix="")`

The function should:

- Accept four parameters: `directory` (string), `pattern` (string), `file_system` (dict), `path_prefix` (string)
- Use sample file system if none provided: `if file_system is None: file_system = create_sample_file_system()`
- Validate inputs: `if not isinstance(file_system, dict): raise TypeError("File system must be a dictionary")`
- Convert pattern to string and lowercase: `pattern = str(pattern).lower()`

- Initialize result list: `matching_files = []`
- Navigate to target directory if specified:
  - Split directory path: `parts = directory.split("/") if directory else []`
  - Navigate through path: `current = file_system`
  - Use loop: `for part in parts: if part and part in current: current = current[part]`
- **Base case**: When value is integer (file): check if filename contains pattern
- **Recursive case**: When value is dictionary (directory): search recursively in subdirectories
- Iterate through current directory: `for name, value in current.items():`
- Check if item is file: `if isinstance(value, int):`
- Check pattern match: `if pattern in name.lower():`
- Construct file path: `file_path = f"{path_prefix}{name}" if not path_prefix else f"{path_prefix}/{name}"`
- Add to results: `matching_files.append(file_path)`
- Check if item is directory: `elif isinstance(value, dict):`
- Construct directory path: `new_prefix = f"{path_prefix}{name}" if not path_prefix else f"{path_prefix}/{name}"`
- Recursive call: `matching_files.extend(find_by_name("", pattern, value, new_prefix))`
- Return matching files: `return matching_files`
- Example: find_by_name("", "project") should return all files containing "project" in their name

5.3 File Analysis Functions

5. Write a Python function to recursively count files by type. Define: `count_files_by_type(directory, file_system=None)`

The function should:

- Accept two parameters: `directory` (string), `file_system` (dict)
- Use sample file system if none provided: `if file_system is None: file_system = create_sample_file_system()`
- Validate input: `if not isinstance(file_system, dict): raise TypeError("File system must be a dictionary")`
- Initialize result dictionary: `type_counts = {}`
- Navigate to target directory if specified:

    - Split directory path: `parts = directory.split("/") if directory else []`
    - Navigate through path: `current = file_system`
    - Use loop: `for part in parts: if part and part in current: current = current[part]`
- **Base case**: When value is integer (file): extract extension and increment count
- **Recursive case**: When value is dictionary (directory): merge counts from subdirectories
- Iterate through current directory: `for name, value in current.items():`
- Check if item is file: `if isinstance(value, int):`
- Extract file extension: `extension = name.split('.')[-1].lower() if '.' in name else "no_extension"`
- Increment count: `type_counts[extension] = type_counts.get(extension, 0) + 1`
- Check if item is directory: `elif isinstance(value, dict):`
- Recursive call: `sub_counts = count_files_by_type("", value)`
- Merge dictionaries: `for ext, count in sub_counts.items(): type_counts[ext] = type_counts.get(ext, 0) + count`
- Return count dictionary: `return type_counts`
- Example: count_files_by_type("") should return {"pdf": 4, "docx": 2, "txt": 2, ...}

6. Write a Python function to recursively find largest files. Define: `find_largest_files(directory, n, file_system=None)`

The function should:

- Accept three parameters: `directory` (string), `n` (integer), `file_system` (dict)
- Validate n parameter: `try: n = int(n)` with exception handling
- Handle negative n: `if n < 0: return []`
- Use sample file system if none provided: `if file_system is None: file_system = create_sample_file_system()`
- Validate input: `if not isinstance(file_system, dict): raise TypeError("File system must be a dictionary")`
- Create helper function to collect all files: `def collect_all_files(current_dict, current_path=""):`
- Helper function logic:
    - Initialize files list: `all_files = []`
    - Iterate through directory: `for name, value in current_dict.items():`
    - **Base case**: If file: `if isinstance(value, int):`
    - Construct path: `file_path = f"{current_path}{name}" if not current_path else f"{current_path}/{name}"`

    - Add tuple: `all_files.append((file_path, value))`
    - **Recursive case**: If directory: `elif isinstance(value, dict):`
    - Construct path: `new_path = f"{current_path}{name}" if not current_path else f"{current_path}/{name}"`
    - Recursive call: `all_files.extend(collect_all_files(value, new_path))`
    - Return files: `return all_files`
- Navigate to target directory if specified:
    - Split directory path: `parts = directory.split("/") if directory else []`
    - Navigate through path: `current = file_system`
    - Use loop: `for part in parts: if part and part in current: current = current[part]`
- Collect all files: `all_files = collect_all_files(current)`
- Sort by size (descending): `all_files.sort(key=lambda x: x[1], reverse=True)`
- Return top n files: `return all_files[:n]`
- Example: find_largest_files("", 5) should return 5 tuples of (path, size) for largest files

5.4 Utility Functions

7. Write a Python function to format file sizes. Define: `format_file_size(size_bytes)`

The function should:

- Accept one parameter: `size_bytes` (numeric value)
- Validate input: `try: size = float(size_bytes)` with exception handling for TypeError
- Handle negative sizes gracefully: return appropriate format
- Define size units: `units = ["B", "KB", "MB", "GB", "TB"]`
- Initialize unit index: `unit_index = 0`
- Convert to appropriate unit using loop: `while size >= 1024 and unit_index < len(units) - 1:`
- Divide by 1024: `size /= 1024`
- Increment index: `unit_index += 1`
- Format result: `if unit_index == 0: return f"{int(size)} {units[unit_index]}"`
- For larger units: `else: return f"{size:.2f} {units[unit_index]}"`
- Example: format_file_size(1024) should return "1.00 KB"

8. Write a Python function to create sample file system. Define: `create_sample_file_system()`

The function should:

- Take no parameters
- Return dictionary structure matching test expectations:

```
return {
   "Documents": {
      "Projects": {
         "project1.docx": 2500000,
         "project2.docx": 1800000,
         "notes.txt": 15000,
         "data.csv": 350000,
      },
      "Personal": {
         "resume.pdf": 520000,
         "budget.xlsx": 480000,
         "Photos": {
            "vacation.jpg": 3500000,
            "family.jpg": 2800000,
            "graduation.png": 4200000,
         }
      },
      "report.pdf": 750000,
   },
   "Downloads": {
      "program.exe": 15000000,
      "Library": {
         "book1.pdf": 12000000,
         "book2.pdf": 9500000,
      },
      "song.mp3": 8000000,
      "video.mp4": 35000000,
   },
   "temp.txt": 2000,
}
```

5.5 Main Program Structure

9. Write a Python main program for file system exploration. Define: `main()` function

The function should:

- Print welcome message: `print("===== FILE SYSTEM EXPLORER =====")`
- Create file system: `file_system = create_sample_file_system()`
- **Directory Summary Section:**
  - Print header: `print("\n----- DIRECTORY SUMMARY -----")`
  - Get all files: `all_files = list_all_files("", file_system)`
  - Calculate total size: `total_size = calculate_directory_size("", file_system)`
  - Format total size: `formatted_size = format_file_size(total_size)`
  - Display results: `print(f"Total files: {len(all_files)}")` and `print(f"Total size: {formatted_size}")`
- **File Type Distribution Section:**
  - Print header: `print("\n----- FILE TYPE DISTRIBUTION -----")`
  - Get type counts: `type_counts = count_files_by_type("", file_system)`
  - Display each type: `for file_type, count in type_counts.items(): print(f"{file_type}: {count} files")`
- **Search by Extension Section:**
  - Print header: `print("\n----- SEARCH BY EXTENSION -----")`
  - Find PDF files: `pdf_files = find_by_extension("", "pdf", file_system)`
  - Display results: `print(f"PDF files found: {len(pdf_files)}")` and list each file
- **Search by Name Section:**
  - Print header: `print("\n----- SEARCH BY NAME -----")`
  - Find project files: `project_files = find_by_name("", "project", file_system)`
  - Display results: `print(f"Project files found: {len(project_files)}")` and list each file
- **Largest Files Section:**
  - Print header: `print("\n----- LARGEST FILES -----")`
  - Find top 5 largest: `largest_files = find_largest_files("", 5, file_system)`
  - Display each file: `for path, size in largest_files: print(f"{path}: {format_file_size(size)}")`
- **Specific Directory Analysis Section:**
  - Print header: `print("\n----- SPECIFIC DIRECTORY ANALYSIS -----")`

  - Analyze Photos directory: `photos_files = list_all_files("Documents/Personal/Photos", file_system)`
  - Calculate Photos size: `photos_size = calculate_directory_size("Documents/Personal/Photos", file_system)`
  - Display Photos analysis: file count and total size
- Include program entry point: `if __name__ == "__main__": main()`

5.6 Implementation Requirements

Key Technical Requirements:

Recursive Structure Requirements:

- **list_all_files()** must use recursion to traverse nested directories
- **calculate_directory_size()** must use recursion to sum sizes from all subdirectories
- **find_by_extension()** must use recursion to search through directory tree
- **find_by_name()** must use recursion to search through directory tree
- **count_files_by_type()** must use recursion and merge results from subdirectories
- **find_largest_files()** must use helper function with recursion to collect all files

Base and Recursive Cases:

- **Base case**: When encountering a file (integer value) - process the file
- **Recursive case**: When encountering a directory (dict value) - call function recursively
- All functions must handle both cases appropriately
- Navigation logic must handle directory paths by splitting and traversing

Path Construction Requirements:

- Use proper path separators (forward slashes)
- Handle empty path prefixes correctly
- Construct full file paths relative to root
- Avoid double separators in paths

Data Validation Requirements:

- Validate file_system parameter is a dictionary
- Handle non-existent directory paths gracefully
- Type checking for numeric parameters (n in find_largest_files)
- Exception handling for invalid inputs

Function Composition Requirements:

- At least one function must call another function and use its return value
- Example: main() function calls multiple other functions
- Helper functions should be used appropriately
- Results from one function should be input to another where logical

Return Value Requirements:

- list_all_files(): list of file path strings
- calculate_directory_size(): integer representing total bytes
- find_by_extension(): list of file path strings matching extension
- find_by_name(): list of file path strings containing pattern
- count_files_by_type(): dictionary with extensions as keys, counts as values
- find_largest_files(): list of tuples (path, size) sorted by size descending
- format_file_size(): string with human-readable size format

6. EXECUTION STEPS TO FOLLOW:

1. Implement the required recursive functions according to specifications.

2. Test each function with the provided sample file system structure.

3. Format the output according to the requirements.

4. Ensure all recursive functions work correctly with different directory depths.

5. Run the main function to demonstrate all functionality.

Execution Steps to Follow:

- All actions like build, compile, running application, running test cases will be through Command Terminal.

- To open the command terminal the test takers, need to go to Application menu (Three horizontal lines at left top) -> Terminal -> New Terminal

- This editor Auto Saves the code

- If you want to exit(logout) and continue the coding later anytime (using Save & Exit option on Assessment Landing Page) .

- These are time bound assessments the timer would stop if you logout and while logging in back using the same credentials the timer would resume from the same time it was stopped from the previous logout.

- To launch application: python3 filename.py

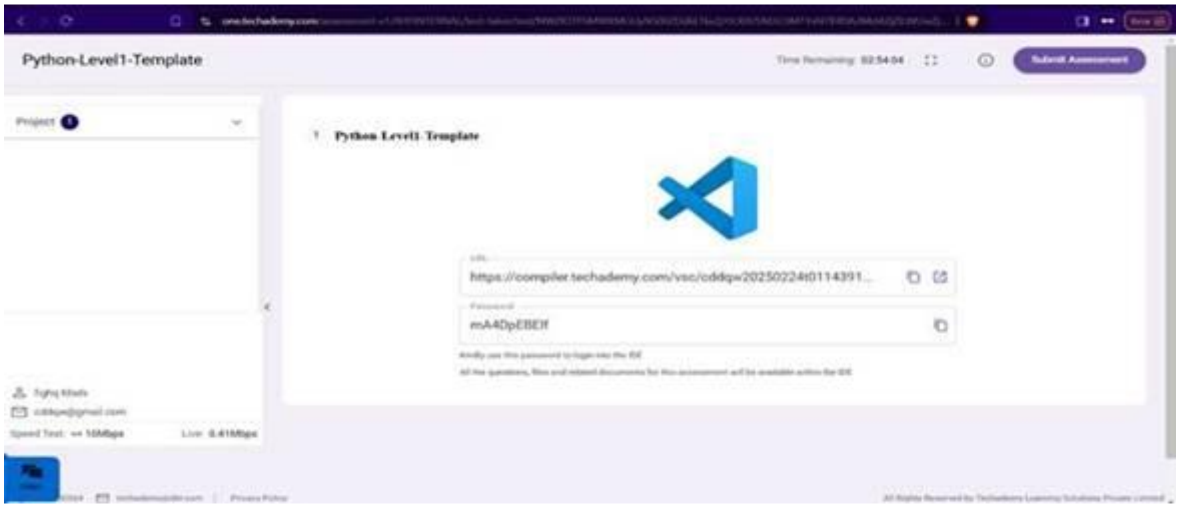
- To run Test cases: python3 -m unittest

<u>Screen shot to run the program</u>

To run the application

python3 filename.py

To run the testcase **python3 -m unittest**



URL: https://compiler.techademy.com/vsc/cddqw20250224t0114391...

Password: mA4DpEBEff

- Once you are done with development and ready with submission, you may navigate to the previous tab and submit the workspace. It is mandatory to click on “Submit Assessment” after you are done with code.