WARP.md

This file provides guidance to WARP (warp.dev) when working with code in this repository.

Project Overview

DFA (Disk-Folder-File Analyzer) is a comprehensive command-line tool for analyzing file types and sizes in directories. It recursively scans directories, categorizes files by extension, and provides detailed statistics including file counts, total sizes, and identifies largest/smallest files per category.

Common Development Commands

Running the Application

```
# Basic analysis using default configuration
python3 main.py

# Analyze specific directory
python3 main.py /path/to/directory

# Use custom configuration
python3 main.py -c custom_config.json

# Sort by file count, show top 20 extensions, save output
python3 main.py -s count -m 20 -o results.txt

# Include hidden files with verbose logging
python3 main.py --show-hidden -v

# Debug mode with raw byte sizes
python3 main.py --debug --raw-sizes
```

Testing and Validation

```
# Compile all Python files to check for syntax errors
python3 -m py_compile *.py

# Test individual modules with standalone functionality
python3 config.py            # Test configuration loading
python3 scanner.py           # Test scanner module with basic functionality
python3 scanner.py /path     # Test scanner on specific directory
python3 stats.py             # Test statistics calculation
python3 stats.py /path       # Test stats calculation on specific directory
python3 output.py            # Test output formatting
```

Development Workflow

```
# Check for syntax errors across all modules
python3 -m py_compile *.py

# Run with different log levels for debugging
python3 main.py --debug --log-file debug.log

# Test with small directory first (current directory)
python3 main.py . -v

# Test with extension filtering enabled
python3 main.py --extension-filter -c config.json

# Test interrupt handling (use Ctrl+C during scan)
python3 main.py /large/directory --debug

# Performance testing on large directories
time python3 main.py /large/directory --no-summary -s size
```

Architecture Overview

Modular Design

The application follows a clean modular architecture with single-responsibility components:

- `main.py`: CLI interface, argument parsing, signal handling, and orchestration
- `config.py`: Configuration management with JSON validation and defaults
- `scanner.py`: Recursive directory traversal with path sanitization and filtering
- `stats.py`: Statistics calculation engine with streaming data processing
- `output.py`: Professional table formatting and display management

Key Architectural Patterns

Generator-Based Processing: Uses Python generators for memory-efficient processing of large directories (tested with 59K+ files)

Streaming Statistics: Statistics are calculated as files are discovered, not stored in memory

Graceful Error Handling: Continues processing despite individual file access failures, with comprehensive logging

Configuration-Driven: All behavior configurable via JSON with runtime CLI overrides

Data Flow

1. **Configuration Loading**: `ConfigManager` loads `config.json`, validates values, and provides defaults
2. **Scanner Initialization**: `DirectoryScanner` configured with filters and exclusion rules
3. **File Discovery**: Scanner yields `FileInfo` dataclass objects via generator pattern from

`scan_directory()`

4. **Statistics Processing**: `StatisticsCalculator.process_files_streaming()` consumes generator and builds `ExtensionStats` per extension
5. **Result Compilation**: Complete `ScanStatistics` object created with sorted extensions and overall metrics
6. **Output Generation**: `OutputManager` formats results into tables and summary sections
7. **Display/Export**: Results shown on console and optionally saved to file

Key Objects: `FileInfo` → `ExtensionStats` → `ScanStatistics` → Formatted Output

Configuration System

The `config.json` file controls application behavior:

```
{
  "starting_directory": "/home/user",
  "extension_list": [".txt", ".pdf", ".doc", ".docx", ".jpg", ".png", ".mp4", ".mp3"],
  "use_extension_list": false,
  "exclude_hidden_files": true,
  "human_readable_sizes": true,
  "log_level": "INFO",
  "log_file": "dfa.log"
}
```

Configuration Precedence: CLI arguments > config.json > defaults

Debugging and Development Patterns

Module Testing: Each module (`config.py`, `scanner.py`, `stats.py`, `output.py`) includes standalone test functionality when run directly with `python3 <module>.py`. The scanner and stats modules accept directory arguments for testing.

Log Analysis: Use `--debug --log-file debug.log` to capture detailed execution flow. The application logs progress every 100 directories and 1000 files during scanning.

Signal Handling: The application implements graceful shutdown on SIGINT (Ctrl+C) and SIGTERM. Test this by starting a large directory scan and interrupting it.

Memory Profiling: The generator-based architecture processes files one-by-one without loading all file data into memory. This can be verified on large directories (tested with 59K+ files).

Key Components Deep Dive

DirectoryScanner (`scanner.py`)

- **Path Sanitization**: All input paths validated and normalized
- **Permission Handling**: Gracefully handles access denied scenarios
- **Filter Support**: Hidden files and extension-based filtering
- **Progress Tracking**: Reports scan progress every 100 directories/1000 files

- **Memory Efficient**: Uses `os.walk` with generator pattern

StatisticsCalculator (`stats.py`)

- **FileInfo**: Dataclass storing path, name, extension, size, hidden status
- **ExtensionStats**: Per-extension statistics with largest/smallest tracking
- **ScanStatistics**: Overall statistics with sorting capabilities
- **Streaming Processing**: Calculates stats without storing all file data

OutputManager (`output.py`)

- **Dynamic Table Formatting**: Calculates column widths automatically
- **Human-Readable Sizes**: Converts bytes to KB/MB/GB format
- **Export Functionality**: Save results to text files
- **Multiple Display Modes**: Summary, table-only, detailed extension info

Performance Characteristics

- **Memory**: Minimal footprint using generators and streaming
- **Scalability**: Successfully tested with 59,106 files (18.1 GB)
- **Speed**: ~13 seconds for enterprise-scale directories
- **Error Recovery**: Continues despite individual file failures
- **Interrupt Safety**: Clean shutdown on Ctrl+C

Development Guidelines

Error Handling Strategy

- Use logging instead of print statements for debug information
- Always validate and sanitize input paths
- Graceful degradation - continue processing despite individual failures
- Provide meaningful error messages to users

Adding New Features

1. **Scanner Features**: Modify `DirectoryScanner` class, maintain generator pattern
2. **Statistics Features**: Extend `ExtensionStats` or `ScanStatistics` classes
3. **Output Features**: Add methods to `OutputManager`, maintain table formatting consistency
4. **Configuration**: Add to `DEFAULT_CONFIG` in `config.py` and update validation

Code Style

- Follow existing docstring patterns with type hints
- Use dataclasses for structured data (`FileInfo`, `ExtensionStats`)
- Maintain modular separation - each file has single responsibility
- Log significant events at appropriate levels (DEBUG/INFO/WARNING/ERROR)

Testing Strategy

The application has been tested with:

- Small directories (10+ files)
- Large production directories (59K+ files, 18GB)
- Permission denied scenarios
- Invalid configurations
- Interrupt handling (Ctrl+C)
- Various CLI argument combinations

When making changes, test with both small and large directories to ensure performance and memory efficiency.