Assignment: A2

The Air-Track Framework

This assignment establishes the algorithmic framework (the Python classes) that will be used in the air-track and the air-table projects. This also introduces two basic calculation concepts:

- Time-based position calculations (our first dose of physics) and,
- The *meters* of the physics "World" and the *pixels* of the display screen, and converting back and forth between them.

(I'll have an air-track set up in the lab in case you've never had a chance to use one.)

Python language topics:

- Code blocks and indenting (a review).
- Functions: what they do and what they return.
- Classes:
    - Methods;
    - Properties;
    - Instantiation: creation and initialization of an instance (an object).
- Namespace and the "main" function.
- Floating-point and integer numbers:
    - Rounding of floating-point numbers.

Note: the links to Python tutorials are in A01_game_loop_and_events.pdf.

Problem statement:

(First, be sure and start with a new Python file! You can use (copy and paste) stuff from the first exercise, but don't edit in your original copy of the previous exercise. That should be your pattern for each assignment: start a new file.)

Write a program to animate two rectangular objects (cars) in a one-dimensional space like an air-track.
Have the following controls:

1. The "esc" key to quit;
2. The number keys, "1" and "2", should be used to start each of two demos (without restarting the program). The two demos should differ in the initial position, velocity, and color of each of the cars.

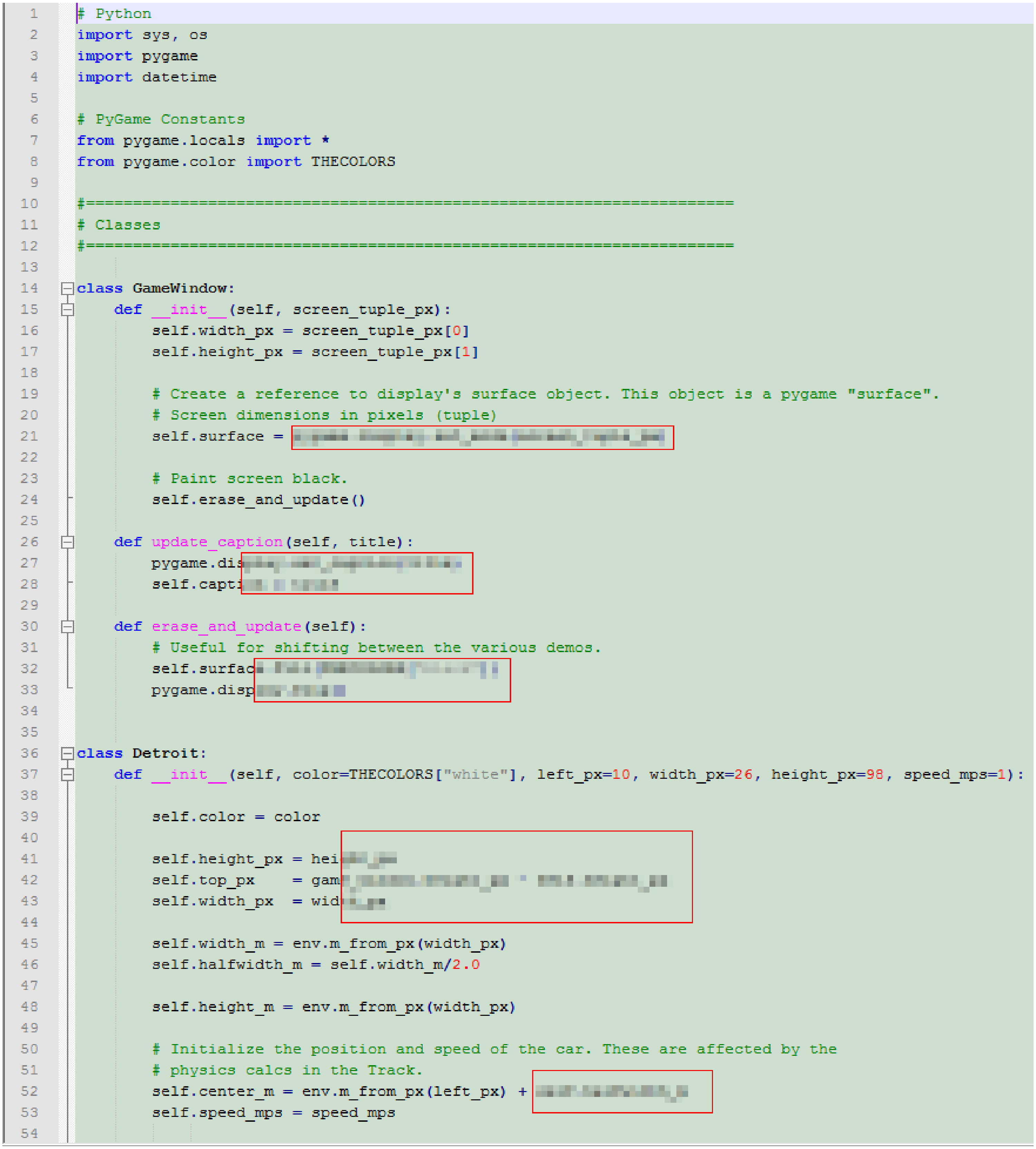
Algorithmic description:

- Import content from modules.
- Define classes and functions.
    - GameWindow
        - Attributes: dimensions of the screen, and the display surface object.
        - Methods:
            - Initialize;

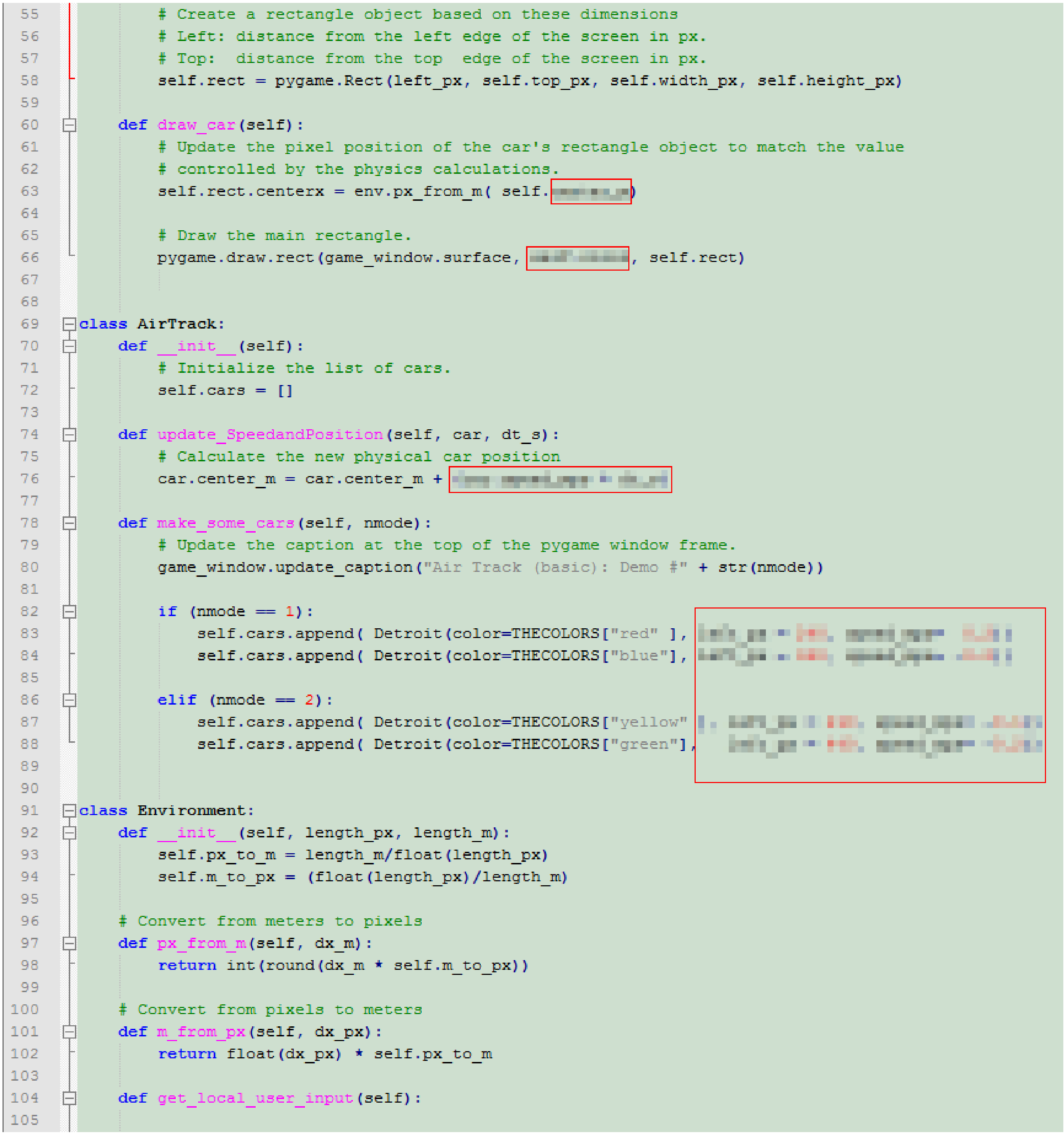
            - Set caption;
            - Erase screen.
    - Detroit
        - Attributes: car dimensions, position, speed, and rectangle object.
        - Methods:
            - Initialize;
            - Draw this car.
    - AirTrack
        - Attributes: the list of cars.
        - Methods:
            - Update car speed and position based on physics;
            - Make (instantiate) some cars based on demo mode.
    - Environment
        - Attributes: pixels-to-meters ratio, meters-to-pixels ratio.
        - Methods:
            - Pixels-to-meters conversion;
            - Meters-to-pixels conversion.

- Initialize the program
    - Initialize the first demo: build (instantiate) two cars, and initialize their position and velocity.
- The main game loop:
    - Erase the surface.
    - Establish the time step dt_s.
    - Check for user input: to quit, or to change demo mode.
    - Update the speed and x position of each car based on the time step for this frame.
        - position += velocity * dt_s
    - Draw each car at its new position
        - Convert from meters to pixels.
        - Then draw it.
    - Update the total time since starting (we don’t actually use this yet but will later).
    - Make this update visible on the screen.

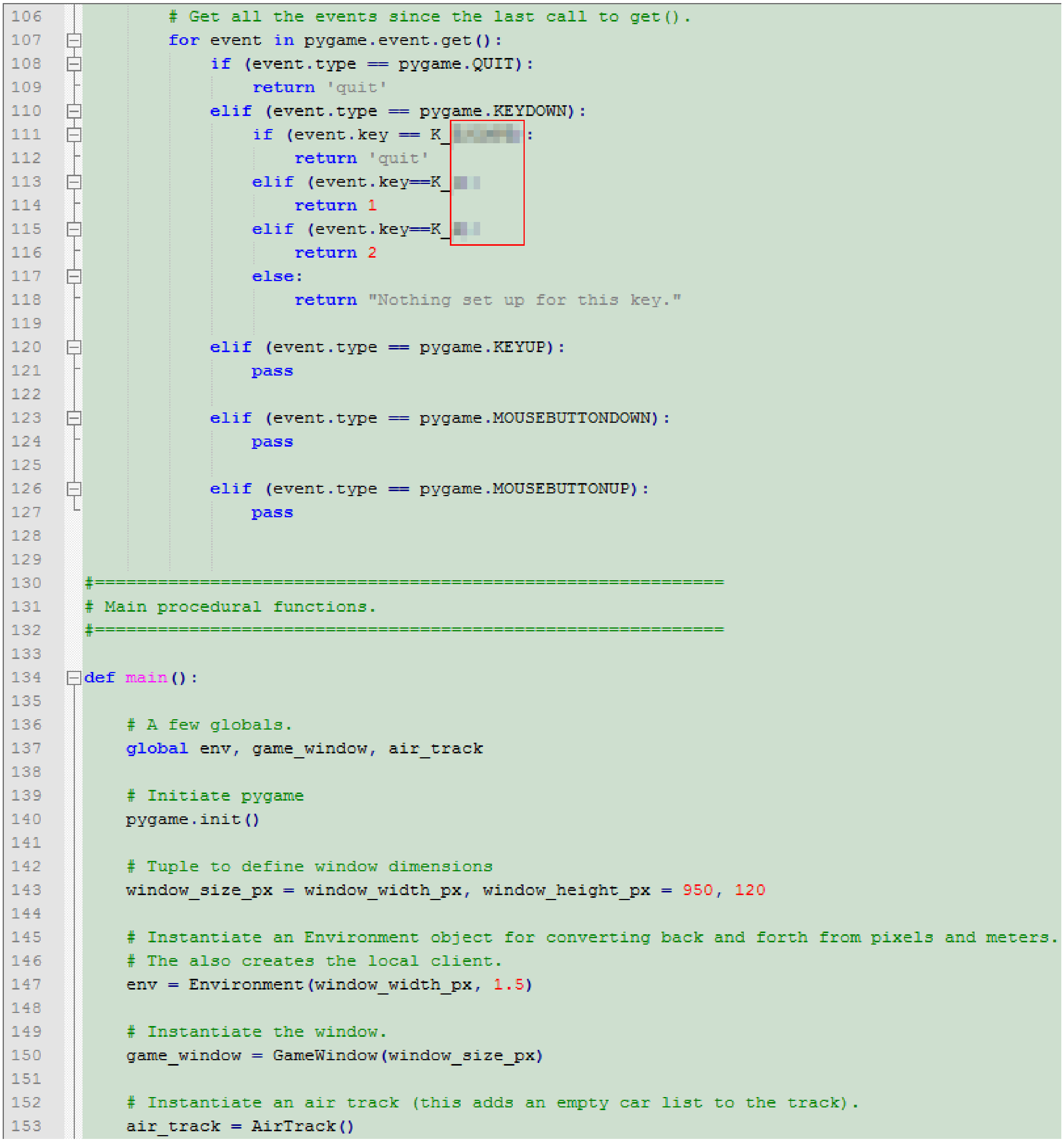
Python code: (see images on next few pages)
Here again, this code solution is provided (as an image) in the assignment statement. Some parts of the image have been obfuscated; you’ll have to fill in the blanks to get this to run. Later in the course, code solutions will be provided as text files a day or two after the assignment is given.



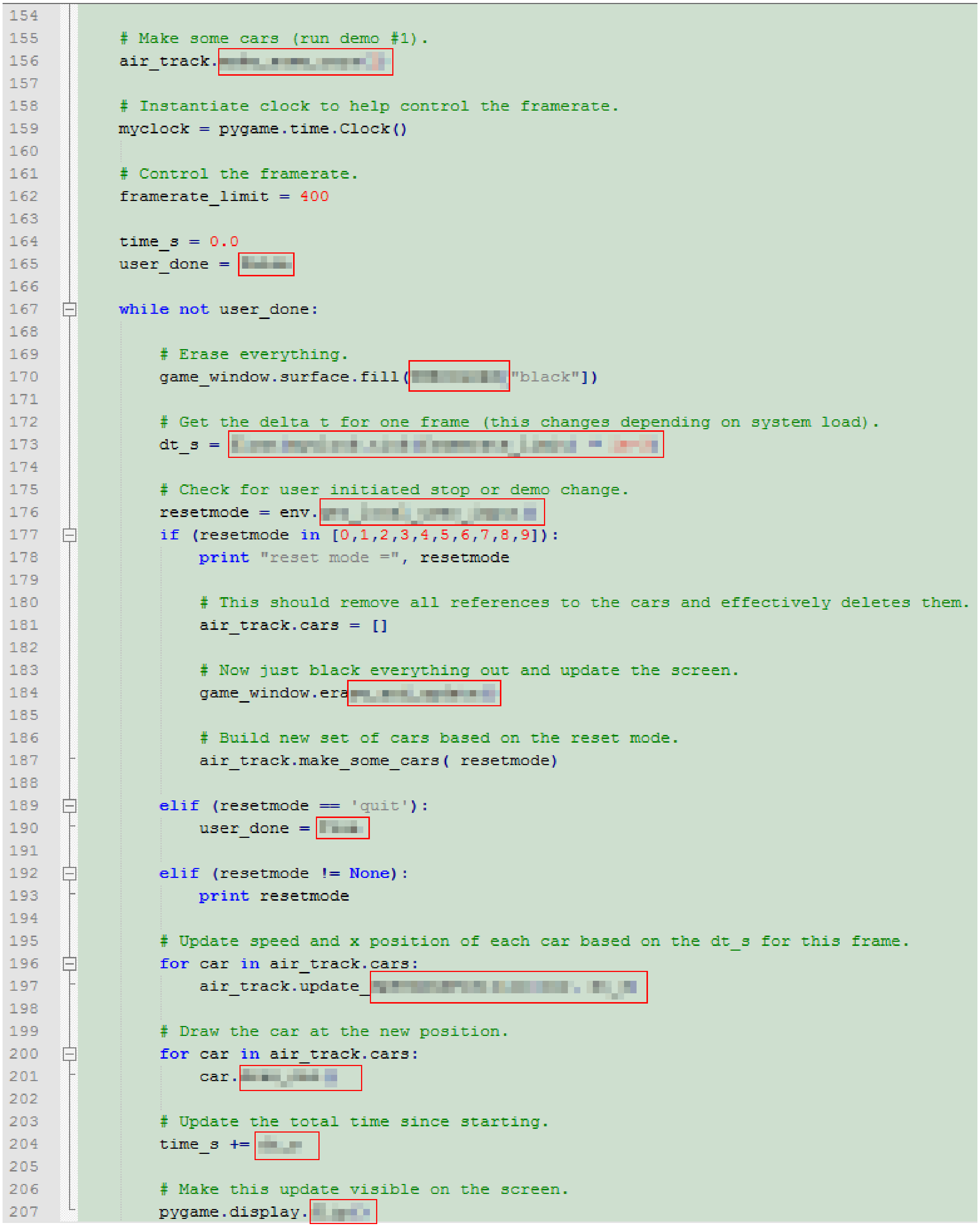
```python
# Python
import sys, os
import pygame
import datetime

# PyGame Constants
from pygame.locals import *
from pygame.color import THECOLORS

#===============================================================================
# Classes
#===============================================================================

class GameWindow:
    def __init__(self, screen_tuple_px):
        self.width_px = screen_tuple_px[0]
        self.height_px = screen_tuple_px[1]

        # Create a reference to display's surface object. This object is a pygame "surface".
        # Screen dimensions in pixels (tuple)
        self.surface = [redacted]

        # Paint screen black.
        self.erase_and_update()

    def update_caption(self, title):
        pygame.dis[redacted]
        self.capti

    def erase_and_update(self):
        # Useful for shifting between the various demos.
        self.surfac[redacted]
        pygame.disp

class Detroit:
    def __init__(self, color=THECOLORS["white"], left_px=10, width_px=26, height_px=98, speed_mps=1):

        self.color = color

        self.height_px = hei[redacted]
        self.top_px     = gam
        self.width_px  = wid

        self.width_m = env.m_from_px(width_px)
        self.halfwidth_m = self.width_m/2.0

        self.height_m = env.m_from_px(width_px)

        # Initialize the position and speed of the car. These are affected by the
        # physics calcs in the Track.
        self.center_m = env.m_from_px(left_px) + [redacted]
        self.speed_mps = speed_mps

```



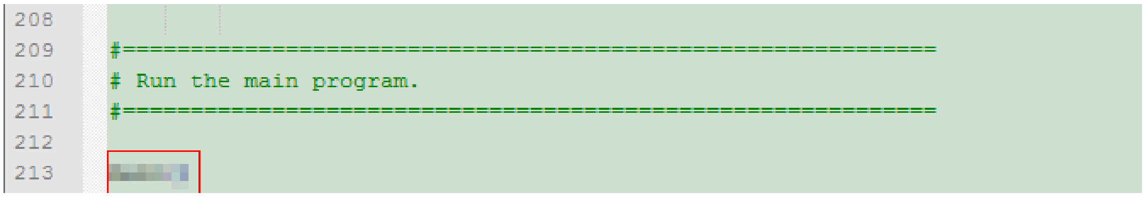
```
        # Create a rectangle object based on these dimensions
        # Left: distance from the left edge of the screen in px.
        # Top:  distance from the top  edge of the screen in px.
        self.rect = pygame.Rect(left_px, self.top_px, self.width_px, self.height_px)

    def draw_car(self):
        # Update the pixel position of the car's rectangle object to match the value
        # controlled by the physics calculations.
        self.rect.centerx = env.px_from_m( self.[redacted])

        # Draw the main rectangle.
        pygame.draw.rect(game_window.surface, [redacted], self.rect)


class AirTrack:
    def __init__(self):
        # Initialize the list of cars.
        self.cars = []

    def update_SpeedandPosition(self, car, dt_s):
        # Calculate the new physical car position
        car.center_m = car.center_m + [redacted]

    def make_some_cars(self, nmode):
        # Update the caption at the top of the pygame window frame.
        game_window.update_caption("Air Track (basic): Demo #" + str(nmode))

        if (nmode == 1):
            self.cars.append( Detroit(color=THECOLORS["red" ], [redacted]
            self.cars.append( Detroit(color=THECOLORS["blue"],

        elif (nmode == 2):
            self.cars.append( Detroit(color=THECOLORS["yellow" [redacted]
            self.cars.append( Detroit(color=THECOLORS["green"],


class Environment:
    def __init__(self, length_px, length_m):
        self.px_to_m = length_m/float(length_px)
        self.m_to_px = (float(length_px)/length_m)

    # Convert from meters to pixels
    def px_from_m(self, dx_m):
        return int(round(dx_m * self.m_to_px))

    # Convert from pixels to meters
    def m_from_px(self, dx_px):
        return float(dx_px) * self.px_to_m

    def get_local_user_input(self):
```



```
        # Get all the events since the last call to get().
        for event in pygame.event.get():
            if (event.type == pygame.QUIT):
                return 'quit'
            elif (event.type == pygame.KEYDOWN):
                if (event.key == K_[redacted]:
                    return 'quit'
                elif (event.key==K_[redacted]
                    return 1
                elif (event.key==K_[redacted]
                    return 2
                else:
                    return "Nothing set up for this key."

            elif (event.type == pygame.KEYUP):
                pass

            elif (event.type == pygame.MOUSEBUTTONDOWN):
                pass

            elif (event.type == pygame.MOUSEBUTTONUP):
                pass


#============================================================
# Main procedural functions.
#============================================================

def main():

    # A few globals.
    global env, game_window, air_track

    # Initiate pygame
    pygame.init()

    # Tuple to define window dimensions
    window_size_px = window_width_px, window_height_px = 950, 120

    # Instantiate an Environment object for converting back and forth from pixels and meters.
    # The also creates the local client.
    env = Environment(window_width_px, 1.5)

    # Instantiate the window.
    game_window = GameWindow(window_size_px)

    # Instantiate an air track (this adds an empty car list to the track).
    air_track = AirTrack()
```



```

    # Make some cars (run demo #1).
    air_track.[redacted]

    # Instantiate clock to help control the framerate.
    myclock = pygame.time.Clock()

    # Control the framerate.
    framerate_limit = 400

    time_s = 0.0
    user_done = [redacted]

    while not user_done:

        # Erase everything.
        game_window.surface.fill([redacted]"black"])

        # Get the delta t for one frame (this changes depending on system load).
        dt_s = [redacted]

        # Check for user initiated stop or demo change.
        resetmode = env.[redacted]
        if (resetmode in [0,1,2,3,4,5,6,7,8,9]):
            print "reset mode =", resetmode

            # This should remove all references to the cars and effectively deletes them.
            air_track.cars = []

            # Now just black everything out and update the screen.
            game_window.era[redacted]

            # Build new set of cars based on the reset mode.
            air_track.make_some_cars( resetmode)

        elif (resetmode == 'quit'):
            user_done = [redacted]

        elif (resetmode != None):
            print resetmode

        # Update speed and x position of each car based on the dt_s for this frame.
        for car in air_track.cars:
            air_track.update_[redacted]

        # Draw the car at the new position.
        for car in air_track.cars:
            car.[redacted]

        # Update the total time since starting.
        time_s += [redacted]

        # Make this update visible on the screen.
        pygame.display.[redacted]
```



```

#============================================================
# Run the main program.
#============================================================

[redacted]
```