Exercise 1.1

```
import math
import cv2
import numpy as np

class Car:
    """
    A class to represent a single tracked vehicle.
    """
    def __init__(self, car_id, x, y, w, h):
        self.car_id = car_id
        self.x = x
        self.y = y
        self.w = w
        self.h = h
        self.cx = int(x + w / 2)
        self.cy = int(y + h / 2)

class CarTracker:
    """
    A simple centroid-based tracker for assigning unique IDs to cars
    and maintaining their positions across multiple frames.

    This class works by calculating the Euclidean distance between the
    centroids of new detections and the centroids of currently tracked
objects.
    If a new detection is close enough to an existing object, it is
    considered the same object, and its position is updated. Otherwise,
    a new ID is assigned.
    """

    def __init__(self, max_distance):
        """
        Initializes the tracker.

        Args:
            max_distance (int): The maximum Euclidean distance a new
centroid
                                can be from an existing one to be considered
                                the same object.
```

```
        """
        self.tracked_objects = {}

        # A counter for assigning new unique IDs.
        self.next_object_id = 0

        # The maximum distance to consider a match.
        self.max_distance = max_distance

    def update(self, detections):
        """
        Takes a list of new detections (bounding boxes) and updates the
        list of tracked objects. The method updates the self.tracked_objects
        dictionary in place and does not return any value.

        Args:
            detections (list): A list of new bounding boxes in the format
                               [(x, y, w, h), ...].
        """
        matched_ids = []
        current_tracked_objects_copy = self.tracked_objects.copy()

        for new_bbox in detections:
            x, y, w, h = new_bbox
            new_cx = int(x + w / 2)
            new_cy = int(y + h / 2)

            is_new_object = True
            min_dist = float('inf')
            closest_id = -1

            for object_id, car_obj in current_tracked_objects_copy.items():
                distance = math.hypot(new_cx - car_obj.cx, new_cy -
car_obj.cy)

                if distance < min_dist:
                    min_dist = distance
                    closest_id = object_id

            if min_dist < self.max_distance:
                # Update the existing Car object with new detection data
                car_obj = self.tracked_objects[closest_id]
                car_obj.x, car_obj.y, car_obj.w, car_obj.h = x, y, w, h
                car_obj.cx, car_obj.cy = new_cx, new_cy
                matched_ids.append(closest_id)
                is_new_object = False
```

```
                if closest_id in current_tracked_objects_copy:
                    del current_tracked_objects_copy[closest_id]

            if is_new_object:
                new_id = self.next_object_id
                new_car = Car(new_id, x, y, w, h)
                self.tracked_objects[new_id] = new_car
                self.next_object_id += 1
                matched_ids.append(new_id)

        objects_to_remove = [
            obj_id for obj_id in self.tracked_objects if obj_id not in
matched_ids
        ]
        for obj_id in objects_to_remove:
            del self.tracked_objects[obj_id]

def region_of_interest(frame):
    height, width, _ = frame.shape
    mask = np.zeros((height, width), dtype=np.uint8)
    cv2.rectangle(mask, (0, height // 2), (width, height), 255, -1)
    mask = cv2.bitwise_and(frame, frame, mask=mask)
    return mask

def main():
    cap = cv2.VideoCapture("Traffic_Laramie_2.mp4")
    subtractor = cv2.createBackgroundSubtractorMOG2()

    tracker = CarTracker(max_distance=120)

    while True:
        ret, frame = cap.read()
        if not ret:
            break

        roi = region_of_interest(frame)

        mask = subtractor.apply(roi)
        _, mask = cv2.threshold(mask, 254, 255, cv2.THRESH_BINARY)
        contours, _ = cv2.findContours(mask, cv2.RETR_TREE,
cv2.CHAIN_APPROX_SIMPLE)

        detections = []
        for cnt in contours:
            area = cv2.contourArea(cnt)
```

```
            if area > 3500:
                x, y, w, h = cv2.boundingRect(cnt)
                detections.append((x, y, w, h))

        # Update the tracker with the new detections, but no return value
        tracker.update(detections)

        # Iterate through the stored Car objects
        for _, car in tracker.tracked_objects.items():
            # Draw the bounding box and ID
            cv2.rectangle(roi, (car.x, car.y), (car.x + car.w, car.y +
car.h), (0, 255, 0), 3)
            cv2.putText(roi, str(car.car_id), (car.x, car.y - 10),
cv2.FONT_HERSHEY_SIMPLEX, 0.9, (0, 255, 0), 2)

            # Draw a circle at the centroid of the car
            cv2.circle(roi, (car.cx, car.cy), 5, (139, 0, 139), -1)



        cv2.imshow("Mask", mask)
        cv2.imshow("Roi", roi)

        key = cv2.waitKey(2)
        if key == 27:
            break

    cap.release()
    cv2.destroyAllWindows()

    key = cv2.waitKey(0)
    if key == 27:
        exit()

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

Exercise 1.2

```
import math
import cv2
import numpy as np

class Car:
```

```
    """
    A class to represent a single tracked vehicle.
    """
    def __init__(self, car_id, x, y, w, h):
        self.car_id = car_id
        self.x = x
        self.y = y
        self.w = w
        self.h = h
        self.cx = int(x + w / 2)
        self.cy = int(y + h / 2)
        self.inside_box1 = False
        self.inside_box2 = False

class CarTracker:
    """
    A simple centroid-based tracker for assigning unique IDs to cars
    and maintaining their positions across multiple frames.

    This class works by calculating the Euclidean distance between the
    centroids of new detections and the centroids of currently tracked
objects.
    If a new detection is close enough to an existing object, it is
    considered the same object, and its position is updated. Otherwise,
    a new ID is assigned.
    """

    def __init__(self, max_distance):
        """
        Initializes the tracker.

        Args:
            max_distance (int): The maximum Euclidean distance a new
centroid
                                can be from an existing one to be considered
                                the same object.
        """
        self.tracked_objects = {}

        # A counter for assigning new unique IDs.
        self.next_object_id = 0

        # The maximum distance to consider a match.
        self.max_distance = max_distance

    def update(self, detections):
```

```
        """
        Takes a list of new detections (bounding boxes) and updates the
        list of tracked objects. The method updates the self.tracked_objects
        dictionary in place and does not return any value.

        Args:
            detections (list): A list of new bounding boxes in the format
                               [(x, y, w, h), ...].
        """
        matched_ids = []
        current_tracked_objects_copy = self.tracked_objects.copy()

        for new_bbox in detections:
            x, y, w, h = new_bbox
            new_cx = int(x + w / 2)
            new_cy = int(y + h / 2)

            is_new_object = True
            min_dist = float('inf')
            closest_id = -1

            for object_id, car_obj in current_tracked_objects_copy.items():
                distance = math.hypot(new_cx - car_obj.cx, new_cy -
car_obj.cy)

                if distance < min_dist:
                    min_dist = distance
                    closest_id = object_id

            if min_dist < self.max_distance:
                # Update the existing Car object with new detection data
                car_obj = self.tracked_objects[closest_id]
                car_obj.x, car_obj.y, car_obj.w, car_obj.h = x, y, w, h
                car_obj.cx, car_obj.cy = new_cx, new_cy
                matched_ids.append(closest_id)
                is_new_object = False

                if closest_id in current_tracked_objects_copy:
                    del current_tracked_objects_copy[closest_id]

            if is_new_object:
                new_id = self.next_object_id
                new_car = Car(new_id, x, y, w, h)
                self.tracked_objects[new_id] = new_car
                self.next_object_id += 1
                matched_ids.append(new_id)
```

```
        objects_to_remove = [
            obj_id for obj_id in self.tracked_objects if obj_id not in
matched_ids
        ]
        for obj_id in objects_to_remove:
            del self.tracked_objects[obj_id]

def region_of_interest(frame):
    height, width, _ = frame.shape
    mask = np.zeros((height, width), dtype=np.uint8)
    cv2.rectangle(mask, (0, height // 2), (width, height), 255, -1)
    mask = cv2.bitwise_and(frame, frame, mask=mask)
    return mask

def main():
    cap = cv2.VideoCapture("Traffic_Laramie_1.mp4")
    subtractor = cv2.createBackgroundSubtractorMOG2()

    tracker = CarTracker(max_distance=120)

    # Define the two boxes for left turn detection
    box1_coords = (400, 450, 550, 600)
    box2_coords = (700, 320, 1040, 425)

    left_turn_counter = 0

    # Get video properties for final calculation
    fps = cap.get(cv2.CAP_PROP_FPS)
    frame_count = int(cap.get(cv2.CAP_PROP_FRAME_COUNT))

    # Calculate video duration upfront
    if fps > 0:
        total_seconds = frame_count / fps
    else:
        total_seconds = 0

    while True:
        ret, frame = cap.read()
        if not ret:
            break

        roi = region_of_interest(frame)

        mask = subtractor.apply(roi)
        _, mask = cv2.threshold(mask, 254, 255, cv2.THRESH_BINARY)
```

```
        contours, _ = cv2.findContours(mask, cv2.RETR_TREE,
cv2.CHAIN_APPROX_SIMPLE)

        detections = []
        for cnt in contours:
            area = cv2.contourArea(cnt)
            if area > 3500:
                x, y, w, h = cv2.boundingRect(cnt)
                detections.append((x, y, w, h))

        # Update the tracker with the new detections, but no return value
        tracker.update(detections)

        # Draw the left turn detection boxes
        x1, y1, x2, y2 = box1_coords
        cv2.rectangle(roi, (x1, y1), (x2, y2), (255, 0, 0), 2)
        cv2.putText(roi, "Box 1", (x1, y1 - 5), cv2.FONT_HERSHEY_SIMPLEX,
0.5, (255, 0, 0), 1)

        x1, y1, x2, y2 = box2_coords
        cv2.rectangle(roi, (x1, y1), (x2, y2), (0, 0, 255), 2)
        cv2.putText(roi, "Box 2", (x1, y1 - 5), cv2.FONT_HERSHEY_SIMPLEX,
0.5, (0, 0, 255), 1)

        # Iterate through the stored Car objects
        for car_id, car in tracker.tracked_objects.items():
            # Check for left turn conditions
            if not car.inside_box1 and box1_coords[0] < car.cx <
box1_coords[2] and box1_coords[1] < car.cy < box1_coords[3]:
                print(f"{car_id} Entered Box1")
                car.inside_box1 = True

            if car.inside_box1 and box2_coords[0] < car.cx < box2_coords[2]
and box2_coords[1] < car.cy < box2_coords[3]:
                left_turn_counter += 1
                print(f"Car ID {car_id} made a left turn!")
                # Reset the flag to avoid double-counting
                car.inside_box1 = False

            # Draw the bounding box and ID
            cv2.rectangle(roi, (car.x, car.y), (car.x + car.w, car.y +
car.h), (0, 255, 0), 3)
            cv2.putText(roi, str(car.car_id), (car.x, car.y - 10),
cv2.FONT_HERSHEY_SIMPLEX, 0.9, (0, 255, 0), 2)

            # Draw a circle at the centroid of the car
```

```
            cv2.circle(roi, (car.cx, car.cy), 5, (139, 0, 139), -1)

        # Display the left turn counter in real-time
        cv2.putText(roi, f"Left Turns: {left_turn_counter}", (20, 50),
cv2.FONT_HERSHEY_SIMPLEX, 1, (255, 255, 255), 2)

        cv2.imshow("Mask", mask)
        cv2.imshow("Roi", roi)

        key = cv2.waitKey(2)
        if key == 27:
            break

    cap.release()
    cv2.destroyAllWindows()

    # Final calculation and output after playback ends
    print("-" * 50)
    print("Video Analysis Summary")
    print("-" * 50)
    print(f"Total Left Turns: {left_turn_counter}")
    if total_seconds > 0:
        cars_per_minute = (left_turn_counter / total_seconds) * 60
        print(f"Total Video Duration: {total_seconds:.2f} seconds")
        print(f"Left Turns Per Minute: {cars_per_minute:.2f}")
    else:
        print("Video duration is zero, cannot calculate turns per minute.")
    print("-" * 50)

    # Create a blank image to display the summary in a new window
    summary_img = np.zeros((200, 500, 3), dtype=np.uint8)
    cv2.putText(summary_img, "Video Analysis Summary", (20, 30),
cv2.FONT_HERSHEY_SIMPLEX, 0.7, (255, 255, 255), 2)
    cv2.putText(summary_img, f"Total Left Turns: {left_turn_counter}", (20,
70), cv2.FONT_HERSHEY_SIMPLEX, 0.6, (255, 255, 255), 1)
    if total_seconds > 0:
        cv2.putText(summary_img, f"Total Video Duration: {total_seconds:.2f}
s", (20, 110), cv2.FONT_HERSHEY_SIMPLEX, 0.6, (255, 255, 255), 1)
        cv2.putText(summary_img, f"Left Turns Per Minute:
{cars_per_minute:.2f}", (20, 150), cv2.FONT_HERSHEY_SIMPLEX, 0.6, (255, 255,
255), 1)
    else:
        cv2.putText(summary_img, "Cannot calculate turns per minute.", (20,
110), cv2.FONT_HERSHEY_SIMPLEX, 0.6, (255, 255, 255), 1)

    # Display the summary window
```

```python
    cv2.imshow("Analysis Summary", summary_img)
    key = cv2.waitKey(0)
    if key == 27:
        exit()

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

Exercise 2

```python
import numpy as np
import wave
from bitarray import bitarray
from bitarray.util import int2ba, ba2int
import json
import os # For path manipulation and checking file existence
import time # For time tracking
from multiprocessing import Pool, cpu_count
from scipy.io import wavfile
# --- Core Rice Coding Functions ---

def write_residuals(bit_array: bitarray, filename: str) -> None:
    # Create a copy to avoid modifying the original bitarray
    padded_bit_array = bit_array.copy()

    # Calculate padding needed to make length a multiple of 8
    padding_count = (8 - len(padded_bit_array) % 8) % 8  # Value will be 0
to 7

    # Extend the bitarray with '0's for padding
    padded_bit_array.extend('0' * padding_count)

    # Convert the padded bitarray to a bytes object
    data_bytes = padded_bit_array.tobytes()

    # Prepend the padding count as the first byte
    output_bytes = bytes([padding_count]) + data_bytes

    with open(filename, 'wb') as f:
        f.write(output_bytes)

    print(f"Wrote {len(bit_array)} to '{filename}'.")
```

```python
def read_residuals(filename: str) -> bitarray:
    with open(filename, 'rb') as f:
        data = f.read()

    if not data:
        return bitarray() # Return empty bitarray for an empty file

    # The first byte is the padding count
    padding_count = data[0]

    # The rest of the bytes are the encoded data
    bitstream_bytes = data[1:]

    # Convert the bytes back to a bitarray
    read_bit_array = bitarray()
    read_bit_array.frombytes(bitstream_bytes)

    # Slice the bitarray to remove the padding bits
    original_length = len(read_bit_array) - padding_count
    unpadded_bit_array = read_bit_array[:original_length]

    print(f"Read {len(unpadded_bit_array)} original bits from
'{filename}'.")

    return unpadded_bit_array
def encode_high_order_predictor(signal):
    # The predictor order is now a fixed value
    predictor_order = 4

    if len(signal) <= predictor_order:
        return signal # Return original signal if too short for prediction

    residuals = np.zeros_like(signal, dtype=signal.dtype)

    # The initial 4 samples are their own "residuals" (or unpredicted
values)
    residuals[:predictor_order] = signal[:predictor_order]

    # Apply the hardcoded 4th-order predictor
    for n in range(predictor_order, len(signal)):
        # Calculate the predicted sample using the 4th-order formula
        predicted_sample = (4 * signal[n-1] - 6 * signal[n-2] + 4 *
signal[n-3] - signal[n-4])
        residuals[n] = signal[n] - predicted_sample

    return residuals
```

```
def decode_high_order_predictor(residuals, dtype: np.dtype):
    """
    Reconstructs the original signal from residuals using a 4th-order linear
predictor.
    """
    predictor_order = 4

    if len(residuals) <= predictor_order:
        return residuals # If too short, residuals are the signal itself

    reconstructed_signal = np.zeros_like(residuals, dtype=dtype)

    # The initial 'predictor_order' samples are directly the residuals
    reconstructed_signal[:predictor_order] = residuals[:predictor_order]

    # Reconstruct the signal using the inverse of the 4th-order predictor
    for n in range(predictor_order, len(residuals)):
        predicted_sample_reconstructed = (
            4 * reconstructed_signal[n-1]
            - 6 * reconstructed_signal[n-2]
            + 4 * reconstructed_signal[n-3]
            - reconstructed_signal[n-4]
        )
        reconstructed_signal[n] = residuals[n] +
predicted_sample_reconstructed

    return reconstructed_signal


def rice_encode(samples: np.ndarray, k: int) -> bitarray:
    """
    Rice encodes an array of integers 'samples' (residuals) with parameter
'k'.
    Handles signed integers by first 'folding' them to a non-negative
    representation. Processes samples sequentially without multiprocessing.

    Args:
        samples (list[int] | np.ndarray): The array or list of integer
residuals to encode.
                                          Can contain positive or negative
values.
        k (int): The Rice parameter (non-negative integer).

    Returns:
        bitarray: The combined Rice code for all input samples as a single
```

```
bitarray.
    """
    num_samples = len(samples)
    final_encoded_bitstream = bitarray()

    print(f"Encoding {num_samples} samples sequentially...")

    # Ensure samples are standard Python integers for consistent to_bytes()
behavior
    # This also helps with the bitarray operations
    # if isinstance(samples, np.ndarray):
        # samples = samples.tolist()

    for i, sample in enumerate(samples):
        # fold to remove signed integers
        unsigned_sample = (sample << 1) ^ (sample >> 31)

        # 2. Rice encode the unsigned sample
        q = unsigned_sample >> k
        r = unsigned_sample & ((2 ** k) - 1)

        single_sample_bitarray = bitarray()

        # Unary part (q ones followed by a zero)
        # Use extend() with a string of '1's
        single_sample_bitarray.extend('1' * q)
        single_sample_bitarray.append(0) # Append a single 0 bit

        for bit_pos in range(k - 1, -1, -1): # From MSB to LSB
            single_sample_bitarray.append((r >> bit_pos) & 1)

        final_encoded_bitstream.extend(single_sample_bitarray)

        # Optional: Basic progress indicator
        if (i) % (num_samples // 100000) == 0:
            print(f"Encoding Progress: {((i) / num_samples) * 100:.2f}% ({i
+ 1}/{num_samples} samples)", end='\r')

    return final_encoded_bitstream
def rice_decode(bit_stream: bitarray, k: int) -> list[int]:
    """
    Decodes all Rice-encoded numbers from a bitarray.
    Returns a list of decoded samples.
    Optimized for performance by using bitarray.index() for unary decoding.
    """
    decoded_samples = []
```

```
    current_bit_offset = 0
    len_bit_stream = len(bit_stream) # Store length to avoid repeated calls

    # Pre-calculate 2^k as it's used repeatedly
    two_pow_k = 1 << k

    while current_bit_offset < len_bit_stream:


        try:
            # Find the '0' terminator for the unary part
            # This directly gives us the end of the unary sequence (and thus
the quotient)
            zero_terminator_index = bit_stream.index(False,
current_bit_offset)
        except ValueError:
            # If no '0' is found, it means the stream is malformed or ends
with '1's.
            # If we are at the very beginning and no '0' (meaning empty
stream), break.
            if current_bit_offset == len_bit_stream:
                break # End of stream, no more numbers
            else:
                raise ValueError(f"Malformed Rice code: No '0' terminator
found after bit offset {current_bit_offset}. Stream ended prematurely or
malformed.")

        # Quotient is the number of '1's before the '0'
        quotient = zero_terminator_index - current_bit_offset
        current_bit_offset = zero_terminator_index + 1 # Move past the '0'
terminator

        # Extract R (remainder) - Binary part of k bits
        remainder_start = current_bit_offset
        remainder_end = current_bit_offset + k

        if remainder_end > len_bit_stream:
            raise ValueError(f"Malformed Rice code: Not enough bits for
remainder (expected {k}, available {len_bit_stream - current_bit_offset}) at
bit offset {remainder_start}.")

        # Convert remainder bits to integer
        remainder = ba2int(bit_stream[remainder_start:remainder_end])
        current_bit_offset = remainder_end # Update offset to after
remainder
```

```
        # Reconstruct folded value: s_folded = Q * 2^k + R
        s_folded = (quotient * two_pow_k) + remainder

        # Unfold the value back to signed integer (signed Golomb-Rice 
coding)
        if s_folded & 1 == 0:  # Check if even using bitwise AND
            s_unfolded = s_folded >> 1
        else:
            s_unfolded = -((s_folded + 1) >> 1)

        decoded_samples.append(s_unfolded)

        if current_bit_offset % 100000 == 0:
            percentage = (current_bit_offset / len_bit_stream) * 100
            print(f"Decoding Progress: {percentage:.2f}%", end='\r')

    return decoded_samples

if __name__ == "__main__":
    base_filename = "Sound2"
    input_wav_path = f"{base_filename}.wav"
    encoded_path = f"{base_filename}_Enc.ex2"

    K = 2

    sr, source_audio_data  = wavfile.read(input_wav_path)

    residuals = encode_high_order_predictor(source_audio_data)
    print(f"{base_filename} Residuals: {residuals}")
    residuals = rice_encode(residuals, K)
    write_residuals(residuals, encoded_path)


    residuals = read_residuals(encoded_path)

    decoded_residuals = rice_decode(residuals, K)
    reconstructed_signal = decode_high_order_predictor(decoded_residuals, 
source_audio_data.dtype)


    roundtrip_path = f"{base_filename}_Enc_Dec.wav"
    wavfile.write(roundtrip_path, sr, reconstructed_signal)

    source_size = os.path.getsize(input_wav_path)
    encoded_size = os.path.getsize(encoded_path)
    print(f"source: {input_wav_path}: {source_size} bytes")
```

```
    print(f"encoded: {encoded_path}: {encoded_size} bytes")
    # difference = abs(source_size - roundtrip_size)
    pdiff = (encoded_size / source_size) * 100
    print(f"%Compression: {pdiff}")





    sr, roundtrip_audio_data  = wavfile.read(roundtrip_path)
    assert np.array_equal(source_audio_data, roundtrip_audio_data)
```

Exercise 3

```
import ffmpeg
import subprocess
import json
import os
from pathlib import Path

def encode_video(input_path: Path, params):
    args = dict(
        vcodec=params["video_codec"],
        acodec=params["audio_codec"],
        r=params["frame_rate"],
        aspect=params['aspect_ratio'],
        # ffmpeg-python expects 'k' suffix for kilobits
        video_bitrate = f"{params['video_bit_rate']['max']}k",
        audio_bitrate = f"{params['audio_bit_rate']['max']}k",
        ac=params['audio_channels']

    )
    stream = ffmpeg.input(input_path)

    res_parts = str(params['resolution']).split('x')
    width = res_parts[0].strip()
    height = res_parts[1].strip()

    stream = stream.video.filter('scale', width=width, height=height)

    combined_stream = ffmpeg.concat(stream, ffmpeg.input(input_path).audio,
v=1, a=1)

    output_path = input_path.parent / Path(str(input_path.stem) +
"_formatOK.mp4")
```

```
    (
        combined_stream
        .output(str(output_path), **args)
        .run(overwrite_output=True)
    )
    print(f"Video encoded successfully to: {output_path}")

def parse_aspect(width, height):
    def gcd(a, b):
        while b:
            a, b = b, a % b
        return a
    common_divisor = gcd(width, height)
    return f"{width // common_divisor}:{height // common_divisor}"

def parse_frame_rate(frame_rate):
    if '/' in frame_rate:
        num, den = map(int, frame_rate.split('/'))
        return round(num / den, 2)

def metadata(video_path):
    if not os.path.exists(video_path):
        print(f"Error: Video file not found at '{video_path}'")
        return None

    command = [
        'ffprobe',
        '-v', 'quiet',
        '-print_format', 'json',
        '-show_format',
        '-show_streams',
        video_path
    ]

    result = subprocess.run(command, capture_output=True, text=True,
check=True)

    # Parse the JSON output
    metadata = json.loads(result.stdout)

    # Extract format information
    format_info = metadata.get('format', {})
    audio = next((s for s in metadata.get("streams") if s.get("codec_type")
== "audio"))
    video = next((s for s in metadata.get("streams") if s.get("codec_type")
```

```python
== "video"))
    return dict(
            video_path = video_path,
            video_format = format_info.get('format_name'),
            video_codec = video["codec_name"],
            audio_codec = audio["codec_name"],
            frame_rate = parse_frame_rate(video["avg_frame_rate"]),
            aspect_ratio = parse_aspect(video["width"], video["height"]),
            resolution = f"{video['width']} x {video['height']}",
            video_bit_rate_mbps = int(video['bit_rate'])  / 1_000_000,
            video_bit_rate = int(video["bit_rate"]),
            audio_bit_rate = int(audio["bit_rate"]),
            audio_channels = audio.get('channels')
    )
if __name__ == "__main__":

    paths = [
        "Cosmos_War_of_the_Planets.mp4",
        "Last_man_on_earth_1964.mov",
        "The_Gun_and_the_Pulpit.avi",
        "The_Hill_Gang_Rides_Again.mp4",
        "Voyage_to_the_Planet_of_Prehistoric_Women.mp4"
    ]

    expected = dict(
        video_codec = "h264",
        audio_codec = "aac",
        frame_rate = 25,
        aspect_ratio = "16:9",
        resolution="640 x 360",
        video_bit_rate=dict(
            min=2_000_000,
            max=5_000_000,
        ),
        audio_bit_rate=dict(
            max=256000
        ),
        audio_channels=2
    )

    info = []

    for path in paths:
        path = Path(f"video/{path}")
        meta = metadata(path)
        requirements = dict(
```

```
            video_codec=meta["video_codec"] == expected['video_codec'],
            audio_codec=meta["audio_codec"] == expected["audio_codec"],
            frame_rate=meta["frame_rate"] == expected["frame_rate"],
            aspect_ratio=meta["aspect_ratio"] == expected["aspect_ratio"],
            resolution=meta["resolution"] == expected["resolution"],
            # ffprobe outputs bit rates in bits per second
            # 1 Mb/s = 1_000_000 bits per second
            # 1 kb/s = 1000 bits per second
            video_bit_rate=expected['video_bit_rate']['min'] <=
meta["video_bit_rate"] <= expected['video_bit_rate']['max'],
            audio_bit_rate=meta["audio_bit_rate"] <=
expected['audio_bit_rate']['max'],
            audio_channels=meta["audio_channels"] ==
expected['audio_channels']
        )

        report = str(meta['video_path']) +  "\n"
        for key, req in requirements.items():
            report += f"  · {key}: {meta[key]}"
            if req:
                report += " OK"
            else:
                if key == "video_bit_rate":
                    report += f" FAIL, {expected[key]['min']} >=
video_bit_rate < {expected[key]['max']}"
                elif key == "audio_bit_rate":
                    report += f" FAIL, < {expected[key]['max']}"
                else:
                    report += f" FAIL, {expected[key]}"

            report += "\n"

        info.append((meta, requirements, report))



    print(f"-------------- REPORTS ---------------")
    for _, _, report in info:
        print(report)
    print(f"-------------- END REPORT ---------------")

    for meta, req, _ in info:
        print(f"-------------- Processing Video {str(meta['video_path'])} --
-------------")
        if any(not r for r in req.values()):
```

```
                encode_video(meta["video_path"], expected)
```