**People counter using Computer Vision**

**Task Report**

***Submitted by:***

**Harsh Singh**

***To***

**Tango IT solutions Private Limited**

**1. Introduction**

In the people counting using computer we used the object tracking algorithm.

Difference between object tracking and object detection algorithm:-

When we apply object detection we are determining where in an image/frame an object is. An object detector is also typically more computionally expensive, and therefore slower, than an object tracking algorithm. Examples of object detection algorithms include Haar cascades, HOG+Linear SVM, and deep learning-based object detectors such as Faster R-CNNs, YOLO, and Single Shot Detectors (SSDs).

An Object tracker, on the other hand, will accept the input (x,y) –coordinates of where an object is in an image and will:

* Assign a unique ID to that particular object.
* Track the object as it moves around a video stream, predicting the new object location in the next frame(gradient, optical flow, etc.)

Examples of object traking algorithms include MedianFlow, MOSSE, GOTRUN, kernalized correlation filters, and discriminative correlation filters, to name a few.

Combining both object detection and object tracking:-

* Phase 1- Detection:- During the detection phase we are running our computionally more expensive object tracker to (1) detect if new objects have entered our view, and (2) see if we can find objects that were “lost” during the tracking phase. For each detected objects that were “lost” during the tracking phase. For each detected object we create or update an object tracker with the new bounding box coordinates. Since our object detector is more computationally expensive we only run this phase once every N frames.
* Phase 2 – Tracking: When we are not in the “detecting” phase we are in the “tracking” phase. For each of our detected objects, we create an object tracker to track the object as it moves around the frame. Our object tracker should be faster and more efficient than the object detector. We’ll continue tracking until we’ve reached the N-th frame and then re-run our object detector. The entire process then repeats.

Benefits:- We can apply highly accurate object detection methods without as much of the computational burden.

**2. Required Libraries**

A) dlib:- To install dlib follow these commands.

1. Open Anaconda Prompt in Administrative mode.
2. *conda update conda*
3. *conda update anaconda*
4. *conda create -n OpenCV\_env python=3.7*
5. *conda activate OpenCV\_env*
6. *conda install -c conda-forge dlib*

B) NumPy:- Type *conda install numpy* in command line with active OpenCV env to install numpy.

C) OpenCV:- Type *conda install opencv-python* in command line with active OpenCV env to install opencv.

D) imutils: - Type *conda install imutils* in command line with active OpenCV env to install imutils.

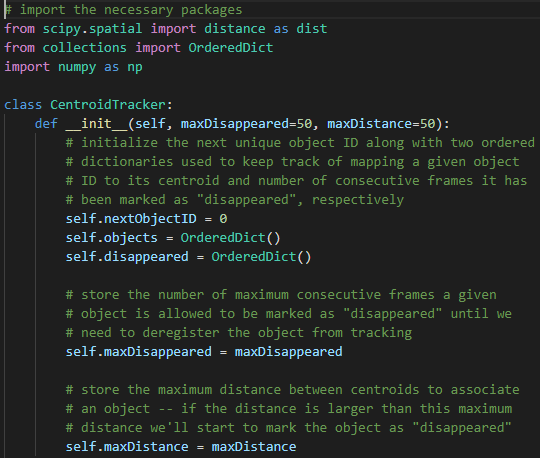
E) scedhule:- Type *conda install secdhule* in command line with active OpenCV env to install opencv.

**3. Required modules**

3.1 myUtils

A) centroidtracker.py

1. Initialization



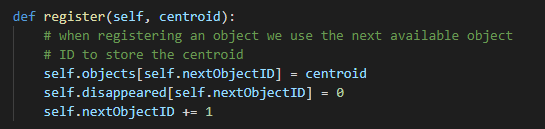
First, we import our required packages and modules –distance, OrderedDict, and numpy.

Our CentroidTracker class is defined, The constructor accepts a single parameter, the maximum number of consecutive frames as given object has to be lost/disappeared for until we remove it from our tracker.

Our constructor builds four class variables:

* nextObjectID: A counter used to assign unique IDs to each object. In the case that an object leaves the frame and doesn’t come back for maxDisappeared frames, a new (next) object ID would be assigned.
* objects: A dictionary that utilizes the object ID has the key and the centroid (x,y)- coordinates as the value.
* disappeared: Maintains number of consecutives frames (value) a particular objectID (key) has been marked as “lost”.
* maxDisappeared: The number of consecutive frames an object is allowed to be marked as “lost/disappeared” until we deregister the object.

1. Register Method

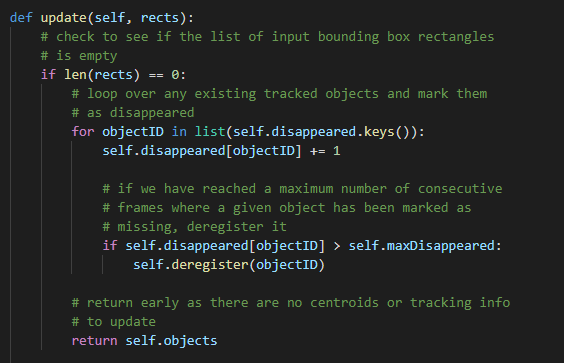


* The register method is defined, It accepts a centroid and then adds it to the objects dictionary using the next available object ID.
* The number of times an object has disappeared is initialized to 0 in the disappeared dictionary.
* Finally, we increment the nextObjectID, so that if a new object comes into view, it will be associated with a unique ID.

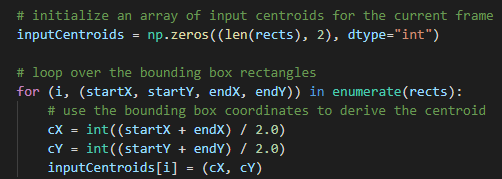
1. Deregister method

* Just like we can add new objects to our tracker, we also need the ability to remove old ones that have been lost or disappeared from our the input frames themselves.
* The deregister method is defined, It simply deletes the objectID in both the objects and disappeared dictionaries, respectively.

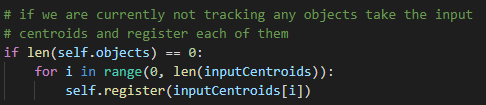
1. Update method

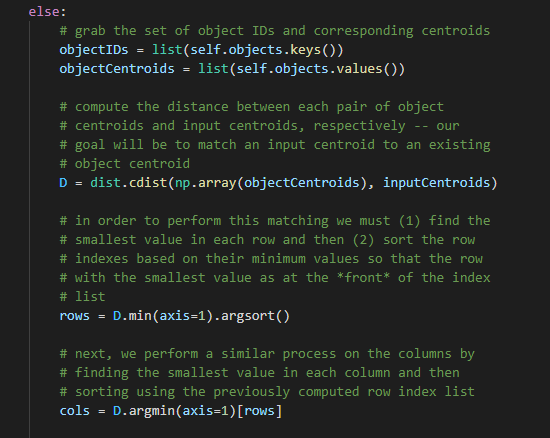


* Update method is heart of the centroid tracker.
* The update method accepts a list of bounding box rectangles, presumably from an object detector (Haar cascade, HOG + Linear SVM, Faster R-CNN, etc.) The format of the rects parameter is assumed to be a tuple with this structure: (startX, startY, endX, endY).
* If there are no detections, we’ll loop over all object IDs and increment their disappeared count. We’ll also check if we have reached the maximum number of consecutive frames a given object has been marked as missing. If that is the case we need to remove it from our tracking systems. Since there is no tracking info to update, we go ahead and return early on.

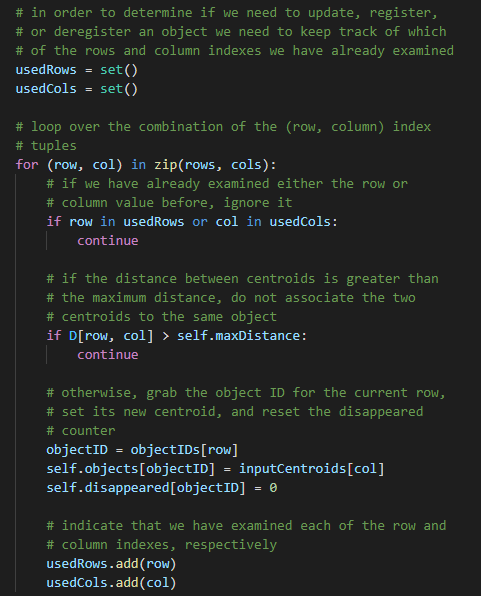


* We’ll initialize a NumPy array to strore the centroids for each rect.
* Then, we loop over bounding box rectangles and compute the centroid and store it in the inputCentroids list.
* If there are currently no objects we are tracking, we’ll register each of the new objects:

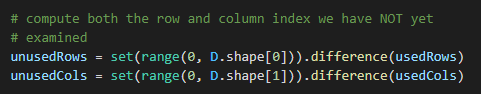




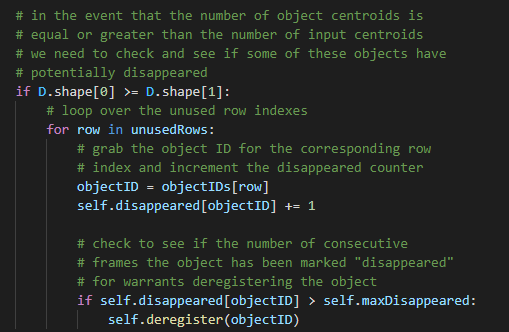
* The update to existing tracked objects take place beginning at he else. The goal is to track the objects and to maintain correct object IDs – this process is accomplished by computing the Euclidean distances between all pairs of objectCentroids and inputCentroids, followed by associating object IDs that minimize the Euclidean distance.
* Inside of the else block beginning, we will:
  + Grab, objectIDs and objectCentroid values.
  + Compute the distance between each pair of existing object centroids and new input centroids. The output NumPy array shape of our distance map will be (# of object centroids, # of input centroids).
  + To perform the matching we must (1) Find the smallest value in each row, and (2) Sort the row indexes based on the minmum values. We perform a very similar process on the columns, finding the smallest value in each column, and then sorting them based on the ordered rows. Our goal is to have the index with the smallest corresponding distance at the front of the list.
* The next step is to use the distances to see if we can associate object IDs:



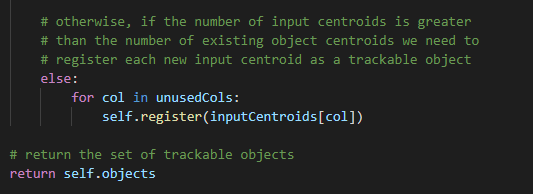
* Intialize two sets to determine which row and column indexes we have already used. Keep in mind that a set is similar to a list but it contains only unique values.
* Then we loop over the combinations of (row, col) index tuples in order to update our object centroids:
* If we’ve already used either this row or column index, ignore it and continue to loop.
* Otherwise, we have found an input centroid that:
* 1. Has the smallest Euclidean distance to an existing centroid
* 2. And has not been matched with any other object.
* In that case, we update the object centroid and make sure to add the row and col to their respective usedRows and usedCols sets.
* They are likely indexes in our usedRows + usedCols sets that we have NOT examined yet:



* So we must determine which centroid indexes we haven’t examined yet and store them in two new convenient sets (unusedRows and unusedCols).
* Our final check handles any objects that have become lost or if they’ve potentially disappeared:

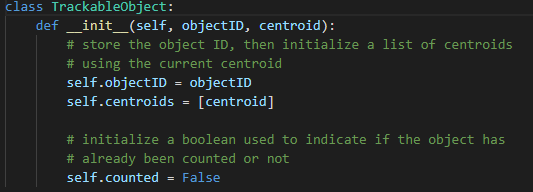


* If the number of object centroids is greater than or equal to the number of input centroids.
* We need to verify if any of these objects are lost or have disappeared by looping over unused row indexes if any.
* In the loop we’ll:
* 1. Increment their disappeared count in the dictionary.
* 2. Check if the disappeared count exceeds the maxDisappeared threshold, and, if so we’ll deregister the object.



* We loop over the unusedCols indexes and we register each new centroid. Finally, we’ll return the set of trackable objects to the calling method.

B) trackableobject.py

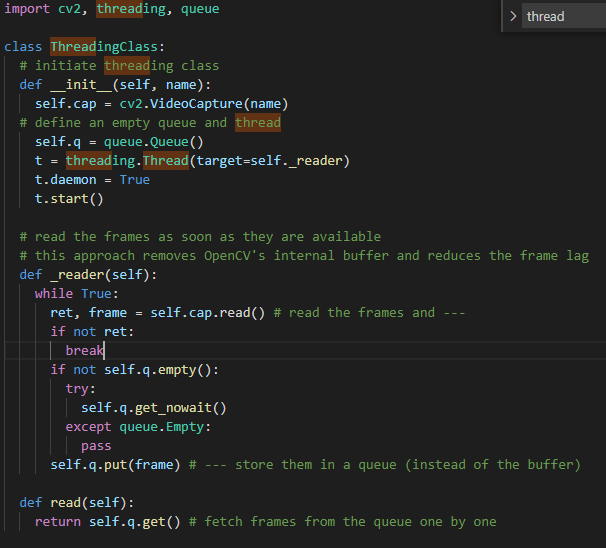


* It is a class named TrackableObject, which store the object ID, then initialize a list of centroids using the current centroid.
* Intialize a Boolean used to indicate if the object has already been counted or not.

C) mailer.py (optional)

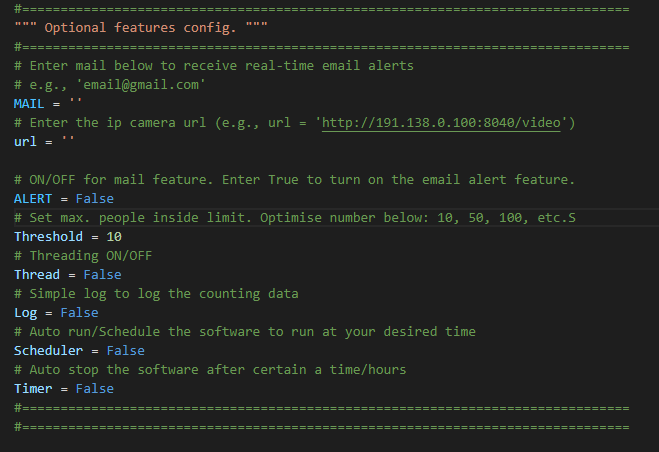
It is a optional class this class is used for generating Alert when no. of people exceeded inside your home or elsewhere.

D) thread.py



This class is used for removing OpenCV's internal buffer and reduces the frame lag.

E) config.py (Optional)



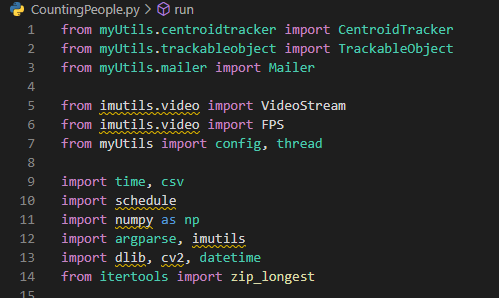
To connect Ip camera url for live people counting.

**4. Object detection model**

A) MobileNetSSD\_deploy.caffemodel

B) MobileNetSSD\_deploy.prototxt

**5. CountingPeople.py**

****

A) We begin by importing our necessary packages:-

1. From the **myUtils** module, we import the **CentroidTracker** ans **TrackabelObject** classes.

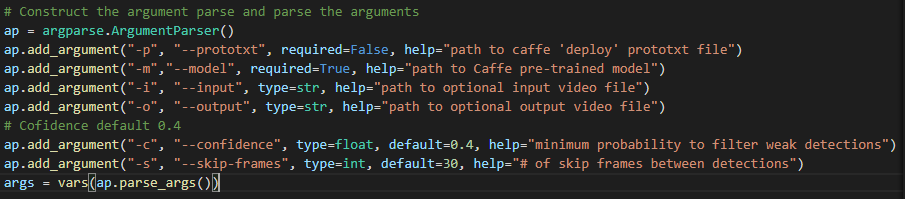
2. The **VideoStream** and FPS from **imutils.video** will help us to work with a webcam and to calculate the estimated Frames per second (FPS) throughout rate.

3. We need **imutils** for its **OpenCV** convenience functions.

4. The **dlib** library will be used for its correlation tracker implementation.

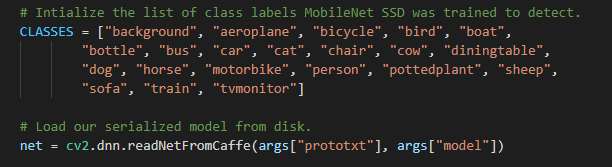
5. **OpenCV** will be used for deep neural network interference, importing object detection models, opening and writing video files, and displaying output frames to our screen.

B) Now, we have six command line arguments which will allow us to pass information to our people counter script from the terminal at runtime.



* --prototxt: path to the caffer deploy protoxt file.
* --model: path to the caffe pre-trained CNN model.
* --input: Optional input video file path. If no path is specified, your webcam will be utilized or camera which is connected to IP url.
* --output: Optonal output video path. If no path is specified, a video will not be recorded.
* --confidence: With a default value of 0.4, this is the minimum probability threshold which helps to fiter out weaks detections.
* --skip-frames: The number of frames to skip before running our DNN detector again on the tracked object. Remember, object detection is computionally expensive, but it does help our tracker to reassess objects in the frame. By default we skip 30 frames between detecting objects with the OpenCV DNN module and our CNN single shot detector model.

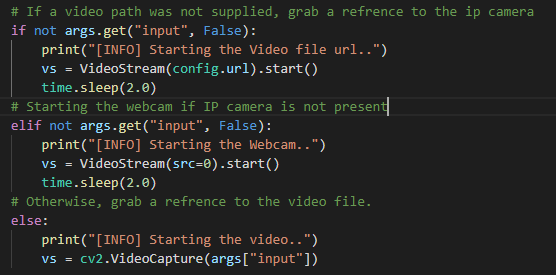
C) Single Shot Detector preparation (SSD)



First, we’ll initialize CLASSES –the list of classes that our SSD supports. The list should not be changed if you’re using the model provided in the “Downloads”. We’re only interested in the “person” class, but you could count other moving objects as well (however, if your “pottedplant”, “sofa”, or “tvmonitor” are moving via person. They will also get counted. You can remove the classes like car, bus, cow, boat, aeroplane, dog, motorbike, horse etc. those who can move by their own.

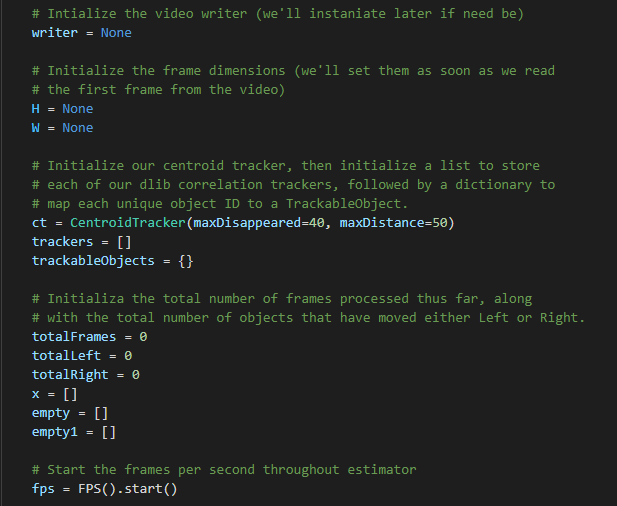
We load our pre-trained MobileNet SSD used to detect objects after declaring list of CLASSES.

D) Initializing our video steam.



First, we handle the case where we’re using a webcam video stream or installed camera which has IP url. Otherwise, we’ll be capturing frames from a video file.

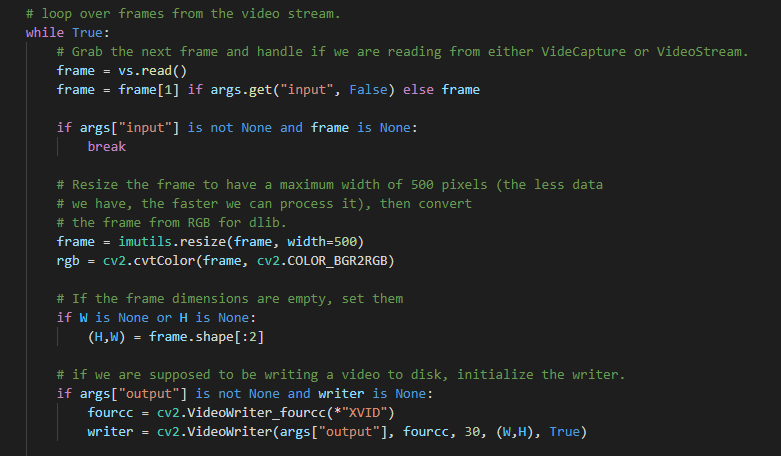
E) Some more initializations.



The remaining initialization include:

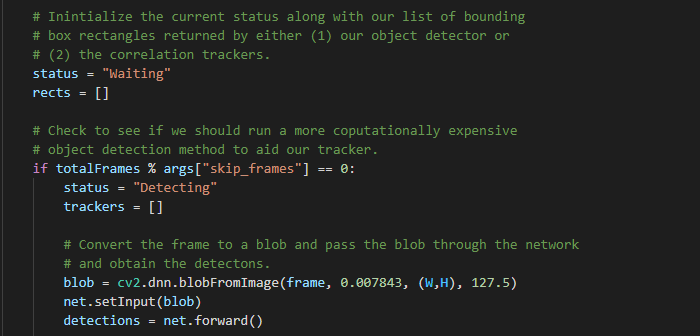
* Writer: Our video writer. We’ll instantiate this object later if we are writing to video.
* W and H: Our frame dimensions. We’ll need to plug these into cv2.VideoWriter.
* ct: Our CentroidTracker.
* trackers: A list to store the dib correlations trackers.
* trackableObjects: A dictionary which maps objectID to a TrackableObject.
* totalFrames: The total number of frames processed.
* totalRight and totalLeft: The total number of objects/people that have moved either down or jup. These variables measure the actual “people counting” results of the script.
* fps: Our frames per second estimator for benchmaking.

F) Loop over the frames from the video stream



* We begin looping, At the top of the loop we grab the next frame. In the event that we’ve reached the end of the video, we’ll break out of the loop.
* Preprocessing the frame takes place at frame = imutils.resize(frame, width=500). This include resizing and swaping color channels as dlib requires an ‘rgb’ image.
* We grab the dimensions of the frame for the video writer.
* From there we’ll instantiate the video writer if an output path was provided via command line argument. To learn more about writing video to disk.

G) Detecting people using Object Detector



We initialize a status as “Waiting”. Possible status include:

* Waiting: In this state, we’re waiting on people to be detected and tracked.
* Detecting: We’re actively in the process of detecting people using the MobileNetSSD.
* Tracking: People are being tracked in the frame and we’re counting the totalLeft and totalRight.

Our rects list will be populated either via detection or tracking. We go ahead and initialize rects.

**Note:** Deep Leaning Object detectors are very computationally expensive, especially if you are running them on your CPU.

To avoid running our object detector on every frame, and to speed up our tracking pipeline, we’ll be skipping every N frames (set by command line argument –skip-frames where 30 is the default). Only every N frames will we exercise our SSD for object detection. Otherwise, we’ll simply be tracking moving objects in-between.

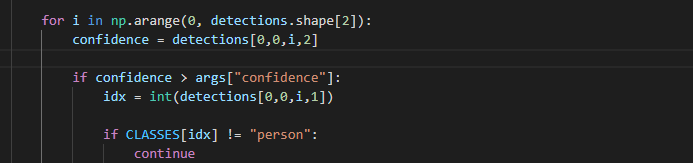
Using the module operator we ensure that we’ll only execute the code in the if-statement every N frames.

Assuming we’ve landed on a multiple of skip\_frames, we’ll update the status to “Detecting”.

Then we initialize our new list of trackers.

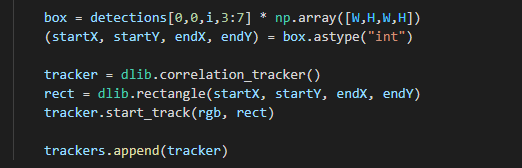
Next, we’ll perform inference via object detection. We begin by creating a blob from the image, followed by passing the blob through the net to obtain detections.

H) Now, we’ll loop over each of the detections to find objects belonging to the “person” class.



* Looping over detections, we proceed to grab the confidence and filter out weak results and those that don’t belong to the “person” class.

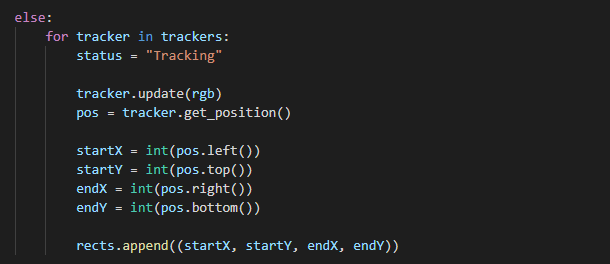
I) computing a bounding box for detection and tracking



* Computing our bounding box takes place.
* Then we instantiate our dlib correlation tracker, followed by passing in the object’s bounding box cordiantes to dlib.rectngle, storing the results as rect.
* Subsequently, we start tracking and append the tracker to the trackers list.

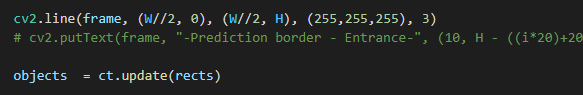
That’s wrap for all operations we do every N skip-frames!

J) Utilizing Object trackers rather than object detections



* Most of the time, we aren’t landing on a skip-frame multiple. During this time, we’ll utilize our trackers to track our object rather than applying detection.
* We begin looping over the available trackers.
* We proceed to update the status to “Tracking” and grab the object position.
* From there we extract the position coordinates followed by populating the information in our rects list.

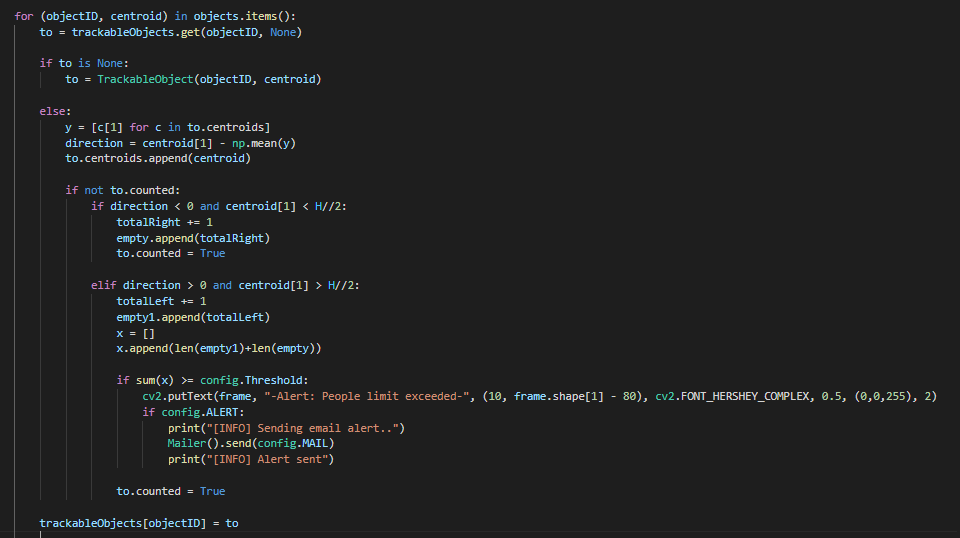
K) Drawing a vertical visualization line



We draw the vertical line which we’ll be using to visualize people “crossing” –once people cross this line we’ll increment our respective counters.

Then, we utilize our CentroidTracker instantiation to accept the list of rects, regardless of whether they were generated via object detection or object tracking. Our centroid tracker will associate object IDs with object locations.

L) Counting persons moving left or moving right.

* We begin by looping over the updated bounding box coordinates of the object IDs.
* We attempt to fetch a TrackableObject for the current objectID.
* If the TrackableObject doesn’t exist for the objectID, we create one.
* Otherwise, there is already an existing TrackableObject, so we need to figure out if the object (person) is moving left or right.
* To do so, we grab the y-cordiante value for all previous centroid locations for the given object. Then we compute the direction by taking the difference between the current centroid location and the mean of all previous centroid locations.

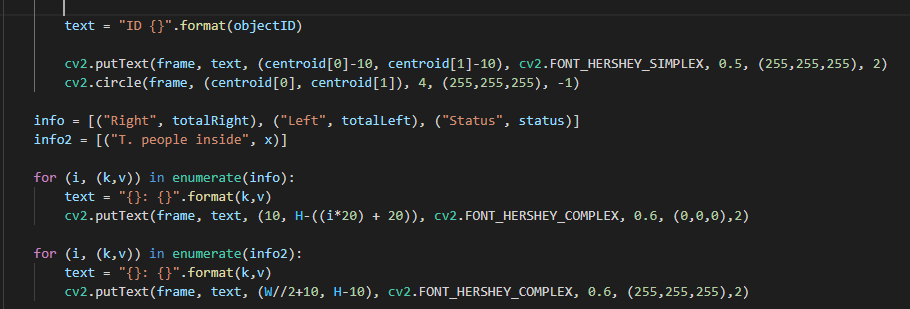
The reason we take the mean is to ensure our direction tracking is more stable. If we stored just the previous centroid location for the person we leave ourselves open to the possibility of false direction counting. Keep in mind that object detection and object tracking algorithms are not “magic” –sometimes they will predict bounding boxes that may be slightly off what you may expect; therefore, by taking the mean, we can make our people counter more accurate.

If the TrackableObject has not been counted, we need to determine if it’s ready to be counted yet by:

1. Checking if the direction is negative (indicating the object is moving **Left**) and the centroid is after the centreline. In this case we increment totalLeft.
2. Or checking if the direction is positive (indicating the object is moving **Right**) and the centroid is before the centreline. If this true, we increment totalDown.

Finally, we store the TrackableObject in our trackableObjects dictionary, so we can grab and update it when the next frame is captured.

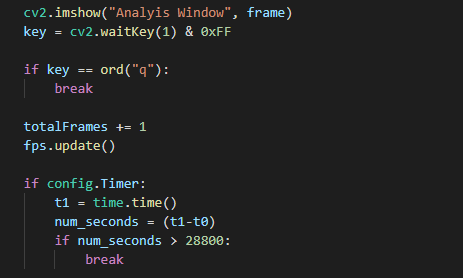
M) Written text on frame window



Here we overlay the following data on the frame:

* ObjectID: Each object’s numerical identifier.
* Centroid: the center of the object will be represented by a “dot” which is created by filling in a circle.
* Info: Includes totalLeft, totalRight and status.
* Info2: Represents the count of total no. people.

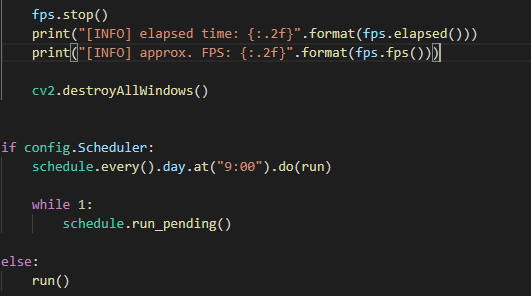
N) Output window



In this block we:

* Write the frame, if necessary, to the output video file.
* Display the frame and handle keypress. If “q” is pressed, we break out of the frame processing loop.
* Update our fps counter.

O) DestroyAllWindows



* We imported the Scheduler to schedule the timings of IP camera.
* fps.elapsed() gives the total elapsed time.
* fps.fps() gives the approx. FPS.

**6. Testing**

****

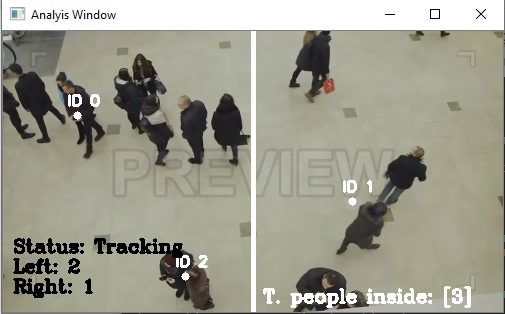
To run the people counter run the following commands:

* conda activate OpenCV env
* Change the path where countingPeople.py file is present.
* cd C:\Users\Harsh\Desktop\Projects\Computer Vision\OpenCV\Counting number of people using opencv
* Then type:-
* python CountingPeople.py --prototxt mobilenet\_ssd\MobileNetSSD\_deploy.prototxt --model mobilenet\_ssd\MobileNetSSD\_deploy.caffemodel --input videos\example.mp4 --output Output\output\_01.avi

Case1:- If you want to use webcam then don’t write –input argument.

Case2:- If you don’t want to save output file. Then leave blank –output argument.

**7. Output**



This is the screen shot of the video. Complete output video is present in the Output folder.