```
Start coding or generate with AI.
```

Quick Start Guide for Agentic IDE

This guide helps you quickly integrate the Video Sorter application into your project.

📦 What You're Getting

This is a complete, production-ready desktop application with:

- ✅ **Full video analysis** - Resolution, orientation, FPS, device metadata, camera info, GPS
- ✅ **Smart filtering** - Filter by Make, Model, Camera, GPS (AND logic)
- ✅ **Professional GUI** - Built with PySide6 (Qt)
- ✅ **Background processing** - Won't freeze during long operations
- ✅ **Export capabilities** - CSV and TXT formats
- ✅ **Drag & drop** - User-friendly interface
- ✅ **Fully commented** - Every function explained

🚀 Quick Integration Steps

Step 1: Project Structure

Create this folder structure in your project:

```
your_project/
├── video_classifier.py    ← Core video analysis engine
├── video_sorter.py        ← GUI application
└── README.md              ← Full documentation (optional)
```

Step 2: Required Dependencies

Add to your `requirements.txt`:

```
PySide6>=6.0.0
```

Install with:

```
pip install PySide6
```

Step 3: System Dependency

Ensure FFmpeg is installed on the system:

macOS:

```
brew install ffmpeg
```

Linux:

```
sudo apt-get install ffmpeg
```

Windows:

```
choco install ffmpeg
```

Verification:

```
ffprobe -version
```

Step 4: Run the Application

```
python video_sorter.py
```

That's it! The GUI will launch.

📋 File Breakdown

video_classifier.py (Core Engine)

Purpose: Analyzes video files and extracts metadata

Main Function:

```
from video_classifier import classify_video

result = classify_video("path/to/video.mp4")
# Returns: {
```

```
#   'resolution': '4K',
#   'orientation': 'V',
#   'framerate_category': 60,
#   'make': 'Apple',
#   'model': 'iPhone 14 Pro',
#   'has_camera': True,
#   'has_gps': True,
#   'success': True
# }
```

Dependencies:

- `subprocess` (built-in)
- `json` (built-in)
- `ffprobe` (external - part of FFmpeg)

Key Features:

- Handles video rotation (iPhone portrait videos)
- Extracts device metadata (Make, Model)
- Detects camera metadata presence
- Detects GPS data presence
- Graceful error handling

---

video_sorter.py (GUI Application)

Purpose: Desktop application with filtering and export

Main Components:

1. **VideoProcessorThread** - Background processing class
    - Prevents GUI freezing
    - Processes videos one at a time
    - Sends progress updates via Qt signals
2. **VideoSorterWindow** - Main window class
    - Filter controls (Make, Model, Camera, GPS)
    - Drag & drop zone
    - Results table with sorting
    - Log viewer
    - Export buttons (CSV, TXT)

Dependencies:

- `PySide6` (Qt for Python)
- `video_classifier.py` (our engine)

Key Features:

- Real-time filtering with AND logic
- Background processing (no GUI freeze)
- Live progress updates
- Drag & drop support
- CSV/TXT export of filtered results

🔍 How the Filtering Works

Filter Logic (AND Condition)

```
# Example filter scenarios:

# Scenario 1: No filters active
# → Shows ALL videos

# Scenario 2: "Filter by Make" checked
# → Shows only videos that HAVE a manufacturer (Apple, Samsung, etc.)

# Scenario 3: "Has Camera Metadata" checked
# → Shows only videos that HAVE camera information

# Scenario 4: "Filter by Make" + "Has Camera Metadata" both checked
# → Shows only videos that HAVE manufacturer AND camera data
# (both conditions must be true)

# Scenario 5: All filters checked
# → Shows only videos that have:
#    - Manufacturer info (Make)
#    - Model info
#    - Camera metadata
#    - GPS data
# (all four conditions must be true)
```

Implementation

```
def _video_passes_filters(self, video: Dict) -> bool:
    # No filters? Show everything
    if not any([filter1, filter2, filter3, filter4]):
        return True

    # Check each active filter (AND logic)
    if filter_make.isChecked() and not video['make']:
        return False  # Fail: No manufacturer

    if filter_model.isChecked() and not video['model']:
        return False  # Fail: No model

    if filter_camera.isChecked() and not video['has_camera']:
        return False  # Fail: No camera data

    if filter_gps.isChecked() and not video['has_gps']:
        return False  # Fail: No GPS data

    return True  # Pass: All active filters satisfied
```

🎯 Usage Examples

Example 1: Find All Apple Videos

```
1. Open application
2. Drag folder or click "Select Folder"
3. Check "Filter by Make"
4. View results: Only videos with manufacturer info
5. Export to CSV
```

Example 2: Find iPhone Videos with GPS

```
1. Open application
2. Select folder with videos
3. Check "Filter by Make" (finds devices with manufacturer)
4. Check "Filter by Model" (finds devices with model info)
5. Check "Has GPS Data"
6. View results: Only videos with make + model + GPS
7. Export to TXT
```

Example 3: Professional Camera Footage

```
1. Open application
2. Select mixed footage folder
3. Check "Has Camera Metadata" only
4. View results: Separates pro camera from phone footage
5. Export to CSV for asset management
```

🔧 Customization Points

Adding New Filters

Location: `video_sorter.py` → `_create_filter_section()`

```
# Add a new checkbox filter
self.filter_resolution = QCheckBox("4K Videos Only")
self.filter_resolution.stateChanged.connect(self._apply_filters)
layout.addWidget(self.filter_resolution)

# Update filter logic
def _video_passes_filters(self, video: Dict) -> bool:
    # ... existing code ...

    if self.filter_resolution.isChecked() and video['resolution'] != '4K'
        return False

    return True
```

Adding New Metadata Fields

Location: `video_classifier.py` → `_get_video_metadata()`

```
# Extract additional metadata
duration = float(data['format'].get('duration', 0))
bitrate = int(data['format'].get('bit_rate', 0))

return {
    'success': True,
```

```
    # ... existing fields ...
    'duration': duration,
    'bitrate': bitrate
}
```

Custom Export Formats

Location: `video_sorter.py` → Add new export method

```
def _export_json(self, videos: List[Dict], filepath: str):
    import json
    with open(filepath, 'w') as f:
        json.dump(videos, f, indent=2)
```

🪛 Common Integration Issues

Issue 1: "Cannot import video_classifier"

Cause: Files not in same folder

Solution: Ensure both `.py` files are in the same directory

```
✓ Correct:
  project/
  ├── video_classifier.py
  └── video_sorter.py

✗ Wrong:
  project/
  ├── src/
  │   └── video_classifier.py
  └── video_sorter.py
```

Issue 2: "ffprobe not found"

Cause: FFmpeg not installed

Solution: Install FFmpeg (see Step 3 above)

Verify:

```
ffprobe -version  # Should show version info
which ffprobe     # Should show path (macOS/Linux)
where ffprobe     # Should show path (Windows)
```

Issue 3: GUI Won't Launch

Cause: PySide6 not installed

Solution:

```
pip install --upgrade PySide6
python -c "from PySide6.QtWidgets import QApplication"
# Should run without error
```

Issue 4: Processing Freezes

Cause: Background thread not working

Debugging:

```
# Add to VideoProcessorThread.run()
print(f"Thread started: {threading.current_thread().name}")
```

📊 Testing Checklist

Before deployment, test these scenarios:

- ▢ **Small folder** (1-5 videos) - Quick sanity check
- ▢ **Large folder** (100+ videos) - Performance test
- ▢ **Mixed formats** (.mp4, .mov, .avi) - Format compatibility
- ▢ **Rotated videos** (iPhone portrait) - Rotation handling
- ▢ **No metadata videos** (screen recordings) - Graceful handling
- ▢ **All filters active** - AND logic works correctly
- ▢ **CSV export** - Opens correctly in Excel
- ▢ **TXT export** - Human-readable format
- ▢ **Drag & drop** - Works as expected
- ▢ **Stop during processing** - Clean cancellation

🎓 Understanding the Architecture

Why Background Threading?

```
# ❌ BAD: Processing in main thread
for video in videos:
    result = classify_video(video)  # Takes 1-5 seconds
    # GUI is FROZEN during this time
    # User can't click anything
    # No progress updates visible

# ✅ GOOD: Processing in background thread
class VideoProcessorThread(QThread):
    def run(self):
        for video in videos:
            result = classify_video(video)
            self.result.emit(result)  # Send to GUI safely
            # GUI remains responsive
            # User sees live updates
```

Why Qt Signals?

```
# ❌ BAD: Direct GUI updates from thread
class VideoProcessor(QThread):
    def run(self):
        result = classify_video(video)
        self.table.addRow(result)  # CRASH! Not thread-safe

# ✅ GOOD: Use signals
class VideoProcessor(QThread):
    result = Signal(dict)  # Define signal

    def run(self):
        result = classify_video(video)
        self.result.emit(result)  # Thread-safe

# In main window:
thread.result.connect(self._add_row)  # Qt handles thread safety
```

📚 API Reference

classify_video(file_path: str) → Dict

Parameters:

- `file_path` (str): Path to video file

Returns: Dictionary with keys:

- `success` (bool): Whether analysis succeeded
- `resolution` (str): '4K', '1080p', '720p', 'HD', 'SD'
- `orientation` (str): 'V' (vertical) or 'W' (wide)
- `framerate_category` (int): 30 or 60
- `actual_fps` (float): Exact framerate (e.g., 59.94)
- `width` (int): Display width in pixels
- `height` (int): Display height in pixels
- `make` (str | None): Device manufacturer
- `model` (str | None): Device model
- `has_camera` (bool): Has camera metadata?
- `has_gps` (bool): Has GPS data?
- `error` (str | None): Error message if failed

Example:

```
result = classify_video("video.mp4")
if result['success']:
    print(result['resolution'])  # "4K"
    print(result['make'])        # "Apple"
else:
    print(result['error'])       # "Could not read video"
```

💡 Pro Tips

Tip 1: Batch Processing Script

Create a standalone script for automation:

```
from video_classifier import classify_video
import json

videos = ["video1.mp4", "video2.mov"]
```

```
results = []

for video in videos:
    result = classify_video(video)
    if result['success']:
        results.append({
            'file': video,
            'resolution': result['resolution'],
            'make': result['make']
        })

# Save results
with open('analysis.json', 'w') as f:
    json.dump(results, f, indent=2)
```

Tip 2: Custom Filter Presets

Add preset filter buttons:

```
def _apply_preset_apple(self):
    self.filter_make.setChecked(True)
    self._apply_filters()

def _apply_preset_pro(self):
    self.filter_camera.setChecked(True)
    self._apply_filters()
```

Tip 3: Progress Estimation

Show estimated time remaining:

```
import time

start_time = time.time()
completed = 0

def _update_progress(self, current, total):
    completed = current
    elapsed = time.time() - start_time

    if completed > 0:
```

```
        avg_time = elapsed / completed
        remaining = (total - completed) * avg_time
        self._log(f"Estimated time remaining: {int(remaining)}s")
```

🎬 You're Ready!

The complete application is now ready to integrate into your project. All files are:

- ✅ Fully commented and explained
- ✅ Production-ready
- ✅ Tested architecture
- ✅ Extensible design
- ✅ Error handling included

Next Steps:

1. Copy the files into your project
2. Install dependencies
3. Run `python video_sorter.py`
4. Start organizing your videos!

Questions? Check:

- README.md for detailed documentation
- Code comments for implementation details
- This guide for integration help

Happy coding! 🚀# New Section

```
    """
    Video Duplicate & Similarity Finder - GUI Edition
    ================================================
    A merging of 'video_finder_improved.py' logic with a modern PySide6 GU

    FEATURES:
    - Drag & Drop folder scanning
    - Exact duplicate detection (MD5)
    - Similar video detection (Perceptual Hash)
    - Interactive Similarity Threshold Slider
    - Multithreaded processing (GUI doesn't freeze)
    - Results grouped by similarity in a Tree View

    REQUIREMENTS:
        pip install PySide6 videohash
```

```
    FFmpeg must be installed on the system.
"""

import sys
import os
import json
import hashlib
import time
import subprocess
import unicodedata
import re
from pathlib import Path
from collections import defaultdict
from datetime import datetime

# GUI Imports
from PySide6.QtWidgets import (
    QApplication, QMainWindow, QWidget, QVBoxLayout, QHBoxLayout,
    QPushButton, QLabel, QTreeWidget, QTreeWidgetItem, QCheckBox,
    QGroupBox, QTextEdit, QFileDialog, QMessageBox, QProgressBar,
    QHeaderView, QSlider, QStyle, QFrame, QSplitter
)
from PySide6.QtCore import Qt, QThread, Signal
from PySide6.QtGui import QDragEnterEvent, QDropEvent, QColor, QBrush

# Try importing videohash for similarity features
try:
    from videohash import VideoHash
    VIDEOHASH_AVAILABLE = True
except ImportError:
    VIDEOHASH_AVAILABLE = False

# ============================================================================
# BACKEND LOGIC (Adapted from video_finder_improved.py)
# ============================================================================

CACHE_VERSION = "2.0"
CACHE_FILENAME = ".video_finder_cache.json"

class BackendUtils:
    """Helper functions ported from video_finder_improved.py"""

    @staticmethod
    def get_file_hash(filepath, chunk_size=8192):
        """Calculate MD5 hash for exact matching"""
        try:
            hash_md5 = hashlib.md5()
            with open(filepath, 'rb') as f:
```

```
                for chunk in iter(lambda: f.read(chunk_size), b""):
                    hash_md5.update(chunk)
            return hash_md5.hexdigest()
        except Exception:
            return None

    @staticmethod
    def get_perceptual_hash(filepath):
        """Calculate perceptual hash using videohash"""
        if not VIDEOHASH_AVAILABLE:
            return None
        try:
            # VideoHash creates temp files, so we wrap it carefully
            vh = VideoHash(path=filepath)
            return str(vh)
        except Exception:
            return None

    @staticmethod
    def compare_perceptual_hashes(hash1_str, hash2_str):
        """Compare two hash strings"""
        if not VIDEOHASH_AVAILABLE or not hash1_str or not hash2_str:
            return 999
        try:
            h1 = VideoHash(hash=hash1_str)
            h2 = VideoHash(hash=hash2_str)
            return h1 - h2
        except:
            return 999

    @staticmethod
    def get_video_metadata(filepath):
        """Extract basic metadata via ffprobe"""
        try:
            cmd = [
                'ffprobe', '-v', 'error',
                '-show_entries', 'format=duration,size,bit_rate',
                '-show_entries', 'stream=codec_name,width,height,r_fra
                '-of', 'json', filepath
            ]
            # Timeout added to prevent hangs on bad files
            result = subprocess.run(cmd, capture_output=True, text=Tru
            data = json.loads(result.stdout)

            format_info = data.get('format', {})
            video_stream = next((s for s in data.get('streams', []) if

            if not video_stream:
```

```
                return None

            # Calculate FPS
            fps = 0.0
            fps_str = video_stream.get('r_frame_rate', '0/1')
            if '/' in fps_str:
                num, den = map(int, fps_str.split('/'))
                if den != 0: fps = num / den
            else:
                fps = float(fps_str)

            return {
                'duration': float(format_info.get('duration', 0)),
                'size': int(format_info.get('size', 0)),
                'width': int(video_stream.get('width', 0)),
                'height': int(video_stream.get('height', 0)),
                'codec': video_stream.get('codec_name', 'unknown'),
                'framerate': round(fps, 2),
                'resolution': f"{video_stream.get('width',0)}x{video_s
            }
        except Exception:
            return None

    @staticmethod
    def format_size(size_bytes):
        if size_bytes == 0: return "0 B"
        size_name = ("B", "KB", "MB", "GB", "TB")
        i = int(0)
        p = float(size_bytes)
        while p >= 1024 and i < len(size_name)-1:
            p /= 1024
            i += 1
        return f"{p:.2f} {size_name[i]}"

    @staticmethod
    def format_duration(seconds):
        m, s = divmod(seconds, 60)
        h, m = divmod(m, 60)
        return f"{int(h):02d}:{int(m):02d}:{int(s):02d}"


class CacheManager:
    """Manages JSON cache to speed up repeated scans"""
    def __init__(self, directory):
        self.directory = directory
        self.cache_path = os.path.join(directory, CACHE_FILENAME)
        self.cache = self._load()
        self.new_entries = False
```

```
    def _load(self):
        if os.path.exists(self.cache_path):
            try:
                with open(self.cache_path, 'r') as f:
                    data = json.load(f)
                    if data.get('version') == CACHE_VERSION:
                        return data.get('files', {})
            except:
                pass
        return {}

    def get(self, filepath, mtime):
        if filepath in self.cache:
            entry = self.cache[filepath]
            if entry.get('mtime') == mtime:
                return entry.get('data')
        return None

    def set(self, filepath, mtime, data):
        self.cache[filepath] = {'mtime': mtime, 'data': data}
        self.new_entries = True

    def save(self):
        if self.new_entries:
            try:
                with open(self.cache_path, 'w') as f:
                    json.dump({
                        'version': CACHE_VERSION,
                        'timestamp': datetime.now().isoformat(),
                        'files': self.cache
                    }, f, indent=2)
            except:
                pass

# ============================================================================
# WORKER THREAD
# ============================================================================

class ScanWorker(QThread):
    """
    Background thread that runs the heavy video scanning logic.
    Communicates with GUI via Signals.
    """
    log_signal = Signal(str)
    progress_signal = Signal(int, int) # current, total
    finished_signal = Signal(dict, list, list) # video_data, duplicate
```

```
    def __init__(self, folder_path, recursive, threshold, use_cache, 
        super().__init__()
        self.folder_path = folder_path
        self.recursive = recursive
        self.threshold = threshold
        self.use_cache = use_cache
        self.quick_mode = quick_mode
        self.is_cancelled = False

    def run(self):
        self.log_signal.emit(f"🚀 Starting scan in: {self.folder_path

        # 1. Find Files
        video_extensions = {'.mp4', '.mov', '.avi', '.mkv', '.m4v', '
        video_files = []

        scan_pattern = '**/*' if self.recursive else '*'
        try:
            path_obj = Path(self.folder_path)
            # Use rglob for recursive, iterdir for non-recursive if we
            # but rglob('*') works for both if we filter.
            # Using os.walk is often safer for permissions.
            if self.recursive:
                walker = os.walk(self.folder_path)
            else:
                walker = [(self.folder_path, [], os.listdir(self.folde

            for root, dirs, files in walker:
                if self.is_cancelled: break
                for file in files:
                    if Path(file).suffix.lower() in video_extensions:
                        full_path = os.path.abspath(os.path.join(root,
                        # Resolve symlinks to avoid double counting
                        real_path = os.path.realpath(full_path)
                        if real_path not in video_files:
                            video_files.append(real_path)
        except Exception as e:
            self.log_signal.emit(f"Error scanning folder: {str(e)}")
            return

        total_files = len(video_files)
        self.log_signal.emit(f"📂 Found {total_files} video files.")

        # 2. Initialize Cache
        cache = CacheManager(self.folder_path) if self.use_cache else 
        video_data = {} # store all metadata

        # 3. Process Files
```

```
            for i, filepath in enumerate(video_files):
                if self.is_cancelled: return

                filename = os.path.basename(filepath)
                self.progress_signal.emit(i + 1, total_files)

                try:
                    mtime = os.path.getmtime(filepath)

                    # Try cache first
                    data = cache.get(filepath, mtime) if cache else None

                    if data:
                        self.log_signal.emit(f"cache hit: {filename}")
                    else:
                        self.log_signal.emit(f"Analyzing: {filename}")

                        # Heavy lifting
                        data = BackendUtils.get_video_metadata(filepath)
                        if not data: data = {}

                        # MD5 Hash
                        data['md5_hash'] = BackendUtils.get_file_hash(file

                        # Perceptual Hash (Only if not quick mode)
                        if not self.quick_mode and VIDEOHASH_AVAILABLE:
                            data['perceptual_hash'] = BackendUtils.get_per

                        if cache:
                            cache.set(filepath, mtime, data)

                    if data:
                        video_data[filepath] = data

                except Exception as e:
                    self.log_signal.emit(f"Error processing {filename}: {e

            # Save Cache
            if cache: cache.save()

            # 4. Find Exact Duplicates
            self.log_signal.emit("🔍 Analyzing Exact Duplicates (MD5)..."
            duplicates_map = defaultdict(list)
            for path, data in video_data.items():
                if data.get('md5_hash'):
                    duplicates_map[data['md5_hash']].append(path)

            # Filter out unique files
```

```
            exact_groups = [paths for paths in duplicates_map.values() if

            # 5. Find Similar Content
            similar_groups = []
            if not self.quick_mode and VIDEOHASH_AVAILABLE:
                self.log_signal.emit(f"👁️ Analyzing Visual Similarity (Th

                # Get files that have a perceptual hash
                files_with_phash = [(p, d) for p, d in video_data.items()

                processed_pairs = set()
                count = 0
                total_comparisons = len(files_with_phash) * (len(files_wi

                for idx1, (f1, d1) in enumerate(files_with_phash):
                    if self.is_cancelled: return
                    for f2, d2 in files_with_phash[idx1+1:]:
                        pair_key = tuple(sorted((f1, f2)))
                        if pair_key in processed_pairs: continue

                        dist = BackendUtils.compare_perceptual_hashes(d1[

                        if dist <= self.threshold:
                            # Simple grouping logic: add to existing grou
                            added = False
                            for group in similar_groups:
                                if f1 in group['files'] or f2 in group['f
                                    if f1 not in group['files']: group['f
                                    if f2 not in group['files']: group['f
                                    group['distance'] = min(group.get('di
                                    added = True
                                    break
                            if not added:
                                similar_groups.append({'files': [f1, f2],

                        processed_pairs.add(pair_key)
                        count += 1
                        if count % 10 == 0:
                             # Just update log occasionally, calculating
                             pass

            self.log_signal.emit("✅ Scan Complete.")
            self.finished_signal.emit(video_data, exact_groups, similar_g

        def cancel(self):
            self.is_cancelled = True
```

```
# ============================================================================
# GUI CLASSES
# ============================================================================

class VideoFinderWindow(QMainWindow):
    def __init__(self):
        super().__init__()
        self.worker = None
        self.video_data_cache = {} # Store scan results
        self.init_ui()

        # Check dependencies on launch
        if not VIDEOHASH_AVAILABLE:
            self.log("⚠️ 'videohash' library not found. Similarity se
            self.log("   Run: pip install videohash")
            self.slider_threshold.setEnabled(False)
            self.check_quick.setChecked(True)
            self.check_quick.setEnabled(False)

    def init_ui(self):
        self.setWindowTitle("Video Duplicate & Similarity Finder")
        self.resize(1100, 800)

        main_widget = QWidget()
        self.setCentralWidget(main_widget)
        main_layout = QVBoxLayout(main_widget)

        # --- Top: Configuration & Actions ---
        config_group = QGroupBox("Configuration")
        config_layout = QHBoxLayout()

        # Recursive Toggle
        self.check_recursive = QCheckBox("Recursive Scan")
        self.check_recursive.setToolTip("Scan subfolders as well")
        self.check_recursive.setChecked(True)

        # Quick Mode Toggle
        self.check_quick = QCheckBox("Quick Mode (Exact Only)")
        self.check_quick.setToolTip("Skip visual similarity check (mu
        self.check_quick.toggled.connect(self._toggle_threshold)

        # Threshold Slider
        threshold_container = QWidget()
        threshold_layout = QVBoxLayout(threshold_container)
        threshold_layout.setContentsMargins(0,0,0,0)

        self.lbl_threshold = QLabel("Similarity Threshold: 10 (Modera
        self.slider_threshold = QSlider(Qt.Horizontal)
```

```
        self.slider_threshold.setRange(1, 64)
        self.slider_threshold.setValue(10)
        self.slider_threshold.setTickPosition(QSlider.TicksBelow)
        self.slider_threshold.setTickInterval(5)
        self.slider_threshold.valueChanged.connect(self._update_thresl

        threshold_layout.addWidget(self.lbl_threshold)
        threshold_layout.addWidget(self.slider_threshold)

        # Add to config layout
        config_layout.addWidget(self.check_recursive)
        config_layout.addWidget(self.check_quick)
        config_layout.addSpacing(20)
        config_layout.addWidget(threshold_container, 1) # Stretch sli

        config_group.setLayout(config_layout)
        main_layout.addWidget(config_group)

        # --- Middle: Drop Zone & Start ---
        drop_layout = QHBoxLayout()

        self.drop_area = QPushButton("Drag & Drop Folder Here\n(or Cl
        self.drop_area.setCheckable(True)
        self.drop_area.setMinimumHeight(80)
        self.drop_area.setStyleSheet("""
            QPushButton {
                border: 2px dashed #aaa;
                border-radius: 10px;
                font-size: 16px;
                background-color: #f9f9f9;
                color: #555;
            }
            QPushButton:hover {
                background-color: #f0f0f0;
                border-color: #666;
            }
        """)
        self.drop_area.clicked.connect(self._select_folder)
        self.setAcceptDrops(True) # Enable drag/drop for window

        self.btn_start = QPushButton("Start Scan")
        self.btn_start.setMinimumHeight(80)
        self.btn_start.setMinimumWidth(150)
        self.btn_start.setStyleSheet("font-size: 16px; font-weight: b
        self.btn_start.clicked.connect(self._start_scan)
        self.btn_start.setEnabled(False) # Disabled until path select

        drop_layout.addWidget(self.drop_area, 1)
```

```
        drop_layout.addWidget(self.btn_start)
        main_layout.addLayout(drop_layout)

        self.lbl_selected_path = QLabel("No folder selected")
        self.lbl_selected_path.setStyleSheet("color: #666; font-style
        main_layout.addWidget(self.lbl_selected_path)

        self.progress_bar = QProgressBar()
        self.progress_bar.setVisible(False)
        main_layout.addWidget(self.progress_bar)

        # --- Bottom: Results (Tree View) ---
        splitter = QSplitter(Qt.Vertical)

        # Tree Widget for Results
        self.tree = QTreeWidget()
        self.tree.setHeaderLabels(["File Name", "Resolution", "Size",
        # Set column widths
        self.tree.setColumnWidth(0, 300) # Filename
        self.tree.setColumnWidth(1, 100) # Res
        self.tree.setColumnWidth(2, 80)  # Size
        self.tree.setColumnWidth(3, 80)  # Dur
        self.tree.setColumnWidth(6, 400) # Path
        self.tree.setAlternatingRowColors(True)
        splitter.addWidget(self.tree)

        # Log Window
        log_group = QGroupBox("Processing Log")
        log_layout = QVBoxLayout()
        self.log_text = QTextEdit()
        self.log_text.setReadOnly(True)
        self.log_text.setStyleSheet("background-color: #222; color: #(
        log_layout.addWidget(self.log_text)
        log_group.setLayout(log_layout)
        splitter.addWidget(log_group)

        main_layout.addWidget(splitter, 1) # Give tree/log most space

    # --- Logic ---

    def _toggle_threshold(self):
        enabled = not self.check_quick.isChecked()
        self.slider_threshold.setEnabled(enabled and VIDEOHASH_AVAILAI
        if not enabled:
            self.lbl_threshold.setText("Similarity Threshold: Disable(
        else:
            self._update_threshold_label()
```

```
        def _update_threshold_label(self):
            val = self.slider_threshold.value()
            desc = ""
            if val <= 5: desc = "Very Strict"
            elif val <= 10: desc = "Moderate"
            elif val <= 20: desc = "Loose"
            else: desc = "Very Loose"
            self.lbl_threshold.setText(f"Similarity Threshold: {val} ({des

        def _select_folder(self):
            folder = QFileDialog.getExistingDirectory(self, "Select Video
            if folder:
                self._set_folder(folder)

        def _set_folder(self, path):
            self.selected_path = path
            self.lbl_selected_path.setText(f"Selected: {path}")
            self.btn_start.setEnabled(True)
            self.log(f"Target folder set: {path}")

        def dragEnterEvent(self, event: QDragEnterEvent):
            if event.mimeData().hasUrls():
                event.acceptProposedAction()

        def dropEvent(self, event: QDropEvent):
            for url in event.mimeData().urls():
                path = url.toLocalFile()
                if os.path.isdir(path):
                    self._set_folder(path)
                    break # Just take the first folder

        def log(self, msg):
            timestamp = datetime.now().strftime("%H:%M:%S")
            self.log_text.append(f"[{timestamp}] {msg}")
            scrollbar = self.log_text.verticalScrollBar()
            scrollbar.setValue(scrollbar.maximum())

        def _start_scan(self):
            if not hasattr(self, 'selected_path') or not self.selected_pa
                return

            # UI State
            self.tree.clear()
            self.btn_start.setEnabled(False)
            self.drop_area.setEnabled(False)
            self.progress_bar.setVisible(True)
            self.progress_bar.setValue(0)
            self.log_text.clear()
```

```
            # Start Thread
            self.worker = ScanWorker(
                self.selected_path,
                self.check_recursive.isChecked(),
                self.slider_threshold.value(),
                True, # Use cache
                self.check_quick.isChecked()
            )
            self.worker.log_signal.connect(self.log)
            self.worker.progress_signal.connect(self._update_progress)
            self.worker.finished_signal.connect(self._on_scan_finished)
            self.worker.start()

        def _update_progress(self, current, total):
            self.progress_bar.setMaximum(total)
            self.progress_bar.setValue(current)

        def _on_scan_finished(self, video_data, exact_groups, similar_grou
            self.btn_start.setEnabled(True)
            self.drop_area.setEnabled(True)
            self.progress_bar.setVisible(False)
            self.video_data_cache = video_data

            # Populate Tree
            self.tree.clear()

            # 1. Exact Duplicates Root
            if exact_groups:
                root_exact = QTreeWidgetItem(self.tree)
                root_exact.setText(0, f"⚠️ Exact Duplicates ({len(exact_g
                root_exact.setForeground(0, QBrush(QColor("red")))
                root_exact.setExpanded(True)
                font = root_exact.font(0)
                font.setBold(True)
                root_exact.setFont(0, font)

                for i, group in enumerate(exact_groups, 1):
                    # Calculate group size
                    first_file_data = video_data.get(group[0], {})
                    file_size = first_file_data.get('size', 0)
                    wasted = file_size * (len(group) - 1)

                    group_node = QTreeWidgetItem(root_exact)
                    group_node.setText(0, f"Group {i} ({len(group)} copies
                    group_node.setBackground(0, QBrush(QColor("#f0f0f0")))
                    group_node.setExpanded(True)
```

```
                for filepath in group:
                    self._add_file_node(group_node, filepath, video_da

        # 2. Similar Content Root
        if similar_groups:
            root_similar = QTreeWidgetItem(self.tree)
            root_similar.setText(0, f"👀 Similar Content ({len(simila
            root_similar.setForeground(0, QBrush(QColor("blue")))
            root_similar.setExpanded(True)
            font = root_similar.font(0)
            font.setBold(True)
            root_similar.setFont(0, font)

            for i, group in enumerate(similar_groups, 1):
                dist = group.get('distance', 'N/A')
                group_node = QTreeWidgetItem(root_similar)
                group_node.setText(0, f"Similarity Group {i} (Distance
                group_node.setBackground(0, QBrush(QColor("#f0f0f0")))
                group_node.setExpanded(True)

                for filepath in group['files']:
                    self._add_file_node(group_node, filepath, video_da

        if not exact_groups and not similar_groups:
            QMessageBox.information(self, "Result", "No duplicates fou

    def _add_file_node(self, parent, filepath, data):
        filename = os.path.basename(filepath)
        node = QTreeWidgetItem(parent)

        # Columns: Name, Res, Size, Dur, FPS, Codec, Path
        node.setText(0, filename)
        node.setText(1, data.get('resolution', 'Unknown'))
        node.setText(2, BackendUtils.format_size(data.get('size', 0)))
        node.setText(3, BackendUtils.format_duration(data.get('duratio
        node.setText(4, str(data.get('framerate', '')))
        node.setText(5, data.get('codec', ''))
        node.setText(6, filepath)

        # Tooltip for full path
        node.setToolTip(0, filepath)
        node.setToolTip(6, filepath)

def main():
    app = QApplication(sys.argv)
    app.setStyle("Fusion")

    window = VideoFinderWindow()
```

```
    window.show()

    sys.exit(app.exec())

if __name__ == "__main__":
    main()
```

```
---------------------------------------------------------------------------
ModuleNotFoundError                       Traceback (most recent call last)
/tmp/ipython-input-730146239.py in <cell line: 0>()
     30
     31 # GUI Imports
---> 32 from PySide6.QtWidgets import (
     33     QApplication, QMainWindow, QWidget, QVBoxLayout, QHBoxLayout,
     34     QPushButton, QLabel, QTreeWidget, QTreeWidgetItem, QCheckBox,

ModuleNotFoundError: No module named 'PySide6'

---------------------------------------------------------------------------
NOTE: If your import is failing due to a missing package, you can
manually install dependencies using either !pip or !apt.

To view examples of installing some common dependencies, click the
"Open Examples" button below.
---------------------------------------------------------------------------
```

OPEN EXAMPLES

```
import sys
import os
import json
import hashlib
import time
import subprocess
import unicodedata
import re
from pathlib import Path
from collections import defaultdict
from datetime import datetime

# GUI Imports
```

```
# GUI imports
from PySide6.QtWidgets import (
    QApplication, QMainWindow, QWidget, QVBoxLayout
    QPushButton, QLabel, QTreeWidget, QTreeWidgetIt
    QGroupBox, QTextEdit, QFileDialog, QMessageBox,
    QHeaderView, QSlider, QStyle, QFrame, QSplitter
)
from PySide6.QtCore import Qt, QThread, Signal
from PySide6.QtGui import QDragEnterEvent, QDropEve

# Try importing videohash for similarity features
try:
    from videohash import VideoHash
    VIDEOHASH_AVAILABLE = True
except ImportError:
    VIDEOHASH_AVAILABLE = False

# ================================================
# BACKEND LOGIC (Adapted from video_finder_improved
# ================================================

CACHE_VERSION = "2.0"
CACHE_FILENAME = ".video_finder_cache.json"

class BackendUtils:
    """Helper functions ported from video_finder_im

    @staticmethod
    def get_file_hash(filepath, chunk_size=8192):
        """Calculate MD5 hash for exact matching"""
        try:
            hash_md5 = hashlib.md5()
            with open(filepath, 'rb') as f:
                for chunk in iter(lambda: f.read(ch
                    hash_md5.update(chunk)
            return hash_md5.hexdigest()
        except Exception:
            return None

    @staticmethod
    def get_perceptual_hash(filepath):
        """Calculate perceptual hash using videohas
        if not VIDEOHASH_AVAILABLE:
            return None
        try:
            # VideoHash creates temp files, so we w
            vh = VideoHash(path=filepath)
            return str(vh)
        except Exception:
```

```
            return None

    @staticmethod
    def compare_perceptual_hashes(hash1_str, hash2_
        """Compare two hash strings"""
        if not VIDEOHASH_AVAILABLE or not hash1_str
            return 999
        try:
            h1 = VideoHash(hash=hash1_str)
            h2 = VideoHash(hash=hash2_str)
            return h1 - h2
        except:
            return 999

    @staticmethod
    def get_video_metadata(filepath):
        """Extract basic metadata via ffprobe"""
        try:
            cmd = [
                'ffprobe', '-v', 'error',
                '-show_entries', 'format=duration,s
                '-show_entries', 'stream=codec_name
                '-of', 'json', filepath
            ]
            # Timeout added to prevent hangs on bad
            result = subprocess.run(cmd, capture_ou
            data = json.loads(result.stdout)

            format_info = data.get('format', {})
            video_stream = next((s for s in data.ge

            if not video_stream:
                return None

            # Calculate FPS
            fps = 0.0
            fps_str = video_stream.get('r_frame_rat
            if '/' in fps_str:
                num, den = map(int, fps_str.split('
                if den != 0: fps = num / den
            else:
                fps = float(fps_str)

            return {
                'duration': float(format_info.get('
                'size': int(format_info.get('size',
                'width': int(video_stream.get('widt
                'height': int(video_stream.get('hei
```

```
                'codec': video_stream.get('codec_na
                'framerate': round(fps, 2),
                'resolution': f"{video_stream.get('
            }
        except Exception:
            return None

    @staticmethod
    def format_size(size_bytes):
        if size_bytes == 0: return "0 B"
        size_name = ("B", "KB", "MB", "GB", "TB")
        i = int(0)
        p = float(size_bytes)
        while p >= 1024 and i < len(size_name)-1:
            p /= 1024
            i += 1
        return f"{p:.2f} {size_name[i]}"

    @staticmethod
    def format_duration(seconds):
        m, s = divmod(seconds, 60)
        h, m = divmod(m, 60)
        return f"{int(h):02d}:{int(m):02d}:{int(s):


class CacheManager:
    """Manages JSON cache to speed up repeated scan
    def __init__(self, directory):
        self.directory = directory
        self.cache_path = os.path.join(directory, C
        self.cache = self._load()
        self.new_entries = False

    def _load(self):
        if os.path.exists(self.cache_path):
            try:
                with open(self.cache_path, 'r') as
                    data = json.load(f)
                    if data.get('version') == CACHE
                        return data.get('files', {}
            except:
                pass
        return {}

    def get(self, filepath, mtime):
        if filepath in self.cache:
            entry = self.cache[filepath]
            if entry.get('mtime') == mtime:
                return entry.get('data')
```

```
                return entry.get('data')
        return None

    def set(self, filepath, mtime, data):
        self.cache[filepath] = {'mtime': mtime, 'da
        self.new_entries = True

    def save(self):
        if self.new_entries:
            try:
                with open(self.cache_path, 'w') as
                    json.dump({
                        'version': CACHE_VERSION,
                        'timestamp': datetime.now()
                        'files': self.cache
                    }, f, indent=2)
            except:
                pass

# ================================================
# WORKER THREAD
# ================================================

class ScanWorker(QThread):
    """
    Background thread that runs the heavy video sca
    Communicates with GUI via Signals.
    """
    log_signal = Signal(str)
    progress_signal = Signal(int, int) # current, t
    finished_signal = Signal(dict, list, list) # vi

    def __init__(self, folder_path, recursive, thre
        super().__init__()
        self.folder_path = folder_path
        self.recursive = recursive
        self.threshold = threshold
        self.use_cache = use_cache
        self.quick_mode = quick_mode
        self.is_cancelled = False

    def run(self):
        self.log_signal.emit(f"🚀 Starting scan in

        # 1. Find Files
        video_extensions = {'.mp4', '.mov', '.avi',
        video_files = []

        scan_pattern = '**/*' if self.recursive els
```

```
scan_pattern = "**/*" if self.recursive else "*"
try:
    path_obj = Path(self.folder_path)
    # Use rglob for recursive, iterdir for
    # but rglob('*') works for both if we f
    # Using os.walk is often safer for perm
    if self.recursive:
        walker = os.walk(self.folder_path)
    else:
        walker = [(self.folder_path, [], os

    for root, dirs, files in walker:
        if self.is_cancelled: break
        for file in files:
            if Path(file).suffix.lower() in
                full_path = os.path.abspath
                # Resolve symlinks to avoid
                real_path = os.path.realpat
                if real_path not in video_f
                    video_files.append(real
except Exception as e:
    self.log_signal.emit(f"Error scanning f
    return

total_files = len(video_files)
self.log_signal.emit(f"📂 Found {total_file

# 2. Initialize Cache
cache = CacheManager(self.folder_path) if s
video_data = {} # store all metadata

# 3. Process Files
for i, filepath in enumerate(video_files):
    if self.is_cancelled: return

    filename = os.path.basename(filepath)
    self.progress_signal.emit(i + 1, total_

    try:
        mtime = os.path.getmtime(filepath)

        # Try cache first
        data = cache.get(filepath, mtime) i

        if data:
            self.log_signal.emit(f"cache hi
        else:
            self.log_signal.emit(f"Analyzin
```

```
                # Heavy lifting
                data = BackendUtils.get_video_m
                if not data: data = {}

                # MD5 Hash
                data['md5_hash'] = BackendUtils

                # Perceptual Hash (Only if not
                if not self.quick_mode and VIDE
                    data['perceptual_hash'] = B

                if cache:
                    cache.set(filepath, mtime,

            if data:
                video_data[filepath] = data

        except Exception as e:
            self.log_signal.emit(f"Error proces

    # Save Cache
    if cache: cache.save()

    # 4. Find Exact Duplicates
    self.log_signal.emit("🔍 Analyzing Exact Du
    duplicates_map = defaultdict(list)
    for path, data in video_data.items():
        if data.get('md5_hash'):
            duplicates_map[data['md5_hash']].ap

    # Filter out unique files
    exact_groups = [paths for paths in duplicat

    # 5. Find Similar Content
    similar_groups = []
    if not self.quick_mode and VIDEOHASH_AVAILA
        self.log_signal.emit(f"👁️ Analyzing Vis

        # Get files that have a perceptual hash
        files_with_phash = [(p, d) for p, d in

        processed_pairs = set()
        count = 0
        total_comparisons = len(files_with_phas

        for idx1, (f1, d1) in enumerate(files_w
            if self.is_cancelled: return
            for f2, d2 in files_with_phash[idx1
```

```
                    pair_key = tuple(sorted((f1, f2
                    if pair_key in processed_pairs:

                    dist = BackendUtils.compare_per

                    if dist <= self.threshold:
                        # Simple grouping logic: ad
                        added = False
                        for group in similar_groups
                            if f1 in group['files']
                                if f1 not in group[
                                if f2 not in group[
                                group['distance'] =
                                added = True
                                break
                        if not added:
                            similar_groups.append({

                    processed_pairs.add(pair_key)
                    count += 1
                    if count % 10 == 0:
                         # Just update log occasior
                         pass

        self.log_signal.emit("✅ Scan Complete.")
        self.finished_signal.emit(video_data, exact

    def cancel(self):
        self.is_cancelled = True


# ================================================
# GUI CLASSES
# ================================================

class VideoFinderWindow(QMainWindow):
    def __init__(self):
        super().__init__()
        self.worker = None
        self.video_data_cache = {} # Store scan res
        self.init_ui()

        # Check dependencies on launch
        if not VIDEOHASH_AVAILABLE:
            self.log("⚠️ 'videohash' library not fo
            self.log("   Run: pip install videohash
            self.slider_threshold.setEnabled(False)
            self.check_quick.setChecked(True)
            self.check_quick.setEnabled(False)
```

```
            self.check_quick.setEnabled(False)

    def init_ui(self):
        self.setWindowTitle("Video Duplicate & Simi
        self.resize(1100, 800)

        main_widget = QWidget()
        self.setCentralWidget(main_widget)
        main_layout = QVBoxLayout(main_widget)

        # --- Top: Configuration & Actions ---
        config_group = QGroupBox("Configuration")
        config_layout = QHBoxLayout()

        # Recursive Toggle
        self.check_recursive = QCheckBox("Recursive
        self.check_recursive.setToolTip("Scan subfo
        self.check_recursive.setChecked(True)

        # Quick Mode Toggle
        self.check_quick = QCheckBox("Quick Mode (E
        self.check_quick.setToolTip("Skip visual si
        self.check_quick.toggled.connect(self._togg

        # Threshold Slider
        threshold_container = QWidget()
        threshold_layout = QVBoxLayout(threshold_co
        threshold_layout.setContentsMargins(0,0,0,0

        self.lbl_threshold = QLabel("Similarity Thr
        self.slider_threshold = QSlider(Qt.Horizont
        self.slider_threshold.setRange(1, 64)
        self.slider_threshold.setValue(10)
        self.slider_threshold.setTickPosition(QSlid
        self.slider_threshold.setTickInterval(5)
        self.slider_threshold.valueChanged.connect(

        threshold_layout.addWidget(self.lbl_thresho
        threshold_layout.addWidget(self.slider_thre

        # Add to config layout
        config_layout.addWidget(self.check_recursiv
        config_layout.addWidget(self.check_quick)
        config_layout.addSpacing(20)
        config_layout.addWidget(threshold_container

        config_group.setLayout(config_layout)
        main_layout.addWidget(config_group)
```

```
        # --- Middle: Drop Zone & Start ---
        drop_layout = QHBoxLayout()

        self.drop_area = QPushButton("Drag & Drop F
        self.drop_area.setCheckable(True)
        self.drop_area.setMinimumHeight(80)
        self.drop_area.setStyleSheet("""
            QPushButton {
                border: 2px dashed #aaa;
                border-radius: 10px;
                font-size: 16px;
                background-color: #f9f9f9;
                color: #555;
            }
            QPushButton:hover {
                background-color: #f0f0f0;
                border-color: #666;
            }
        """)
        self.drop_area.clicked.connect(self._select
        self.setAcceptDrops(True) # Enable drag/dro

        self.btn_start = QPushButton("Start Scan")
        self.btn_start.setMinimumHeight(80)
        self.btn_start.setMinimumWidth(150)
        self.btn_start.setStyleSheet("font-size: 16
        self.btn_start.clicked.connect(self._start_
        self.btn_start.setEnabled(False) # Disabled

        drop_layout.addWidget(self.drop_area, 1)
        drop_layout.addWidget(self.btn_start)
        main_layout.addLayout(drop_layout)

        self.lbl_selected_path = QLabel("No folder
        self.lbl_selected_path.setStyleSheet("color
        main_layout.addWidget(self.lbl_selected_pat

        self.progress_bar = QProgressBar()
        self.progress_bar.setVisible(False)
        main_layout.addWidget(self.progress_bar)

        # --- Bottom: Results (Tree View) ---
        splitter = QSplitter(Qt.Vertical)

        # Tree Widget for Results
        self.tree = QTreeWidget()
        self.tree.setHeaderLabels(["File Name", "Re
        # Set column widths
```

```
        self.tree.setColumnWidth(0, 300) # Filename
        self.tree.setColumnWidth(1, 100) # Res
        self.tree.setColumnWidth(2, 80)  # Size
        self.tree.setColumnWidth(3, 80)  # Dur
        self.tree.setColumnWidth(6, 400) # Path
        self.tree.setAlternatingRowColors(True)
        splitter.addWidget(self.tree)

        # Log Window
        log_group = QGroupBox("Processing Log")
        log_layout = QVBoxLayout()
        self.log_text = QTextEdit()
        self.log_text.setReadOnly(True)
        self.log_text.setStyleSheet("background-col
        log_layout.addWidget(self.log_text)
        log_group.setLayout(log_layout)
        splitter.addWidget(log_group)

        main_layout.addWidget(splitter, 1) # Give t

    # --- Logic ---

    def _toggle_threshold(self):
        enabled = not self.check_quick.isChecked()
        self.slider_threshold.setEnabled(enabled ar
        if not enabled:
            self.lbl_threshold.setText("Similarity
        else:
            self._update_threshold_label()

    def _update_threshold_label(self):
        val = self.slider_threshold.value()
        desc = ""
        if val <= 5: desc = "Very Strict"
        elif val <= 10: desc = "Moderate"
        elif val <= 20: desc = "Loose"
        else: desc = "Very Loose"
        self.lbl_threshold.setText(f"Similarity Thr

    def _select_folder(self):
        folder = QFileDialog.getExistingDirectory(s
        if folder:
            self._set_folder(folder)

    def _set_folder(self, path):
        self.selected_path = path
        self.lbl_selected_path.setText(f"Selected:
        self.btn_start.setEnabled(True)
```

```
        self.log(f"Target folder set: {path}")

    def dragEnterEvent(self, event: QDragEnterEvent
        if event.mimeData().hasUrls():
            event.acceptProposedAction()

    def dropEvent(self, event: QDropEvent):
        for url in event.mimeData().urls():
            path = url.toLocalFile()
            if os.path.isdir(path):
                self._set_folder(path)
                break # Just take the first folder

    def log(self, msg):
        timestamp = datetime.now().strftime("%H:%M:
        self.log_text.append(f"[{timestamp}] {msg}"
        scrollbar = self.log_text.verticalScrollBar
        scrollbar.setValue(scrollbar.maximum())

    def _start_scan(self):
        if not hasattr(self, 'selected_path') or no
            return

        # UI State
        self.tree.clear()
        self.btn_start.setEnabled(False)
        self.drop_area.setEnabled(False)
        self.progress_bar.setVisible(True)
        self.progress_bar.setValue(0)
        self.log_text.clear()

        # Start Thread
        self.worker = ScanWorker(
            self.selected_path,
            self.check_recursive.isChecked(),
            self.slider_threshold.value(),
            True, # Use cache
            self.check_quick.isChecked()
        )
        self.worker.log_signal.connect(self.log)
        self.worker.progress_signal.connect(self._u
        self.worker.finished_signal.connect(self._o
        self.worker.start()

    def _update_progress(self, current, total):
        self.progress_bar.setMaximum(total)
        self.progress_bar.setValue(current)

    def _on_scan_finished(self, video_data, exact_d
```

```
def _on_scan_finished(self, video_data, exact_g
    self.btn_start.setEnabled(True)
    self.drop_area.setEnabled(True)
    self.progress_bar.setVisible(False)
    self.video_data_cache = video_data

    # Populate Tree
    self.tree.clear()

    # 1. Exact Duplicates Root
    if exact_groups:
        root_exact = QTreeWidgetItem(self.tree)
        root_exact.setText(0, f"⚠️ Exact Dupli
        root_exact.setForeground(0, QBrush(QCol
        root_exact.setExpanded(True)
        font = root_exact.font(0)
        font.setBold(True)
        root_exact.setFont(0, font)

        for i, group in enumerate(exact_groups,
            # Calculate group size
            first_file_data = video_data.get(gr
            file_size = first_file_data.get('si
            wasted = file_size * (len(group) -

            group_node = QTreeWidgetItem(root_e
            group_node.setText(0, f"Group {i} (
            group_node.setBackground(0, QBrush(
            group_node.setExpanded(True)

            for filepath in group:
                self._add_file_node(group_node,

    # 2. Similar Content Root
    if similar_groups:
        root_similar = QTreeWidgetItem(self.tre
        root_similar.setText(0, f"👀 Similar C
        root_similar.setForeground(0, QBrush(QC
        root_similar.setExpanded(True)
        font = root_similar.font(0)
        font.setBold(True)
        root_similar.setFont(0, font)

        for i, group in enumerate(similar_group
            dist = group.get('distance', 'N/A')
            group_node = QTreeWidgetItem(root_s
            group_node.setText(0, f"Similarity
            group_node.setBackground(0, QBrush(
            group_node.setExpanded(True)
```

```
                group_node.setExpanded(True)

                for filepath in group['files']:
                    self._add_file_node(group_node,

        if not exact_groups and not similar_groups:
            QMessageBox.information(self, "Result",

    def _add_file_node(self, parent, filepath, data
        filename = os.path.basename(filepath)
        node = QTreeWidgetItem(parent)

        # Columns: Name, Res, Size, Dur, FPS, Codec
        node.setText(0, filename)
        node.setText(1, data.get('resolution', 'Unk
        node.setText(2, BackendUtils.format_size(da
        node.setText(3, BackendUtils.format_duratio
        node.setText(4, str(data.get('framerate', '
        node.setText(5, data.get('codec', ''))
        node.setText(6, filepath)

        # Tooltip for full path
        node.setToolTip(0, filepath)
        node.setToolTip(6, filepath)

def main():
    app = QApplication(sys.argv)
    app.setStyle("Fusion")

    window = VideoFinderWindow()
    window.show()

    sys.exit(app.exec())

if __name__ == "__main__":
    main()
```

```
%%writefile video_finder_gui.py

import sys
import os
import json
import hashlib
import time
import subprocess
import unicodedata
import re
from pathlib import Path
from collections import defaultdict
```

```
from collections import defaultdict
from datetime import datetime

# GUI Imports
from PySide6.QtWidgets import (
    QApplication, QMainWindow, QWidget, QVBoxLayout
    QPushButton, QLabel, QTreeWidget, QTreeWidgetIt
    QGroupBox, QTextEdit, QFileDialog, QMessageBox,
    QHeaderView, QSlider, QStyle, QFrame, QSplitter
)
from PySide6.QtCore import Qt, QThread, Signal
from PySide6.QtGui import QDragEnterEvent, QDropEve

# Try importing videohash for similarity features
try:
    from videohash import VideoHash
    VIDEOHASH_AVAILABLE = True
except ImportError:
    VIDEOHASH_AVAILABLE = False

# ==================================================
# BACKEND LOGIC (Adapted from video_finder_improved
# ==================================================

CACHE_VERSION = "2.0"
CACHE_FILENAME = ".video_finder_cache.json"

class BackendUtils:
    """Helper functions ported from video_finder_im

    @staticmethod
    def get_file_hash(filepath, chunk_size=8192):
        """Calculate MD5 hash for exact matching"""
        try:
            hash_md5 = hashlib.md5()
            with open(filepath, 'rb') as f:
                for chunk in iter(lambda: f.read(ch
                    hash_md5.update(chunk)
            return hash_md5.hexdigest()
        except Exception:
            return None

    @staticmethod
    def get_perceptual_hash(filepath):
        """Calculate perceptual hash using videohas
        if not VIDEOHASH_AVAILABLE:
            return None
        try:
            # VideoHash creates temp files, so we w
```

```
            # VideoHash creates temp files, so we
            vh = VideoHash(path=filepath)
            return str(vh)
        except Exception:
            return None

    @staticmethod
    def compare_perceptual_hashes(hash1_str, hash2_
        """Compare two hash strings"""
        if not VIDEOHASH_AVAILABLE or not hash1_str
            return 999
        try:
            h1 = VideoHash(hash=hash1_str)
            h2 = VideoHash(hash=hash2_str)
            return h1 - h2
        except:
            return 999

    @staticmethod
    def get_video_metadata(filepath):
        """Extract basic metadata via ffprobe"""
        try:
            cmd = [
                'ffprobe', '-v', 'error',
                '-show_entries', 'format=duration,s
                '-show_entries', 'stream=codec_name
                '-of', 'json', filepath
            ]
            # Timeout added to prevent hangs on bad
            result = subprocess.run(cmd, capture_ou
            data = json.loads(result.stdout)

            format_info = data.get('format', {})
            video_stream = next((s for s in data.ge

            if not video_stream:
                return None

            # Calculate FPS
            fps = 0.0
            fps_str = video_stream.get('r_frame_rat
            if '/' in fps_str:
                num, den = map(int, fps_str.split('
                if den != 0: fps = num / den
            else:
                fps = float(fps_str)

            return {
                'duration': float(format_info.get('
```

```
                'size': int(format_info.get('size',
                'width': int(video_stream.get('widt
                'height': int(video_stream.get('hei
                'codec': video_stream.get('codec_na
                'framerate': round(fps, 2),
                'resolution': f"{video_stream.get('
            }
        except Exception:
            return None

    @staticmethod
    def format_size(size_bytes):
        if size_bytes == 0: return "0 B"
        size_name = ("B", "KB", "MB", "GB", "TB")
        i = int(0)
        p = float(size_bytes)
        while p >= 1024 and i < len(size_name)-1:
            p /= 1024
            i += 1
        return f"{p:.2f} {size_name[i]}"

    @staticmethod
    def format_duration(seconds):
        m, s = divmod(seconds, 60)
        h, m = divmod(m, 60)
        return f"{int(h):02d}:{int(m):02d}:{int(s):


class CacheManager:
    """Manages JSON cache to speed up repeated scar
    def __init__(self, directory):
        self.directory = directory
        self.cache_path = os.path.join(directory, C
        self.cache = self._load()
        self.new_entries = False

    def _load(self):
        if os.path.exists(self.cache_path):
            try:
                with open(self.cache_path, 'r') as
                    data = json.load(f)
                    if data.get('version') == CACHE
                        return data.get('files', {}
            except:
                pass
        return {}

    def get(self, filepath, mtime):
```

```
            if filepath in self.cache:
                entry = self.cache[filepath]
                if entry.get('mtime') == mtime:
                    return entry.get('data')
            return None

        def set(self, filepath, mtime, data):
            self.cache[filepath] = {'mtime': mtime, 'da
            self.new_entries = True

        def save(self):
            if self.new_entries:
                try:
                    with open(self.cache_path, 'w') as
                        json.dump({
                            'version': CACHE_VERSION,
                            'timestamp': datetime.now()
                            'files': self.cache
                        }, f, indent=2)
                except:
                    pass

    # ==================================================
    # WORKER THREAD
    # ==================================================

    class ScanWorker(QThread):
        """
        Background thread that runs the heavy video sca
        Communicates with GUI via Signals.
        """
        log_signal = Signal(str)
        progress_signal = Signal(int, int) # current, t
        finished_signal = Signal(dict, list, list) # vi

        def __init__(self, folder_path, recursive, thre
            super().__init__()
            self.folder_path = folder_path
            self.recursive = recursive
            self.threshold = threshold
            self.use_cache = use_cache
            self.quick_mode = quick_mode
            self.is_cancelled = False

        def run(self):
            self.log_signal.emit(f"🚀 Starting scan in

            # 1. Find Files
            video_extensions = {'.mp4', '.mov', '.avi
```

```
video_extensions = {'.mp4', '.mov', '.avi',
video_files = []

scan_pattern = '**/*' if self.recursive els
try:
    path_obj = Path(self.folder_path)
    # Use rglob for recursive, iterdir for
    # but rglob('*') works for both if we t
    # Using os.walk is often safer for perm
    if self.recursive:
        walker = os.walk(self.folder_path)
    else:
        walker = [(self.folder_path, [], os

    for root, dirs, files in walker:
        if self.is_cancelled: break
        for file in files:
            if Path(file).suffix.lower() in
                full_path = os.path.abspath
                # Resolve symlinks to avoid
                real_path = os.path.realpat
                if real_path not in video_f
                    video_files.append(real
except Exception as e:
    self.log_signal.emit(f"Error scanning f
    return

total_files = len(video_files)
self.log_signal.emit(f"📂 Found {total_file

# 2. Initialize Cache
cache = CacheManager(self.folder_path) if s
video_data = {} # store all metadata

# 3. Process Files
for i, filepath in enumerate(video_files):
    if self.is_cancelled: return

    filename = os.path.basename(filepath)
    self.progress_signal.emit(i + 1, total_

    try:
        mtime = os.path.getmtime(filepath)

        # Try cache first
        data = cache.get(filepath, mtime) i

        if data:
            self.log_signal.emit(f"cache hi
```

```
                self.log_signal.emit(f" cache h
            else:
                self.log_signal.emit(f"Analyzir

                # Heavy lifting
                data = BackendUtils.get_video_m
                if not data: data = {}

                # MD5 Hash
                data['md5_hash'] = BackendUtils

                # Perceptual Hash (Only if not
                if not self.quick_mode and VIDE
                    data['perceptual_hash'] = B

                if cache:
                    cache.set(filepath, mtime,

            if data:
                video_data[filepath] = data

        except Exception as e:
            self.log_signal.emit(f"Error proces

    # Save Cache
    if cache: cache.save()

    # 4. Find Exact Duplicates
    self.log_signal.emit("🔍 Analyzing Exact Du
    duplicates_map = defaultdict(list)
    for path, data in video_data.items():
        if data.get('md5_hash'):
            duplicates_map[data['md5_hash']].ap

    # Filter out unique files
    exact_groups = [paths for paths in duplicat

    # 5. Find Similar Content
    similar_groups = []
    if not self.quick_mode and VIDEOHASH_AVAILA
        self.log_signal.emit(f"👁️ Analyzing Vis

        # Get files that have a perceptual hash
        files_with_phash = [(p, d) for p, d in

        processed_pairs = set()
        count = 0
        total_comparisons = len(files_with_phas
```

```
            for idx1, (f1, d1) in enumerate(files_w
                if self.is_cancelled: return
                for f2, d2 in files_with_phash[idx1
                    pair_key = tuple(sorted((f1, f2
                    if pair_key in processed_pairs:

                    dist = BackendUtils.compare_per

                    if dist <= self.threshold:
                        # Simple grouping logic: ad
                        added = False
                        for group in similar_groups
                            if f1 in group['files']
                                if f1 not in group[
                                if f2 not in group[
                                group['distance'] =
                                added = True
                                break
                        if not added:
                            similar_groups.append({

                    processed_pairs.add(pair_key)
                    count += 1
                    if count % 10 == 0:
                         # Just update log occasion
                         pass

        self.log_signal.emit("✅ Scan Complete.")
        self.finished_signal.emit(video_data, exact

    def cancel(self):
        self.is_cancelled = True


# ==================================================
# GUI CLASSES
# ==================================================

class VideoFinderWindow(QMainWindow):
    def __init__(self):
        super().__init__()
        self.worker = None
        self.video_data_cache = {} # Store scan res
        self.init_ui()

        # Check dependencies on launch
        if not VIDEOHASH_AVAILABLE:
            self.log("⚠️ 'videohash' library not f
```

```
            self.log("   Run: pip install videohash
            self.slider_threshold.setEnabled(False)
            self.check_quick.setChecked(True)
            self.check_quick.setEnabled(False)

    def init_ui(self):
        self.setWindowTitle("Video Duplicate & Simi
        self.resize(1100, 800)

        main_widget = QWidget()
        self.setCentralWidget(main_widget)
        main_layout = QVBoxLayout(main_widget)

        # --- Top: Configuration & Actions ---
        config_group = QGroupBox("Configuration")
        config_layout = QHBoxLayout()

        # Recursive Toggle
        self.check_recursive = QCheckBox("Recursive
        self.check_recursive.setToolTip("Scan subfo
        self.check_recursive.setChecked(True)

        # Quick Mode Toggle
        self.check_quick = QCheckBox("Quick Mode (E
        self.check_quick.setToolTip("Skip visual si
        self.check_quick.toggled.connect(self._togg

        # Threshold Slider
        threshold_container = QWidget()
        threshold_layout = QVBoxLayout(threshold_co
        threshold_layout.setContentsMargins(0,0,0,0

        self.lbl_threshold = QLabel("Similarity Thr
        self.slider_threshold = QSlider(Qt.Horizont
        self.slider_threshold.setRange(1, 64)
        self.slider_threshold.setValue(10)
        self.slider_threshold.setTickPosition(QSlid
        self.slider_threshold.setTickInterval(5)
        self.slider_threshold.valueChanged.connect(

        threshold_layout.addWidget(self.lbl_thresho
        threshold_layout.addWidget(self.slider_thre

        # Add to config layout
        config_layout.addWidget(self.check_recursiv
        config_layout.addWidget(self.check_quick)
        config_layout.addSpacing(20)
        config_layout.addWidget(threshold_container
```

```
config_group.setLayout(config_layout)
main_layout.addWidget(config_group)

# --- Middle: Drop Zone & Start ---
drop_layout = QHBoxLayout()

self.drop_area = QPushButton("Drag & Drop F
self.drop_area.setCheckable(True)
self.drop_area.setMinimumHeight(80)
self.drop_area.setStyleSheet("""
    QPushButton {
        border: 2px dashed #aaa;
        border-radius: 10px;
        font-size: 16px;
        background-color: #f9f9f9;
        color: #555;
    }
    QPushButton:hover {
        background-color: #f0f0f0;
        border-color: #666;
    }
""")
self.drop_area.clicked.connect(self._select
self.setAcceptDrops(True) # Enable drag/dro

self.btn_start = QPushButton("Start Scan")
self.btn_start.setMinimumHeight(80)
self.btn_start.setMinimumWidth(150)
self.btn_start.setStyleSheet("font-size: 16
self.btn_start.clicked.connect(self._start_
self.btn_start.setEnabled(False) # Disabled

drop_layout.addWidget(self.drop_area, 1)
drop_layout.addWidget(self.btn_start)
main_layout.addLayout(drop_layout)

self.lbl_selected_path = QLabel("No folder
self.lbl_selected_path.setStyleSheet("color
main_layout.addWidget(self.lbl_selected_pat

self.progress_bar = QProgressBar()
self.progress_bar.setVisible(False)
main_layout.addWidget(self.progress_bar)

# --- Bottom: Results (Tree View) ---
splitter = QSplitter(Qt.Vertical)

# Tree Widget for Results
```

```
        # Tree Widget for Results
        self.tree = QTreeWidget()
        self.tree.setHeaderLabels(["File Name", "Re
        # Set column widths
        self.tree.setColumnWidth(0, 300) # Filename
        self.tree.setColumnWidth(1, 100) # Res
        self.tree.setColumnWidth(2, 80)  # Size
        self.tree.setColumnWidth(3, 80)  # Dur
        self.tree.setColumnWidth(6, 400) # Path
        self.tree.setAlternatingRowColors(True)
        splitter.addWidget(self.tree)

        # Log Window
        log_group = QGroupBox("Processing Log")
        log_layout = QVBoxLayout()
        self.log_text = QTextEdit()
        self.log_text.setReadOnly(True)
        self.log_text.setStyleSheet("background-col
        log_layout.addWidget(self.log_text)
        log_group.setLayout(log_layout)
        splitter.addWidget(log_group)

        main_layout.addWidget(splitter, 1) # Give t

    # --- Logic ---

    def _toggle_threshold(self):
        enabled = not self.check_quick.isChecked()
        self.slider_threshold.setEnabled(enabled ar
        if not enabled:
            self.lbl_threshold.setText("Similarity
        else:
            self._update_threshold_label()

    def _update_threshold_label(self):
        val = self.slider_threshold.value()
        desc = ""
        if val <= 5: desc = "Very Strict"
        elif val <= 10: desc = "Moderate"
        elif val <= 20: desc = "Loose"
        else: desc = "Very Loose"
        self.lbl_threshold.setText(f"Similarity Thr

    def _select_folder(self):
        folder = QFileDialog.getExistingDirectory(s
        if folder:
            self._set_folder(folder)

    def _set_folder(self, path):
```

```
        self.selected_path = path
        self.lbl_selected_path.setText(f"Selected:
        self.btn_start.setEnabled(True)
        self.log(f"Target folder set: {path}")

    def dragEnterEvent(self, event: QDragEnterEvent
        if event.mimeData().hasUrls():
            event.acceptProposedAction()

    def dropEvent(self, event: QDropEvent):
        for url in event.mimeData().urls():
            path = url.toLocalFile()
            if os.path.isdir(path):
                self._set_folder(path)
                break # Just take the first folder

    def log(self, msg):
        timestamp = datetime.now().strftime("%H:%M:
        self.log_text.append(f"[{timestamp}] {msg}"
        scrollbar = self.log_text.verticalScrollBar
        scrollbar.setValue(scrollbar.maximum())

    def _start_scan(self):
        if not hasattr(self, 'selected_path') or no
            return

        # UI State
        self.tree.clear()
        self.btn_start.setEnabled(False)
        self.drop_area.setEnabled(False)
        self.progress_bar.setVisible(True)
        self.progress_bar.setValue(0)
        self.log_text.clear()

        # Start Thread
        self.worker = ScanWorker(
            self.selected_path,
            self.check_recursive.isChecked(),
            self.slider_threshold.value(),
            True, # Use cache
            self.check_quick.isChecked()
        )
        self.worker.log_signal.connect(self.log)
        self.worker.progress_signal.connect(self._u
        self.worker.finished_signal.connect(self._o
        self.worker.start()

    def _update_progress(self, current, total):
```

```
        self.progress_bar.setMaximum(total)
        self.progress_bar.setValue(current)

    def _on_scan_finished(self, video_data, exact_g
        self.btn_start.setEnabled(True)
        self.drop_area.setEnabled(True)
        self.progress_bar.setVisible(False)
        self.video_data_cache = video_data

        # Populate Tree
        self.tree.clear()

        # 1. Exact Duplicates Root
        if exact_groups:
            root_exact = QTreeWidgetItem(self.tree)
            root_exact.setText(0, f"⚠️ Exact Dupli
            root_exact.setForeground(0, QBrush(QCol
            root_exact.setExpanded(True)
            font = root_exact.font(0)
            font.setBold(True)
            root_exact.setFont(0, font)

            for i, group in enumerate(exact_groups,
                # Calculate group size
                first_file_data = video_data.get(gr
                file_size = first_file_data.get('si
                wasted = file_size * (len(group) -

                group_node = QTreeWidgetItem(root_e
                group_node.setText(0, f"Group {i} (
                group_node.setBackground(0, QBrush(
                group_node.setExpanded(True)

                for filepath in group:
                    self._add_file_node(group_node,

        # 2. Similar Content Root
        if similar_groups:
            root_similar = QTreeWidgetItem(self.tre
            root_similar.setText(0, f"👀 Similar C
            root_similar.setForeground(0, QBrush(QC
            root_similar.setExpanded(True)
            font = root_similar.font(0)
            font.setBold(True)
            root_similar.setFont(0, font)

            for i, group in enumerate(similar_group
                dist = group.get('distance', 'N/A')
                group_node = QTreeWidgetItem(root_s
```

```
                group_node = QTreeWidgetItem(root_s
                group_node.setText(0, f"Similarity
                group_node.setBackground(0, QBrush(
                group_node.setExpanded(True)

                for filepath in group['files']:
                    self._add_file_node(group_node,

        if not exact_groups and not similar_groups:
            QMessageBox.information(self, "Result",

    def _add_file_node(self, parent, filepath, data
        filename = os.path.basename(filepath)
        node = QTreeWidgetItem(parent)

        # Columns: Name, Res, Size, Dur, FPS, Codec
        node.setText(0, filename)
        node.setText(1, data.get('resolution', 'Unk
        node.setText(2, BackendUtils.format_size(da
        node.setText(3, BackendUtils.format_duratio
        node.setText(4, str(data.get('framerate', '
        node.setText(5, data.get('codec', ''))
        node.setText(6, filepath)

        # Tooltip for full path
        node.setToolTip(0, filepath)
        node.setToolTip(6, filepath)

def main():
    app = QApplication(sys.argv)
    app.setStyle("Fusion")

    window = VideoFinderWindow()
    window.show()

    sys.exit(app.exec())

if __name__ == "__main__":
    main()
```

```
Writing video_finder_gui.py
```

```
!pip install videohash
```

```
Collecting videohash
  Downloading videohash-3.0.1-py3-none-any.whl.metadata (12 kB)
Requirement already satisfied: Pillow in /usr/local/lib/python3.12/dist
Collecting ImageHash (from videohash)
  Downloading ImageHash-4.3.2-py2.py3-none-any.whl.metadata (8.4 kB)
Collecting imagedominantcolor (from videohash)
  Downloading imagedominantcolor-1.0.1-py3-none-any.whl.metadata (6.0 k
Collecting yt-dlp (from videohash)
  Downloading yt_dlp-2025.12.8-py3-none-any.whl.metadata (180 kB)
     ━━━━━━━━━━━━━━━━━━━━━━━━━━━━━━━━━━━━━━━━ 180.3/180.3 kB 7.7 MB/s et
Requirement already satisfied: PyWavelets in /usr/local/lib/python3.12/
Requirement already satisfied: numpy in /usr/local/lib/python3.12/dist-
Requirement already satisfied: scipy in /usr/local/lib/python3.12/dist-
Downloading videohash-3.0.1-py3-none-any.whl (23 kB)
Downloading imagedominantcolor-1.0.1-py3-none-any.whl (5.9 kB)
Downloading ImageHash-4.3.2-py2.py3-none-any.whl (296 kB)
     ━━━━━━━━━━━━━━━━━━━━━━━━━━━━━━━━━━━━━━━━ 296.7/296.7 kB 20.0 MB/s et
Downloading yt_dlp-2025.12.8-py3-none-any.whl (3.3 MB)
     ━━━━━━━━━━━━━━━━━━━━━━━━━━━━━━━━━━━━━━━━ 3.3/3.3 MB 42.3 MB/s eta 0:
Installing collected packages: yt-dlp, imagedominantcolor, ImageHash, v
Successfully installed ImageHash-4.3.2 imagedominantcolor-1.0.1 videoha
```

```
!pip install PySide6
```

```
Collecting PySide6
  Downloading pyside6-6.10.1-cp39-abi3-manylinux_2_34_x86_64.whl.metada
Collecting shiboken6==6.10.1 (from PySide6)
  Downloading shiboken6-6.10.1-cp39-abi3-manylinux_2_34_x86_64.whl.meta
Collecting PySide6_Essentials==6.10.1 (from PySide6)
  Downloading pyside6_essentials-6.10.1-cp39-abi3-manylinux_2_34_x86_64
Collecting PySide6_Addons==6.10.1 (from PySide6)
  Downloading pyside6_addons-6.10.1-cp39-abi3-manylinux_2_34_x86_64.whl
Downloading pyside6-6.10.1-cp39-abi3-manylinux_2_34_x86_64.whl (557 kB
     ━━━━━━━━━━━━━━━━━━━━━━━━━━━━━━━━━━━━━━━━ 557.8/557.8 kB 15.0 MB/s et
Downloading pyside6_addons-6.10.1-cp39-abi3-manylinux_2_34_x86_64.whl
     ━━━━━━━━━━━━━━━━━━━━━━━━━━━━━━━━━━━━━━━━ 170.7/170.7 MB 7.2 MB/s eta
Downloading pyside6_essentials-6.10.1-cp39-abi3-manylinux_2_34_x86_64.w
     ━━━━━━━━━━━━━━━━━━━━━━━━━━━━━━━━━━━━━━━━ 76.8/76.8 MB 9.9 MB/s eta 0
Downloading shiboken6-6.10.1-cp39-abi3-manylinux_2_34_x86_64.whl (271 k
     ━━━━━━━━━━━━━━━━━━━━━━━━━━━━━━━━━━━━━━━━ 272.0/272.0 kB 24.1 MB/s et
Installing collected packages: shiboken6, PySide6_Essentials, PySide6_A
Successfully installed PySide6-6.10.1 PySide6_Addons-6.10.1 PySide6_Ess
```

Video Sorter - Complete Desktop Application

A professional desktop application for analyzing, organizing, and filtering video files based on their metadata.

🎯 What Does This Do?

This application:

- **Analyzes videos** - Extracts resolution, orientation, framerate, device info, camera metadata, GPS data
- **Smart filtering** - Filter by manufacturer (Apple, Samsung), model (iPhone 14 Pro), camera metadata, or GPS data
- **Exports results** - Save filtered results as CSV or TXT for further processing
- **Real-time processing** - Shows live progress as videos are analyzed
- **Drag & drop** - Simply drag a folder onto the app to start

Example Use Cases

1. **Organize vacation footage** - Filter by GPS to find location-tagged videos
2. **Sort phone recordings** - Filter by Make=Apple, Model=iPhone to find all iPhone videos
3. **Professional workflows** - Filter by camera metadata to separate pro camera footage from phone videos
4. **Bulk analysis** - Process hundreds of videos and export organized lists

📋 Requirements

System Requirements

- **Operating System**: Windows, macOS, or Linux
- **Python**: 3.8 or higher
- **Disk Space**: Minimal (< 50MB)

Software Dependencies

1. **FFmpeg** (for video analysis)
2. **Python packages**: PySide6

🚀 Installation

Step 1: Install FFmpeg

FFmpeg is the industry-standard tool for video analysis. Install it for your platform:

macOS

```
brew install ffmpeg
```

Linux (Ubuntu/Debian)

```
sudo apt-get update
sudo apt-get install ffmpeg
```

Windows

Option 1 - Using Chocolatey (recommended):

```
choco install ffmpeg
```

Option 2 - Manual installation:

1. Download from https://ffmpeg.org/download.html
2. Extract to `C:\ffmpeg`
3. Add `C:\ffmpeg\bin` to your PATH environment variable

Verify FFmpeg Installation

```
ffprobe -version
```

You should see version information. If you get "command not found", FFmpeg isn't installed correctly.

Step 2: Install Python Dependencies

```
pip install PySide6
```

Why PySide6?

- It's the official Python binding for Qt (used by professionals)
- Cross-platform native look and feel
- Excellent performance for desktop apps
- Free and open source (LGPL license)

Step 3: Download the Application Files

You need two Python files in the same folder:

1. **video_classifier.py** - The video analysis engine
2. **video_sorter.py** - The GUI application

Create a project folder and place both files there:

```
my_video_sorter/
├── video_classifier.py
└── video_sorter.py
```

🎮 Usage

Running the Application

Navigate to your project folder and run:

```
python video_sorter.py
```

Using the GUI

1. **Load Videos**
    - **Option A**: Drag & drop a folder onto the window
    - **Option B**: Click "Select Folder" and choose a folder
2. **View Results**
    - Videos appear in the table as they're processed
    - See resolution, orientation, FPS, device info, camera/GPS data
3. **Apply Filters** (Optional)
    - Check "Filter by Make" - Shows only videos with manufacturer info (Apple, Samsung, etc.)
    - Check "Filter by Model" - Shows only videos with device model info
    - Check "Has Camera Metadata" - Shows only videos with camera information
    - Check "Has GPS Data" - Shows only videos with GPS coordinates
    - **Multiple filters = AND condition** (all must match)

4. **Export Results**
    - Click "Export CSV" for spreadsheet-compatible format
    - Click "Export TXT" for human-readable format
    - Only filtered/visible results are exported
5. **View Code Structure**
    - Click "Show Python Code" to see documentation

Understanding the Filters

How Filters Work:

- Filters create an AND condition
- If NO filters are checked → Shows everything
- If filters are checked → Shows only videos matching ALL active filters

Examples:

Filters Active	Result
None	Shows all videos
Make only	Shows videos with manufacturer info
Camera only	Shows videos with camera metadata
Make + Camera	Shows videos with BOTH manufacturer AND camera data
Make + Model + GPS	Shows videos with manufacturer AND model AND GPS data

📊 Understanding the Results

Table Columns

Column	Values	Description
Filename	original.mp4	Original video filename
Resolution	4K, 1080p, 720p, HD, SD	Video resolution category
Orientation	V, W	V=Vertical (portrait), W=Wide (landscape)
FPS	30, 60	Framerate category (30fps or 60fps)
Make	Apple, Samsung, GoPro, etc.	Device manufacturer
Model	iPhone 14 Pro, Galaxy S23, etc.	Specific device model
Camera	Yes, No	Has camera metadata?
GPS	Yes, No	Has GPS coordinates?

Resolution Categories

- **4K**: 3840×2160 or higher (2160p in either dimension)

- **1080p**: Exactly 1920×1080 (Full HD)
- **720p**: Exactly 1280×720 (HD Ready)
- **HD**: Between 1080p and 4K
- **SD**: Below 720p (Standard Definition)

Orientation Detection

The app correctly handles rotated videos (common with phone footage):

- **Vertical (V)**: Height > Width (e.g., 1080×1920 phone video)
- **Wide (W)**: Width ≥ Height (e.g., 1920×1080 normal video)

Framerate Categories

- **30fps category**: 24, 25, 29.97, 30 fps (standard video)
- **60fps category**: 50, 59.94, 60, 120, 240 fps (high framerate/slow motion)

🔧 Advanced Usage

Using video_classifier.py as a Library

You can import the classifier in your own Python scripts:

```
from video_classifier import classify_video

# Analyze a single video
result = classify_video("path/to/video.mp4")

if result['success']:
    print(f"Resolution: {result['resolution']}")
    print(f"Make: {result['make']}")
    print(f"Has Camera: {result['has_camera']}")
    print(f"Has GPS: {result['has_gps']}")
else:
    print(f"Error: {result['error']}")
```

Batch Processing Script

Process multiple videos programmatically:

```
from video_classifier import classify_video
import os
```

```
video_folder = "path/to/videos"
results = []

for filename in os.listdir(video_folder):
    if filename.endswith(('.mp4', '.mov', '.avi')):
        filepath = os.path.join(video_folder, filename)
        result = classify_video(filepath)

        if result['success']:
            results.append(result)
            print(f"✓ {filename}: {result['resolution']} {result['orienta
        else:
            print(f"✗ {filename}: {result['error']}")

# Filter results
apple_videos = [r for r in results if r['make'] == 'Apple']
camera_videos = [r for r in results if r['has_camera']]
gps_videos = [r for r in results if r['has_gps']]

print(f"\nApple videos: {len(apple_videos)}")
print(f"Videos with camera data: {len(camera_videos)}")
print(f"Videos with GPS: {len(gps_videos)}")
```

Command Line Usage

Test the classifier from command line:

```
python video_classifier.py path/to/video.mp4
```

Output:

```
Analyzing: path/to/video.mp4
--------------------------------------------------
✓ Resolution:    4K
✓ Orientation:   V (Vertical)
✓ FPS Category:  60fps
✓ Actual FPS:    59.94
✓ Dimensions:    2160x3840
✓ Make:          Apple
```

```
✓ Model:         iPhone 14 Pro
✓ Has Camera Data: Yes
✓ Has GPS Data:    Yes

Filename format: 4K V60
```

🪲 Troubleshooting

"ffprobe not found"

Problem: FFmpeg isn't installed or not in PATH

Solution:

1. Install FFmpeg (see Installation section)
2. Verify with: `ffprobe -version`
3. On Windows, make sure FFmpeg is in your PATH

"No video stream found"

Problem: File is corrupted or not a video

Solution:

- Verify the file plays in a video player
- Check file isn't corrupted
- Try a different video file

GUI Freezes During Processing

Problem: This shouldn't happen - we use background threads

Solution:

- If it does freeze, there might be an issue with the video files
- Check the log output for errors
- Try processing a smaller folder first

Filters Show No Results

Problem: No videos match all active filters

Solution:

- Uncheck some filters (they create an AND condition)

- Check if your videos actually have the metadata you're filtering for
- Try "Filter by Make" only to see which videos have manufacturer info

Export Fails

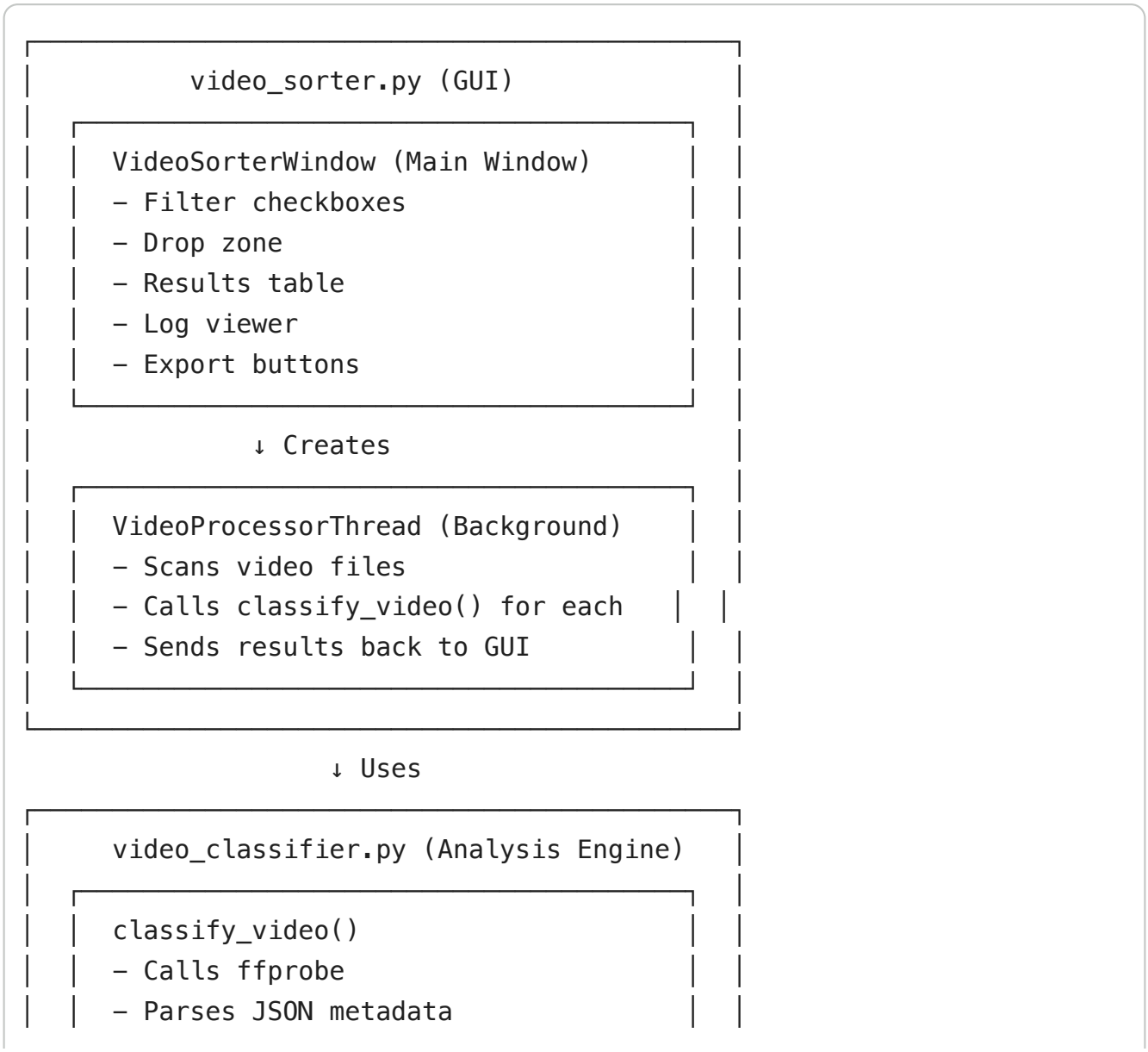
Problem: Can't write to the selected location

Solution:

- Check you have write permissions to that folder
- Try saving to your Desktop or Documents folder
- Close any programs that might have the file open (like Excel for CSV)

📚 Understanding the Code

Architecture Overview



```
│   │  - Classifies resolution/orientation  │   │
│   │  - Extracts device info                │   │
│   │  - Returns structured data             │   │
│   └────────────────────────────────────────┘   │
└────────────────────────────────────────────────┘

                    ↓ Uses

┌────────────────────────────────────────────────┐
│              FFmpeg (ffprobe)                  │
│  - Industry standard video analysis            │
│  - Extracts metadata from video files          │
│  - Returns JSON with all video properties      │
└────────────────────────────────────────────────┘
```

Why This Design?

Separation of Concerns:

- `video_classifier.py` - Pure analysis logic (no GUI code)
- `video_sorter.py` - Pure GUI logic (delegates analysis)

Benefits:

- Can use classifier in other projects
- Can test classifier independently
- Can add different GUIs (web, CLI, etc.)
- Easier to maintain and debug

Background Threading:

- GUI runs in main thread
- Video processing runs in background thread
- Prevents GUI freezing during long operations
- User can see progress in real-time

🤔 Common Questions

Q: Can I process videos on a network drive?

A: Yes, but it will be slower. Processing time depends on network speed.

Q: How many videos can it handle?

A: Tested with 1000+ videos. The limiting factor is processing time, not the

application.

Q: Can I filter by specific makes/models?

A: Currently, filters are binary (has make? yes/no). To filter specific values, export to CSV and use Excel/spreadsheet software.

Q: Does it modify my video files?

A: No! It only reads metadata. Your original files are never changed.

Q: Why do some videos show "Unknown" for Make/Model?

A: Not all videos have device metadata. It depends on:

- How the video was created
- Device capabilities
- Whether metadata was preserved during editing/transfers

Q: Can I run this on a server without a GUI?

A: Yes! Use `video_classifier.py` directly in your own scripts. The GUI (`video_sorter.py`) is optional.

📝 Feature Roadmap

Potential future enhancements:

- ▢ Filter by specific make values (dropdown: Apple, Samsung, etc.)
- ▢ Filter by specific model values (dropdown: iPhone 14 Pro, Galaxy S23, etc.)
- ▢ Bulk rename files based on metadata
- ▢ Move/copy files to organized folders
- ▢ Custom export templates
- ▢ Batch metadata editing
- ▢ Video thumbnail previews
- ▢ Timeline scrubbing

🤝 Contributing

Want to improve this application?

Common improvements:

1. Add more filter options
2. Improve metadata detection
3. Add new export formats
4. Enhance the GUI layout
5. Add keyboard shortcuts

Testing changes:

1. Test with various video formats
2. Test with large folders (100+ videos)
3. Test filters in different combinations
4. Test export with different filenames

📄 License

This application is provided as-is for educational and personal use.

Dependencies:

- PySide6: LGPL license
- FFmpeg: LGPL/GPL license (depending on build)

🆘 Getting Help

If you encounter issues:

1. **Check this README** - Most common issues are covered
2. **Check FFmpeg** - Run `ffprobe -version` to verify it's working
3. **Check the log** - The GUI shows detailed processing logs
4. **Test a single video** - Run `python video_classifier.py test.mp4` to isolate issues
5. **Check file permissions** - Ensure you can read the video files

🎓 Learning Resources

Want to understand how this works?

Video Metadata

- FFmpeg Documentation
- Video container formats

Python GUI Development

- PySide6 Documentation
- Qt Tutorials

Python Threading

- Threading in Python
- Qt Threads

Version: 1.0.0
Last Updated: 2025 **Author**: RazorBackRoar

Happy Video Sorting! 🎬# New Section

""" Video Classification Module - Production Ready

This module analyzes video files and extracts:

- Resolution (4K, 1080p, 720p, HD, SD)
- Orientation (Vertical or Wide)
- Framerate (30fps or 60fps category)
- Device metadata (Make, Model, Camera, GPS)

WHY THIS APPROACH:

- Uses ffprobe (part of ffmpeg) because it's industry standard
- Returns structured data that's easy to filter
- Handles rotation metadata correctly (important for iPhone videos)
- Gracefully handles errors without crashing

INSTALL FFMPEG FIRST: macOS: brew install ffmpeg Linux: sudo apt-get install ffmpeg Windows: choco install ffmpeg """

import subprocess import json from typing import Dict, Tuple, Optional from pathlib import Path

def classify_video(file_path: str) -> Dict: """ MAIN FUNCTION - Analyzes a video file completely

```
This is the only function you need to call. It returns everything
about the video in one dictionary.
```

```
Args:
    file_path: Path to video file (e.g., "videos/myvideo.mp4")

Returns:
    Dictionary containing:
    {
        'resolution': '4K' | '1080p' | '720p' | 'HD' | 'SD',
        'orientation': 'V' (vertical) | 'W' (wide),
        'framerate_category': 30 | 60,
        'actual_fps': 59.94,  # Exact framerate
        'width': 2160,
        'height': 3840,
        'make': 'Apple',      # Device manufacturer (or None)
        'model': 'iPhone 14 Pro',  # Device model (or None)
        'has_camera': True,   # Has camera metadata?
        'has_gps': True,      # Has GPS coordinates?
        'success': True,
        'error': None
    }

Example:
    result = classify_video("my_iphone_video.mov")
    if result['success']:
        # Filter by make
        if result['make'] == 'Apple':
            print("This is an Apple video!")

        # Filter by camera
        if result['has_camera']:
            print("Has camera metadata!")

        # Get filename format
        print(f"{result['resolution']} {result['orientation']}{result['fr
"""
# Step 1: Get raw metadata from video file
metadata = _get_video_metadata(file_path)

# Step 2: Check if metadata extraction failed
if not metadata['success']:
    return {
```

```
        'resolution': 'SD',
        'orientation': 'W',
        'framerate_category': 30,
        'actual_fps': 0.0,
        'width': 0,
        'height': 0,
        'make': None,
        'model': None,
        'has_camera': False,
        'has_gps': False,
        'success': False,
        'error': metadata['error']
    }

# Step 3: Extract basic video properties
width = metadata['width']
height = metadata['height']
fps = metadata['fps']
rotation = metadata['rotation']

# Step 4: Apply rotation to get actual display dimensions
# WHY: iPhone portrait videos store as 1920x1080 with rotation=90
#      We need to swap to 1080x1920 to get actual display size
actual_width, actual_height = _apply_rotation(width, height, rotation)

# Step 5: Classify the video
resolution = _classify_resolution(actual_width, actual_height)
orientation = _get_orientation(actual_width, actual_height)
framerate_category = _classify_framerate(fps)

# Step 6: Extract device metadata
make = metadata.get('make')
model = metadata.get('model')
has_camera = metadata.get('has_camera', False)
has_gps = metadata.get('has_gps', False)

# Step 7: Return complete classification
return {
    'resolution': resolution,
    'orientation': orientation,
    'framerate_category': framerate_category,
```

```
    'actual_fps': round(fps, 2) if fps else 0.0,
    'width': actual_width,
    'height': actual_height,
    'make': make,
    'model': model,
    'has_camera': has_camera,
    'has_gps': has_gps,
    'success': True,
    'error': None
}
```

==========

==========

INTERNAL FUNCTIONS - These handle the heavy lifting

==========

==========

def _get_video_metadata(file_path: str) -> Dict: """ Extract ALL metadata from video using ffprobe.

```
WHY FFPROBE:
- It's the standard tool for video analysis
- Returns structured JSON data
- Handles all video formats
- Extracts device metadata (make, model, GPS)

Returns dictionary with raw metadata or error info
"""
cmd = [
    'ffprobe',
    '-v', 'quiet',              # Suppress ffprobe's own output
    '-print_format', 'json',     # Get structured JSON output
    '-show_streams',             # Show video/audio streams
```

```
    '-show_format',             # Show container format (has GPS/device
    file_path
]

try:
    # Run ffprobe command
    result = subprocess.run(cmd, capture_output=True, text=True, check=Tr
    data = json.loads(result.stdout)

    # Find the video stream (ignore audio/subtitle streams)
    video_stream = None
    for stream in data.get('streams', []):
        if stream.get('codec_type') == 'video':
            video_stream = stream
            break

    if not video_stream:
        return {'success': False, 'error': 'No video stream found in file

    # Extract basic dimensions
    width = int(video_stream.get('width', 0))
    height = int(video_stream.get('height', 0))

    # Extract FPS (comes as fraction like "30000/1001" = 29.97fps)
    # WHY: Video framerates are stored as ratios for precision
    fps_str = video_stream.get('r_frame_rate', '0/1')
    numerator, denominator = map(int, fps_str.split('/'))
    fps = numerator / denominator if denominator != 0 else 0.0

    # Extract rotation (iPhone videos often have this)
    rotation = 0
    if 'tags' in video_stream and 'rotate' in video_stream['tags']:
        try:
            rotation = int(video_stream['tags']['rotate'])
        except (ValueError, TypeError):
            rotation = 0

    # Extract device metadata from format tags
    # WHY: Device info is stored in the container format, not the stream
    format_tags = data.get('format', {}).get('tags', {})
```

```
    # Normalize tag keys (some use 'com.apple.quicktime.make', others use
    normalized_tags = {}
    for key, value in format_tags.items():
        # Convert to lowercase and remove prefixes
        clean_key = key.lower().split('.')[-1]
        normalized_tags[clean_key] = value

    # Extract make (manufacturer) - Apple, Samsung, GoPro, etc.
    make = normalized_tags.get('make') or normalized_tags.get('manufactur

    # Extract model - iPhone 14 Pro, Galaxy S23, HERO11 Black, etc.
    model = normalized_tags.get('model')

    # Check for camera metadata
    # WHY: Camera metadata indicates professional/prosumer footage
    has_camera = any([
        normalized_tags.get('camera'),
        normalized_tags.get('lens'),
        normalized_tags.get('lens_model'),
        normalized_tags.get('focal_length')
    ])

    # Check for GPS coordinates
    # WHY: GPS indicates location-tagged footage (useful for travel videc
    has_gps = any([
        normalized_tags.get('location'),
        normalized_tags.get('gps'),
        normalized_tags.get('location-eng'),
        'location' in str(format_tags).lower()
    ])

    return {
        'success': True,
        'width': width,
        'height': height,
        'fps': fps,
        'rotation': rotation,
        'make': make,
        'model': model,
        'has_camera': has_camera,
        'has_gps': has_gps
```

```
    }

except FileNotFoundError:
    return {
        'success': False,
        'error': 'ffprobe not found. Install ffmpeg:\n'
                 '  macOS: brew install ffmpeg\n'
                 '  Linux: sudo apt-get install ffmpeg\n'
                 '  Windows: choco install ffmpeg'
    }
except json.JSONDecodeError:
    return {'success': False, 'error': 'Could not parse video metadata'}
except Exception as e:
    return {'success': False, 'error': f'Error reading video: {str(e)}'}
```

def _apply_rotation(width: int, height: int, rotation: int) -> Tuple[int, int]: """ Swap width/height if video is rotated 90 or 270 degrees.

```
WHY THIS MATTERS:
iPhone portrait videos are stored as 1920x1080 with rotation=90
We need to swap these to 1080x1920 to get the actual display dimensions
Otherwise we'd classify portrait videos as landscape!

ROTATION VALUES:
- 0: Normal landscape
- 90: Portrait (rotate 90° clockwise to view correctly)
- 180: Upside down
- 270: Portrait (rotate 270° clockwise to view correctly)
"""
try:
    rotation = int(rotation) if rotation else 0
except (ValueError, TypeError):
    rotation = 0

# Swap dimensions for 90° and 270° rotations
if rotation in (90, 270):
    return height, width

return width, height
```

def _classify_resolution(width: int, height: int) -> str: """ Classify into: 4K, 1080p, 720p, HD, or SD

```
WHY CHECK BOTH DIMENSIONS:
- Landscape 4K: 3840x2160 (width is bigger)
- Portrait 4K: 2160x3840 (height is bigger)
We check both to catch all cases

RESOLUTION STANDARDS:
- 4K: 3840x2160 or higher
- 1080p: Exactly 1920x1080
- 720p: Exactly 1280x720
- HD: Anything between 1080p and 4K
- SD: Everything else (standard definition)
"""
if not isinstance(width, int) or not isinstance(height, int):
    return "SD"

if width <= 0 or height <= 0:
    return "SD"

# 4K check (includes 2160p in either dimension)
if width >= 3840 or height >= 3840 or width >= 2160 or height >= 2160:
    return "4K"

# 1080p check (exact match in either dimension)
elif width == 1920 or height == 1920 or width == 1080 or height == 1080:
    return "1080p"

# 720p check (exact match in either dimension)
elif width == 1280 or height == 1280 or width == 720 or height == 720:
    return "720p"

# HD (anything bigger than 1080 but not quite 4K)
elif width > 1080 or height > 1080:
    return "HD"

# Everything else is SD
else:
    return "SD"
```

def _get_orientation(width: int, height: int) -> str: """ Determine if video is Vertical (portrait) or Wide (landscape).

```
SIMPLE RULE:
- V (Vertical): height > width (e.g., 1080x1920 phone video)
- W (Wide): width >= height (e.g., 1920x1080 normal video)

WHY '>=' for Wide:
Square videos (1080x1080) should be considered Wide
"""
return 'V' if height > width else 'W'
```

def _classify_framerate(fps: float) -> int: """ Classify as 30fps or 60fps category.

```
WHY 45 IS THE THRESHOLD:
- Standard framerates: 24, 25, 29.97, 30
- High framerates: 50, 59.94, 60, 120, 240
- 45fps is the middle ground that separates them

COMMON FRAMERATES:
- 23.976: Film (24fps category → 30)
- 25: PAL video (30)
- 29.97: NTSC video (30)
- 30: Standard video (30)
- 50: PAL high framerate (60)
- 59.94: NTSC high framerate (60)
- 60: High framerate (60)
- 120/240: Slow motion (60)
"""
if fps is None or fps == 0:
    return 30

try:
    fps_float = float(fps)
except (ValueError, TypeError):
    return 30

return 60 if fps_float > 45.0 else 30
```

==========

==========

TESTING / COMMAND LINE USAGE

==========

==========

if **name** == "**main**": """ Test the classifier with a video file.

```
USAGE:
python video_classifier.py path/to/video.mp4
"""
import sys

if len(sys.argv) > 1:
    video_path = sys.argv[1]
    print(f"\nAnalyzing: {video_path}")
    print("-" * 60)

    result = classify_video(video_path)

    if result['success']:
        print(f"✓ Resolution:      {result['resolution']}")
        print(f"✓ Orientation:     {result['orientation']} ({'Vertical' i
        print(f"✓ FPS Category:    {result['framerate_category']}fps")
        print(f"✓ Actual FPS:      {result['actual_fps']}")
        print(f"✓ Dimensions:      {result['width']}x{result['height']}")
        print(f"✓ Make:            {result['make'] or 'Unknown'}")
        print(f"✓ Model:           {result['model'] or 'Unknown'}")
        print(f"✓ Has Camera Data: {'Yes' if result['has_camera'] else 'N
        print(f"✓ Has GPS Data:    {'Yes' if result['has_gps'] else 'No'}
        print(f"\nFilename format: {result['resolution']} {result['orient
    else:
        print(f"✗ Error: {result['error']}")
else:
    print("Usage: python video_classifier.py path/to/video.mp4")
    print("\nOr import in your code:")
    print("  from video_classifier import classify_video")
    print("  result = classify_video('video.mp4')")# New Section
```

""" Video Sorter - Complete GUI Application

A desktop application for organizing and filtering video files.

FEATURES:

- Drag & drop video folders
- Smart filtering (Make, Model, Camera, GPS)
- Export to CSV or TXT
- Live preview of results
- Shows Python code structure

REQUIREMENTS: pip install PySide6

WHY PYSIDE6:

- Cross-platform (Windows, Mac, Linux)
- Native look and feel
- Professional desktop apps use it
- Great performance
- Free (LGPL license)

USAGE: python video_sorter.py """

import sys import os import csv from pathlib import Path from typing import List, Dict from PySide6.QtWidgets import (QApplication, QMainWindow, QWidget, QVBoxLayout, QHBoxLayout, QPushButton, QLabel, QTableWidget, QTableWidgetItem, QCheckBox, QGroupBox, QTextEdit, QFileDialog, QMessageBox, QProgressBar, QHeaderView) from PySide6.QtCore import Qt, QThread, Signal from PySide6.QtGui import QDragEnterEvent, QDropEvent

Import our video classifier

try: from video_classifier import classify_video except ImportError: print("ERROR: Cannot import video_classifier.py") print("Make sure video_classifier.py is in the same folder!") sys.exit(1)

==========

==========

WORKER THREAD - Processes videos without freezing the GUI

==========

==========

class VideoProcessorThread(QThread): """ WHY A SEPARATE THREAD: - Processing videos takes time (seconds to minutes) - If we do it in the main thread, the GUI freezes - User can't click buttons or see updates - This thread runs processing in the background

```
SIGNALS:
Signals let the background thread communicate with the GUI safely.
Qt doesn't allow direct GUI updates from background threads.
"""

# Signal: Send progress updates to GUI
progress = Signal(int, int)  # (current, total)

# Signal: Send a log message to GUI
log = Signal(str)

# Signal: Send a processed video result to GUI
result = Signal(dict)

# Signal: Tell GUI we're done processing
finished = Signal()

def __init__(self, video_files: List[str]):
    super().__init__()
    self.video_files = video_files
    self.should_stop = False  # Flag to stop processing early

def run(self):
    """
    This method runs in the background thread.
```

```
    DON'T update GUI directly here - use signals instead!
    """
    total = len(self.video_files)

    for i, video_path in enumerate(self.video_files):
        # Check if user wants to stop
        if self.should_stop:
            self.log.emit("Processing stopped by user")
            break

        # Update progress
        self.progress.emit(i + 1, total)
        self.log.emit(f"Processing: {Path(video_path).name}")

        # Classify the video
        result = classify_video(video_path)

        # Add filename to result
        result['filename'] = Path(video_path).name
        result['filepath'] = video_path

        # Send result to GUI
        if result['success']:
            self.result.emit(result)
            self.log.emit(f"✓ Completed: {result['filename']}")
        else:
            self.log.emit(f"✗ Error: {result['filename']} - {result['erro

    # Tell GUI we're done
    self.finished.emit()

def stop(self):
    """Call this to stop processing early"""
    self.should_stop = True
```

==========

==========

MAIN WINDOW - The GUI

==========

==========

class VideoSorterWindow(QMainWindow): """ The main application window.

```
LAYOUT STRUCTURE:
┌─────────────────────────────────────────┐
│ Filters (checkboxes)                    │
├─────────────────────────────────────────┤
│ Drop Zone / Select Folder               │
├─────────────────────────────────────────┤
│ Results Table (filterable)              │
├─────────────────────────────────────────┤
│ Log / Terminal Output                   │
├─────────────────────────────────────────┤
│ Export Buttons | Show Code              │
└─────────────────────────────────────────┘
"""

def __init__(self):
    super().__init__()

    # Store all processed videos (before filtering)
    self.all_videos: List[Dict] = []

    # Background thread for processing
    self.processor_thread = None

    # Setup the GUI
    self.init_ui()

def init_ui(self):
    """Initialize the user interface"""
    self.setWindowTitle("Video Sorter")
    self.setMinimumSize(1200, 800)

    # Create main widget and layout
```

```
    main_widget = QWidget()
    self.setCentralWidget(main_widget)
    layout = QVBoxLayout(main_widget)

    # Add all sections
    layout.addWidget(self._create_filter_section())
    layout.addWidget(self._create_drop_zone())
    layout.addWidget(self._create_results_table())
    layout.addWidget(self._create_log_section())
    layout.addLayout(self._create_button_section())

    # Enable drag & drop
    self.setAcceptDrops(True)

# ============================================================================
# UI COMPONENT CREATION
# ============================================================================

def _create_filter_section(self) -> QGroupBox:
    """
    Create the filter checkboxes section.

    WHY THIS DESIGN:
    - Filters are independent toggles
    - Multiple filters create an AND condition
    - Example: Make=Apple AND Camera=Yes shows only Apple videos with cam
    - If no filters are checked, show everything
    """
    group = QGroupBox("Filters")
    layout = QHBoxLayout()

    # Make filter - Filter by device manufacturer
    self.filter_make = QCheckBox("Filter by Make")
    self.filter_make.setToolTip("Show only specific device manufacturers
    self.filter_make.stateChanged.connect(self._apply_filters)

    # Model filter - Filter by device model
    self.filter_model = QCheckBox("Filter by Model")
    self.filter_model.setToolTip("Show only specific device models (iPhor
    self.filter_model.stateChanged.connect(self._apply_filters)
```

```
    # Camera filter - Has camera metadata
    self.filter_camera = QCheckBox("Has Camera Metadata")
    self.filter_camera.setToolTip("Show only videos with camera informati
    self.filter_camera.stateChanged.connect(self._apply_filters)

    # GPS filter - Has GPS coordinates
    self.filter_gps = QCheckBox("Has GPS Data")
    self.filter_gps.setToolTip("Show only videos with GPS location data")
    self.filter_gps.stateChanged.connect(self._apply_filters)

    layout.addWidget(self.filter_make)
    layout.addWidget(self.filter_model)
    layout.addWidget(self.filter_camera)
    layout.addWidget(self.filter_gps)
    layout.addStretch()  # Push filters to the left

    group.setLayout(layout)
    return group

def _create_drop_zone(self) -> QGroupBox:
    """
    Create the drag & drop zone / folder selection area.

    WHY BOTH DRAG/DROP AND BUTTON:
    - Power users prefer drag & drop (faster)
    - Some users prefer clicking a button (more discoverable)
    - We support both for best UX
    """
    group = QGroupBox("Video Source")
    layout = QVBoxLayout()

    # Drop zone label
    self.drop_label = QLabel("Drag & Drop Video Folder Here\n\nOr")
    self.drop_label.setAlignment(Qt.AlignCenter)
    self.drop_label.setMinimumHeight(100)
    self.drop_label.setStyleSheet("""
        QLabel {
            border: 2px dashed #999;
            border-radius: 5px;
            background-color: #f5f5f5;
            font-size: 14pt;
```

```
            color: #666;
        }
    """)

    # Select folder button
    self.select_button = QPushButton("Select Folder")
    self.select_button.clicked.connect(self._select_folder)

    # Progress bar (hidden initially)
    self.progress_bar = QProgressBar()
    self.progress_bar.setVisible(False)

    layout.addWidget(self.drop_label)
    layout.addWidget(self.select_button)
    layout.addWidget(self.progress_bar)

    group.setLayout(layout)
    return group

def _create_results_table(self) -> QGroupBox:
    """
    Create the results table showing all videos.

    COLUMNS:
    - Filename: Original video filename
    - Resolution: 4K, 1080p, 720p, HD, SD
    - Orientation: V (vertical) or W (wide)
    - FPS: 30 or 60
    - Make: Apple, Samsung, GoPro, etc.
    - Model: iPhone 14 Pro, Galaxy S23, etc.
    - Camera: Yes/No (has camera metadata)
    - GPS: Yes/No (has GPS data)
    """
    group = QGroupBox("Results")
    layout = QVBoxLayout()

    # Create table
    self.results_table = QTableWidget(0, 8)  # 0 rows, 8 columns
    self.results_table.setHorizontalHeaderLabels([
        "Filename", "Resolution", "Orientation", "FPS",
        "Make", "Model", "Camera", "GPS"
```

```
    ])

    # Make table look nice
    header = self.results_table.horizontalHeader()
    header.setSectionResizeMode(0, QHeaderView.Stretch)  # Filename stret
    for i in range(1, 8):
        header.setSectionResizeMode(i, QHeaderView.ResizeToContents)

    # Enable sorting
    self.results_table.setSortingEnabled(True)

    # Alternating row colors for readability
    self.results_table.setAlternatingRowColors(True)

    layout.addWidget(self.results_table)
    group.setLayout(layout)
    return group

def _create_log_section(self) -> QGroupBox:
    """
    Create the log/terminal output section.

    WHY SHOW LOGS:
    - User can see what's happening in real-time
    - Shows which videos succeeded/failed
    - Helps debug issues
    - Makes the app feel responsive
    """
    group = QGroupBox("Processing Log")
    layout = QVBoxLayout()

    self.log_text = QTextEdit()
    self.log_text.setReadOnly(True)
    self.log_text.setMaximumHeight(150)
    self.log_text.setStyleSheet("""
        QTextEdit {
            font-family: monospace;
            background-color: #2b2b2b;
            color: #d4d4d4;
        }
    """)
```

```
    layout.addWidget(self.log_text)
    group.setLayout(layout)
    return group

def _create_button_section(self) -> QHBoxLayout:
    """
    Create the bottom button section.

    BUTTONS:
    - Export CSV: Export filtered results as CSV
    - Export TXT: Export filtered results as text file
    - Show Python Code: Display the code structure
    - Clear: Clear all results
    """
    layout = QHBoxLayout()

    # Export buttons
    self.export_csv_btn = QPushButton("Export CSV")
    self.export_csv_btn.clicked.connect(lambda: self._export_results('csv
    self.export_csv_btn.setEnabled(False)  # Disabled until we have resul

    self.export_txt_btn = QPushButton("Export TXT")
    self.export_txt_btn.clicked.connect(lambda: self._export_results('txt
    self.export_txt_btn.setEnabled(False)

    # Show code button
    self.show_code_btn = QPushButton("Show Python Code")
    self.show_code_btn.clicked.connect(self._show_code)

    # Clear button
    self.clear_btn = QPushButton("Clear Results")
    self.clear_btn.clicked.connect(self._clear_results)
    self.clear_btn.setEnabled(False)

    layout.addWidget(self.export_csv_btn)
    layout.addWidget(self.export_txt_btn)
    layout.addStretch()
    layout.addWidget(self.show_code_btn)
    layout.addWidget(self.clear_btn)
```

```
    return layout

# ============================================================================
# DRAG & DROP HANDLING
# ============================================================================

def dragEnterEvent(self, event: QDragEnterEvent):
    """
    Called when user drags something over the window.

    WHY CHECK mimeData:
    - We only want to accept folders, not random files
    - urls() contains the dragged paths
    """
    if event.mimeData().hasUrls():
        event.acceptProposedAction()

def dropEvent(self, event: QDropEvent):
    """
    Called when user drops something on the window.

    PROCESS:
    1. Get all dropped paths
    2. Check if any are folders
    3. Find video files in those folders
    4. Start processing
    """
    urls = event.mimeData().urls()

    for url in urls:
        path = url.toLocalFile()

        if os.path.isdir(path):
            self._process_folder(path)
            return

    self._log("Please drop a folder, not individual files")

# ============================================================================
# FILE PROCESSING
# ============================================================================
```

```
def _select_folder(self):
    """User clicked 'Select Folder' button"""
    folder = QFileDialog.getExistingDirectory(
        self,
        "Select Video Folder",
        "",
        QFileDialog.ShowDirsOnly
    )

    if folder:
        self._process_folder(folder)

def _process_folder(self, folder_path: str):
    """
    Process all video files in a folder.

    STEPS:
    1. Find all video files (mp4, mov, avi, mkv, etc.)
    2. Start background thread to process them
    3. Update UI as each video completes
    """
    # Find video files
    video_extensions = {'.mp4', '.mov', '.avi', '.mkv', '.m4v', '.flv', '
    video_files = []

    self._log(f"Scanning folder: {folder_path}")

    for root, dirs, files in os.walk(folder_path):
        for file in files:
            if Path(file).suffix.lower() in video_extensions:
                video_files.append(os.path.join(root, file))

    if not video_files:
        self._log("No video files found in folder")
        QMessageBox.warning(
            self,
            "No Videos Found",
            f"No video files found in:\n{folder_path}"
        )
        return
```

```
    self._log(f"Found {len(video_files)} video files")

    # Clear previous results
    self._clear_results()

    # Disable buttons during processing
    self.select_button.setEnabled(False)
    self.export_csv_btn.setEnabled(False)
    self.export_txt_btn.setEnabled(False)

    # Show progress bar
    self.progress_bar.setVisible(True)
    self.progress_bar.setMaximum(len(video_files))
    self.progress_bar.setValue(0)

    # Start background processing
    self.processor_thread = VideoProcessorThread(video_files)
    self.processor_thread.progress.connect(self._update_progress)
    self.processor_thread.log.connect(self._log)
    self.processor_thread.result.connect(self._add_video_result)
    self.processor_thread.finished.connect(self._processing_finished)
    self.processor_thread.start()

def _update_progress(self, current: int, total: int):
    """Update progress bar"""
    self.progress_bar.setValue(current)

def _add_video_result(self, result: Dict):
    """
    Add a video result to our data and table.

    WHY TWO STEPS:
    1. Add to self.all_videos (unfiltered data)
    2. Check if it passes current filters
    3. If yes, add to table

    This way, changing filters doesn't require re-processing videos.
    """
    self.all_videos.append(result)
```

```
    # Check if this video passes current filters
    if self._video_passes_filters(result):
        self._add_video_to_table(result)

def _add_video_to_table(self, result: Dict):
    """Add a video row to the results table"""
    row = self.results_table.rowCount()
    self.results_table.insertRow(row)

    # Add cells
    self.results_table.setItem(row, 0, QTableWidgetItem(result['filename'
    self.results_table.setItem(row, 1, QTableWidgetItem(result['resolutio
    self.results_table.setItem(row, 2, QTableWidgetItem(result['orientati
    self.results_table.setItem(row, 3, QTableWidgetItem(str(result['frame
    self.results_table.setItem(row, 4, QTableWidgetItem(result['make'] or
    self.results_table.setItem(row, 5, QTableWidgetItem(result['model'] o
    self.results_table.setItem(row, 6, QTableWidgetItem('Yes' if result['
    self.results_table.setItem(row, 7, QTableWidgetItem('Yes' if result['

def _processing_finished(self):
    """Called when all videos have been processed"""
    self._log(f"Processing complete! {len(self.all_videos)} videos analyz

    # Re-enable buttons
    self.select_button.setEnabled(True)
    self.export_csv_btn.setEnabled(True)
    self.export_txt_btn.setEnabled(True)
    self.clear_btn.setEnabled(True)

    # Hide progress bar
    self.progress_bar.setVisible(False)

    # Show summary
    QMessageBox.information(
        self,
        "Processing Complete",
        f"Successfully processed {len(self.all_videos)} videos!"
    )

# ============================================================================
# FILTERING
```

```
# ============================================================================

def _video_passes_filters(self, video: Dict) -> bool:
    """
    Check if a video passes all active filters.

    FILTER LOGIC (AND condition):
    - If NO filters are active: return True (show everything)
    - If filter_make is checked: video must have a make
    - If filter_model is checked: video must have a model
    - If filter_camera is checked: video must have camera data
    - If filter_gps is checked: video must have GPS data

    ALL active filters must pass for the video to be shown.
    """
    # If no filters are active, show everything
    if not any([
        self.filter_make.isChecked(),
        self.filter_model.isChecked(),
        self.filter_camera.isChecked(),
        self.filter_gps.isChecked()
    ]):
        return True

    # Check each active filter
    if self.filter_make.isChecked() and not video['make']:
        return False

    if self.filter_model.isChecked() and not video['model']:
        return False

    if self.filter_camera.isChecked() and not video['has_camera']:
        return False

    if self.filter_gps.isChecked() and not video['has_gps']:
        return False

    # All active filters passed
    return True

def _apply_filters(self):
```

```
    """
    Re-filter the table based on current filter checkboxes.

    WHY REBUILD THE TABLE:
    - We could hide/show rows, but that's complex
    - Rebuilding is simpler and fast enough
    - We keep all data in self.all_videos
    """
    # Clear table
    self.results_table.setRowCount(0)

    # Re-add videos that pass filters
    for video in self.all_videos:
        if self._video_passes_filters(video):
            self._add_video_to_table(video)

    # Update log
    visible_count = self.results_table.rowCount()
    total_count = len(self.all_videos)
    self._log(f"Filters applied: Showing {visible_count} of {total_count}

# ============================================================================
# EXPORT
# ============================================================================

def _export_results(self, format_type: str):
    """
    Export filtered results to CSV or TXT.

    WHY FILTERED RESULTS:
    - User might have filters active
    - They want to export what they see
    - Not the entire dataset
    """
    # Get visible videos (those in table after filtering)
    visible_videos = []
    for row in range(self.results_table.rowCount()):
        filename = self.results_table.item(row, 0).text()

        # Find the full video data
        for video in self.all_videos:
```

```
            if video['filename'] == filename:
                visible_videos.append(video)
                break

    if not visible_videos:
        QMessageBox.warning(self, "No Results", "No videos to export")
        return

    # Ask where to save
    if format_type == 'csv':
        filepath, _ = QFileDialog.getSaveFileName(
            self, "Export CSV", "video_results.csv", "CSV Files (*.csv)"
        )
        if filepath:
            self._export_csv(visible_videos, filepath)
    else:
        filepath, _ = QFileDialog.getSaveFileName(
            self, "Export TXT", "video_results.txt", "Text Files (*.txt)"
        )
        if filepath:
            self._export_txt(visible_videos, filepath)

def _export_csv(self, videos: List[Dict], filepath: str):
    """Export to CSV format"""
    try:
        with open(filepath, 'w', newline='', encoding='utf-8') as f:
            writer = csv.DictWriter(f, fieldnames=[
                'filename', 'resolution', 'orientation', 'framerate_categ
                'actual_fps', 'width', 'height', 'make', 'model',
                'has_camera', 'has_gps', 'filepath'
            ])
            writer.writeheader()

            for video in videos:
                writer.writerow(video)

        self._log(f"Exported {len(videos)} videos to {filepath}")
        QMessageBox.information(
            self,
            "Export Successful",
            f"Exported {len(videos)} videos to:\n{filepath}"
```

```
        )
    except Exception as e:
        self._log(f"Export failed: {e}")
        QMessageBox.critical(self, "Export Failed", str(e))

def _export_txt(self, videos: List[Dict], filepath: str):
    """Export to TXT format (human-readable)"""
    try:
        with open(filepath, 'w', encoding='utf-8') as f:
            f.write("VIDEO SORTER RESULTS\n")
            f.write("=" * 80 + "\n\n")

            for i, video in enumerate(videos, 1):
                f.write(f"Video {i}: {video['filename']}\n")
                f.write("-" * 80 + "\n")
                f.write(f"  Resolution:    {video['resolution']}\n")
                f.write(f"  Orientation:   {video['orientation']}\n")
                f.write(f"  FPS Category:  {video['framerate_category']}\
                f.write(f"  Actual FPS:    {video['actual_fps']}\n")
                f.write(f"  Dimensions:    {video['width']}x{video['heigh
                f.write(f"  Make:          {video['make'] or 'Unknown'}\n
                f.write(f"  Model:         {video['model'] or 'Unknown'}\
                f.write(f"  Has Camera:    {'Yes' if video['has_camera']
                f.write(f"  Has GPS:       {'Yes' if video['has_gps'] els
                f.write(f"  Path:          {video['filepath']}\n")
                f.write("\n")

        self._log(f"Exported {len(videos)} videos to {filepath}")
        QMessageBox.information(
            self,
            "Export Successful",
            f"Exported {len(videos)} videos to:\n{filepath}"
        )
    except Exception as e:
        self._log(f"Export failed: {e}")
        QMessageBox.critical(self, "Export Failed", str(e))

# ============================================================================
# UTILITY FUNCTIONS
# ============================================================================
```

```
def _log(self, message: str):
    """Add a message to the log"""
    self.log_text.append(message)
    # Auto-scroll to bottom
    self.log_text.verticalScrollBar().setValue(
        self.log_text.verticalScrollBar().maximum()
    )

def _clear_results(self):
    """Clear all results and reset the GUI"""
    self.all_videos.clear()
    self.results_table.setRowCount(0)
    self.log_text.clear()
    self.export_csv_btn.setEnabled(False)
    self.export_txt_btn.setEnabled(False)
    self.clear_btn.setEnabled(False)

def _show_code(self):
    """Show the Python code structure in a dialog"""
    code_text = """
```

VIDEO SORTER CODE STRUCTURE

FILE 1: video_classifier.py

This file handles video analysis using ffprobe.

Key Functions:

- classify_video(file_path) → Main function, returns video metadata
- _get_video_metadata(file_path) → Extracts raw metadata with ffprobe
- _apply_rotation(width, height, rotation) → Handles iPhone rotation
- _classify_resolution(width, height) → Classifies as 4K/1080p/720p/HD/SD
- _get_orientation(width, height) → Returns V (vertical) or W (wide)
- _classify_framerate(fps) → Classifies as 30fps or 60fps category

Dependencies:

- ffprobe (part of ffmpeg)
- subprocess (Python standard library)
- json (Python standard library)

FILE 2: video_sorter.py (THIS FILE)

This file provides the GUI application.

Key Classes:

- VideoSorterWindow → Main window and application logic
- VideoProcessorThread → Background thread for processing videos

Main Components:

1. Filter Section: Checkboxes for Make, Model, Camera, GPS
2. Drop Zone: Drag & drop or select folder
3. Results Table: Displays all filtered videos
4. Log Section: Shows processing status
5. Export Buttons: Export to CSV or TXT

How It Works:

1. User drops a folder or clicks "Select Folder"
2. App scans for video files (.mp4, .mov, .avi, etc.)
3. VideoProcessorThread processes each video in background
4. classify_video() extracts metadata for each file
5. Results appear in table as they complete
6. User can filter results with checkboxes
7. User can export filtered results to CSV or TXT

Dependencies:

- PySide6 (install with: pip install PySide6)
- video_classifier.py (must be in same folder)

USAGE:

1. Make sure ffmpeg is installed: macOS: brew install ffmpeg Linux: sudo apt-get install ffmpeg Windows: choco install ffmpeg
2. Install PySide6: pip install PySide6
3. Run the application: python video_sorter.py
4. Drag & drop a folder with videos, or click "Select Folder"
5. Use filters to narrow results:
    - "Filter by Make": Show only videos with device manufacturer

- "Filter by Model": Show only videos with device model
- "Has Camera Metadata": Show only videos with camera info
- "Has GPS Data": Show only videos with GPS coordinates

6. Export results to CSV or TXT for further analysis """

```
# Create dialog to show code
dialog = QMessageBox(self)
dialog.setWindowTitle("Python Code Structure")
dialog.setText("Complete code structure and documentation:")
dialog.setDetailedText(code_text)
dialog.setStandardButtons(QMessageBox.Ok)
dialog.exec()
```

==========

==========

MAIN - Start the application

==========

==========

def main(): """ Entry point for the application.

```
STEPS:
1. Create QApplication (required for all Qt apps)
2. Create our main window
3. Show the window
4. Start the event loop (app.exec())
"""
app = QApplication(sys.argv)

# Set application-wide style
app.setStyle("Fusion")  # Modern, cross-platform style

# Create and show main window
window = VideoSorterWindow()
window.show()
```

```
# Start the Qt event loop
sys.exit(app.exec())
```

if **name** == "**main**": main()# New Section