**Q:** If I put "vision" or "ram" what will be the differences in the observation shape. What if I put "revised"?   
**Q:** Another example, what if on a frame there are 8 objects and on the next frame there are 9, then how do you specify the global observation shape size?   
**Q:** Is there a way to get the maximum number of objects that could appear in a frame?

“Dear client, if your goal is to train a DQN agent on images, we recommend to use the raw value for the mode argument of the ocatari class. Indeed, when doing so the resulting observation space of the initialized environment will be images, I.E, 3D arrays that are stacked or not. This is done in the source code with the functions ....”

##############################################################################  
In ***OC\_Atari\ocatari\core.py:***

The code checks if the torch module (PyTorch), cv2 (OpenCV) and pygame are installed. If any of these is not found, the code suggests installing it with pip install “missing module”.

AVAILABLE\_GAMES = ["Alien", "Assault", "Asterix", "Asteroids", "Atlantis", "BeamRider", "Berzerk", "Bowling", "Boxing",

                   "Breakout", "Carnival", "Centipede", "ChoppperCommand", "DemonAttack", "Enduro", "FishingDerby", "Freeway",

                   "Frostbite", "Gopher", "Hero", "IceHockey", "Kangaroo", "MontezumaRevenge", "MsPacman","Pitfall", "Pong", "PrivateEye",

                   "Qbert", "Riverraid", "RoadRunner", "Seaquest", "Skiing", "SpaceInvaders", "Tennis","Videocube", "Venture", "Yarsrevenge"]

1)Render mode = *human*:   
displays the game in front of you using pygame window, while limiting the fps to 60 to avoid super-fast movement

2)Render mode = *rgb\_array*:   
does not display it, just stores it as a 3d array.

**NB:** Ipdb= IPython Debugger

1)"Vision" Mode:

In "vision" mode, the environment is observed using vision-based sensors, typically capturing the visual information (e.g., pixels, screenshots) of the game screen.

This mode is suitable for tasks that require analyzing and making decisions based on the visual output of the game.

2)"Raw" Mode:

In "raw" mode, the environment is observed using raw data, such as reading the values of specific RAM addresses of the emulator.

This mode is suitable for tasks that do not require visual information and instead rely on specific game state variables stored in RAM.

3)"Revised" Mode:

In "revised" mode, the environment processing is adapted to a specific mode that includes some modifications. This mode is tailored to a particular game environment, and the modifications are likely to be specific to that game.

The detect\_objects\_revised function is used to detect and process objects based on the modified game state variables.

This mode is suitable for games where the default "vision" and "raw" modes do not provide the necessary information, and custom processing is required.

4)"Both" Mode:

In "both" mode, both "vision" and "revised" modes are available simultaneously. This means that the environment can be observed and interacted with using both visual information and modified game state variables.

Separate functions, detect\_objects\_vision and detect\_objects\_revised, are used to detect and process objects based on visual and modified game state data, respectively.

The \_step\_test method is used to interact with the environment and allows you to choose whether to use visual or modified game state information for interaction.

This mode is useful when you want to experiment with agents that utilize both **visual and revised** game state information.

The primary use case for the "both" mode is when you require access to both types of information, likely for debugging, analysis, or specific research purposes. You might want to observe visual information and access game state variables simultaneously to gain insights into how the game environment works or to facilitate specific types of analysis.

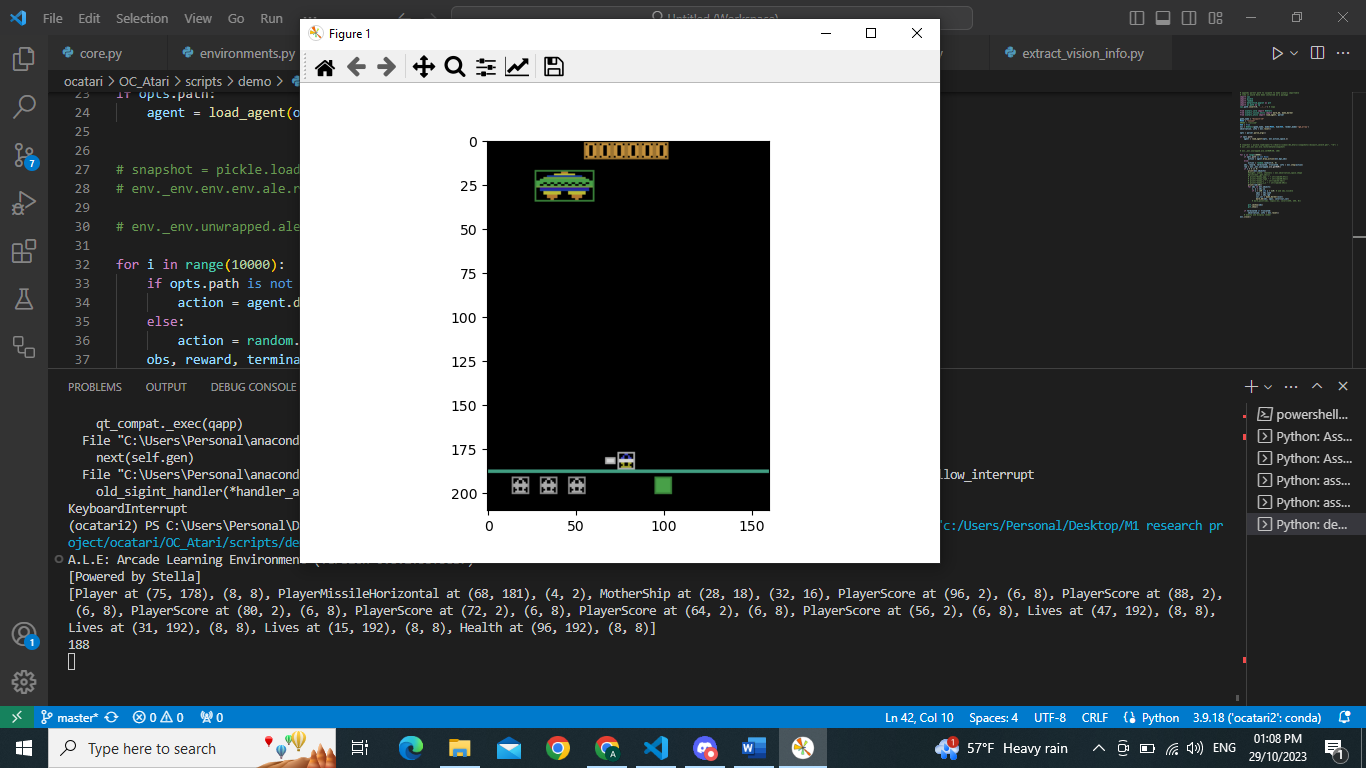
***1)USING “VISION”:***

As we can see here the initial observation was made when we had 13 objects on the screen:

* Player at (75, 178), (8, 8),
* PlayerMissileHorizontal at (68, 181), (4, 2),
* MotherShip at (28, 18), (32, 16),
* PlayerScore at (96, 2), (6, 8), PlayerScore at (88, 2), (6, 8), PlayerScore at (80, 2), (6, 8), PlayerScore at (72, 2), (6, 8), PlayerScore at (64, 2), (6, 8), PlayerScore at (56, 2), (6, 8),
* Lives at (47, 192), (8, 8), Lives at (31, 192), (8, 8), Lives at (15, 192), (8, 8),
* Health at (96, 192), (8, 8)

Notice that first 2 digits being the coordinates, the second two being their dimensions.

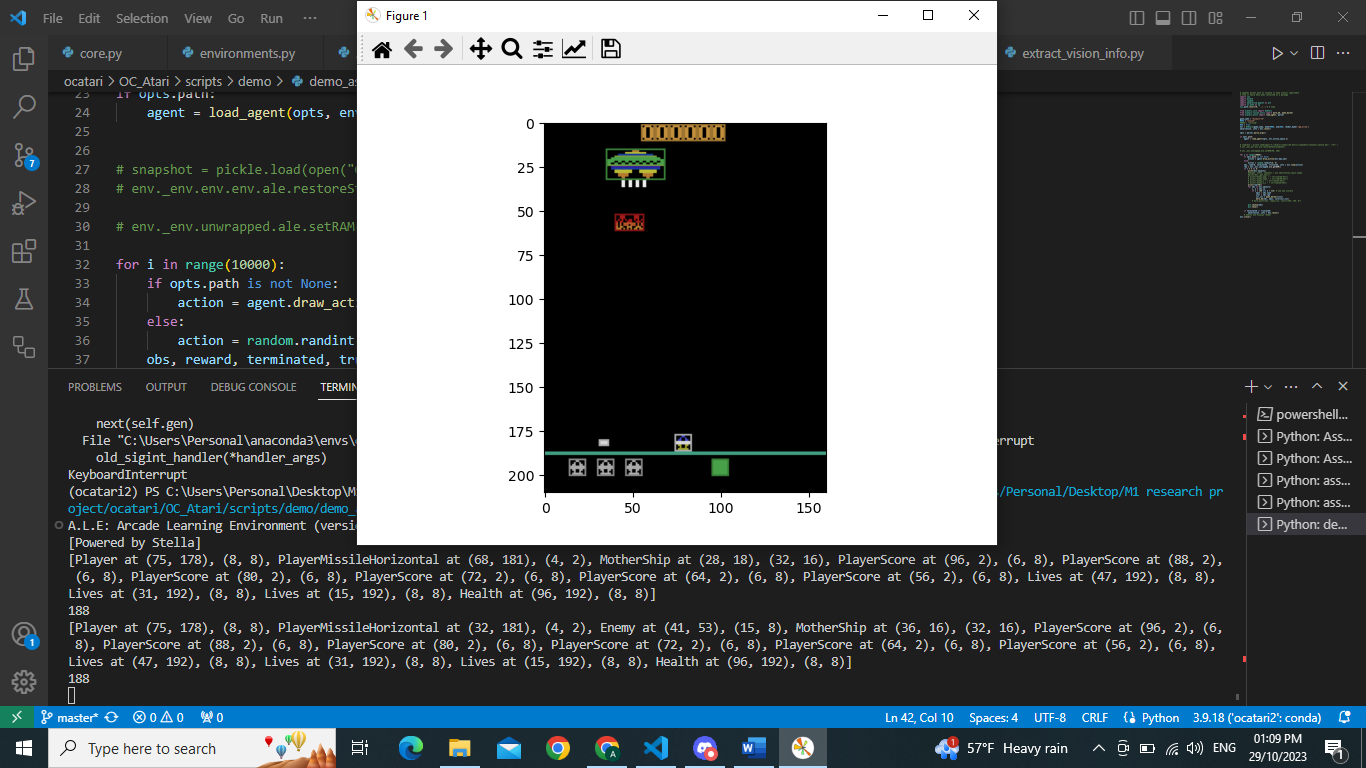
Also the player score has 6 objects 1 for each digit. Each life is also counted as an object.

******

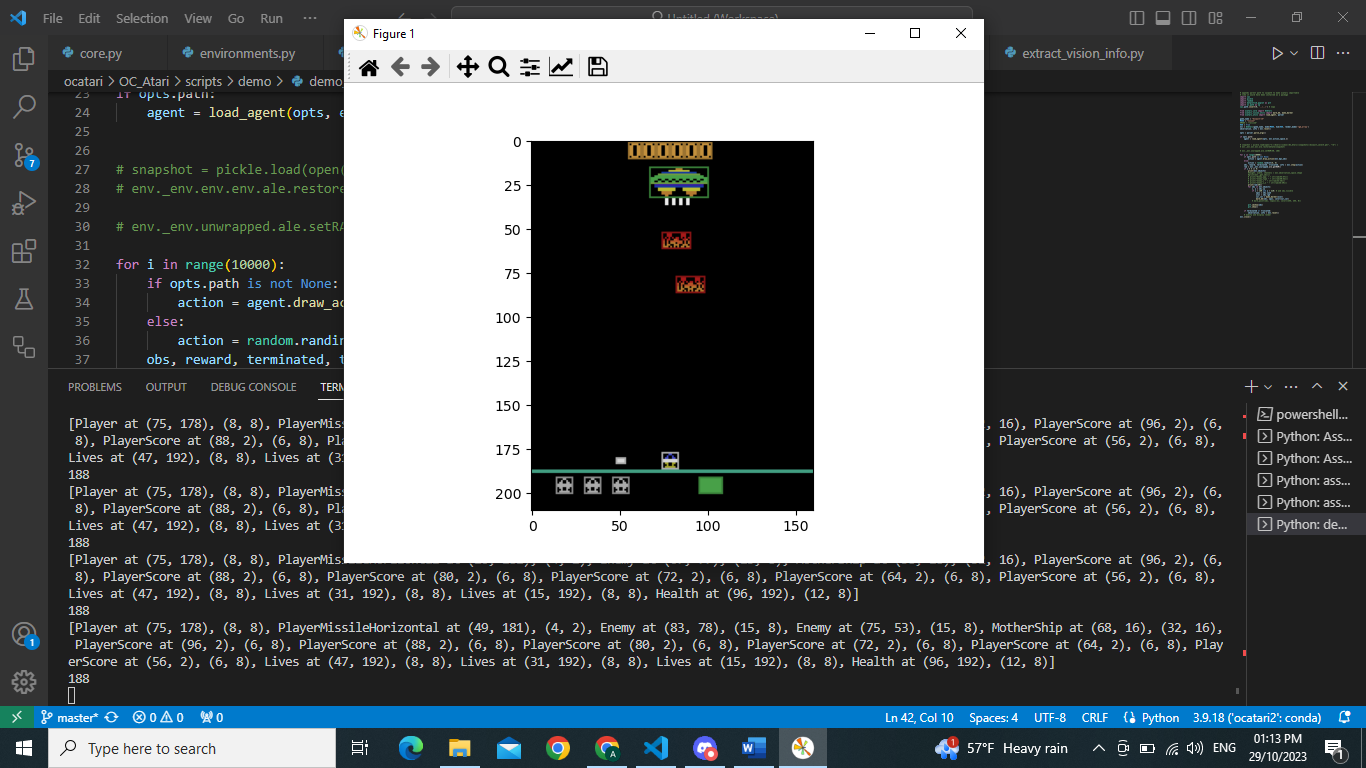
The second observation was made after w new enemy has appeared in the frame resulting in 14 objects on screen:

* Player at (75, 178), (8, 8),
* PlayerMissileHorizontal at (32, 181), (4, 2),
* **Enemy** at (41, 53), (15, 8),
* MotherShip at (36, 16), (32, 16),
* PlayerScore at (96, 2), (6, 8), PlayerScore at (88, 2), (6, 8), PlayerScore at (80, 2), (6, 8), PlayerScore at (72, 2), (6, 8), PlayerScore at (64, 2), (6, 8), PlayerScore at (56, 2), (6, 8),
* Lives at (47, 192), (8, 8), Lives at (31, 192), (8, 8), Lives at (15, 192), (8, 8),
* Health at (96, 192), (8, 8)]

Notice how we see that the array size increases with each new object that appears on the screen(here the new enemy)



Another enemy appears:



***2)USING “RAW”:***

As we can see here the initial observation was made when we had

Number of objects: 8

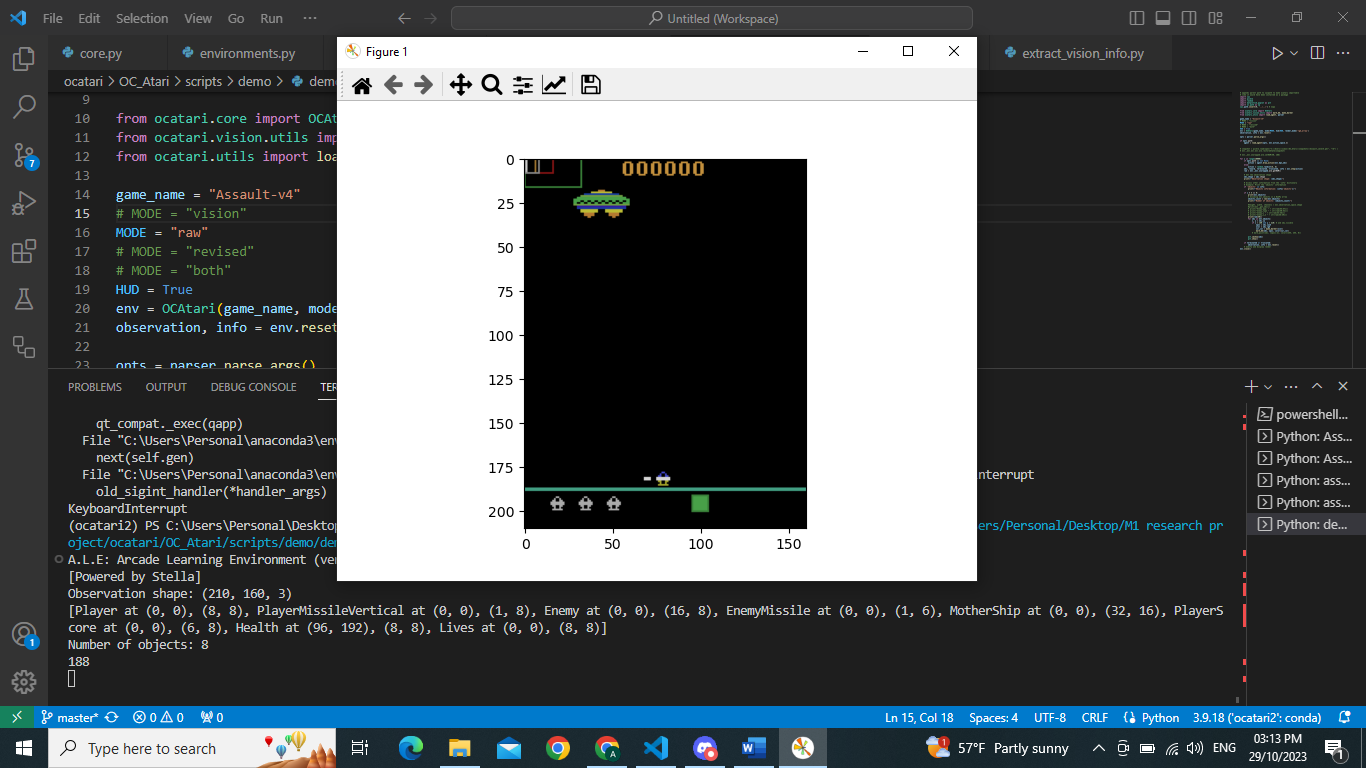
Observation shape: (210, 160, 3)

* [Player at (0, 0), (8, 8),
* PlayerMissileVertical at (0, 0), (1, 8),
* Enemy at (0, 0), (16, 8),
* EnemyMissile at (0, 0), (1, 6),
* MotherShip at (0, 0), (32, 16),
* PlayerScore at (0, 0), (6, 8),
* Health at (96, 192), (8, 8),
* Lives at (0, 0), (8, 8)]

First 2 digits being the coordinates, the second two being their dimensions.

We have “Enemy at” and “EnemyMissile at” even though we don’t have them on screen yet.

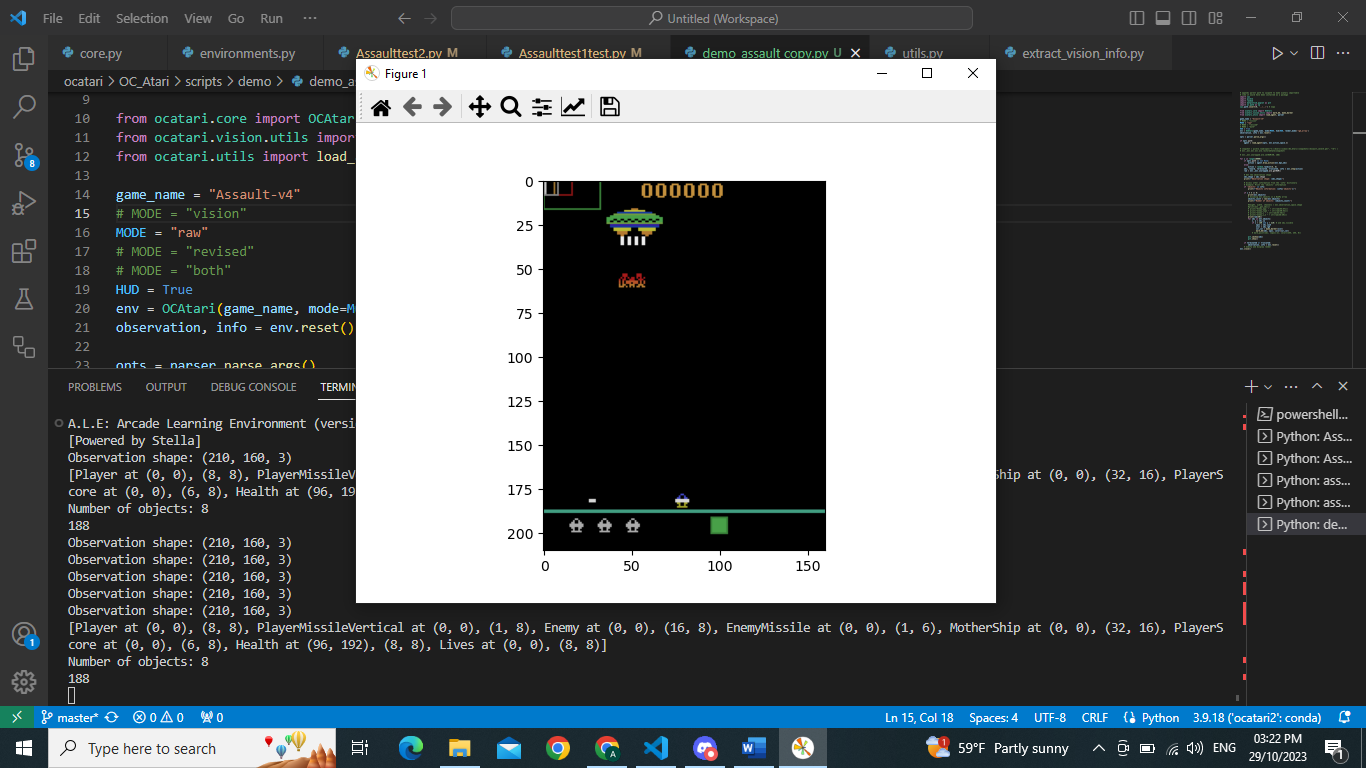
Notice how we cannot keep track of the position of each object.



Observation shape: (210, 160, 3)

Number of objects: 8

* [Player at (0, 0), (8, 8),
* PlayerMissileVertical at (0, 0), (1, 8),
* Enemy at (0, 0), (16, 8),
* EnemyMissile at (0, 0), (1, 6),
* MotherShip at (0, 0), (32, 16),
* PlayerScore at (0, 0), (6, 8),
* Health at (96, 192), (8, 8),
* Lives at (0, 0), (8, 8)]



***3)USING “REVISED”:***

As we can see here the initial observation was made when we had

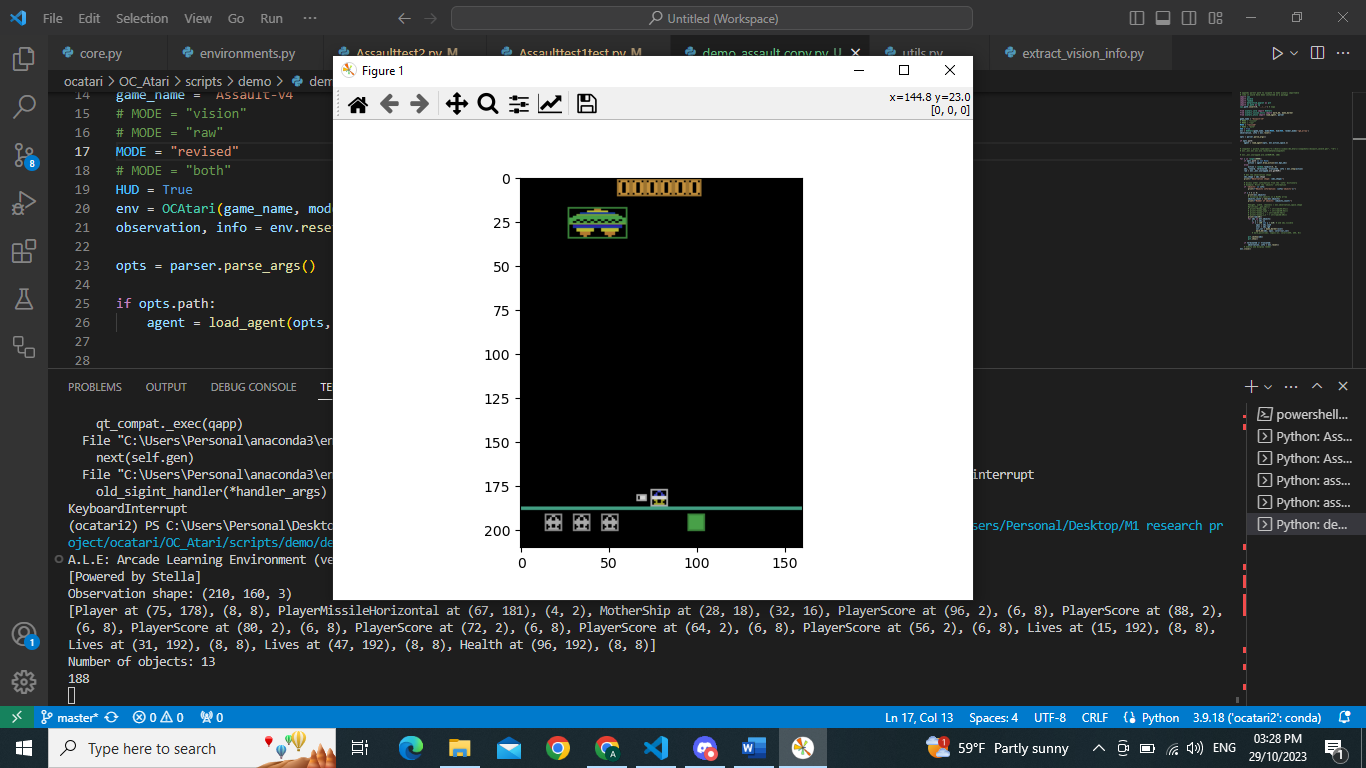
Number of objects: 13

Observation shape: (210, 160, 3)

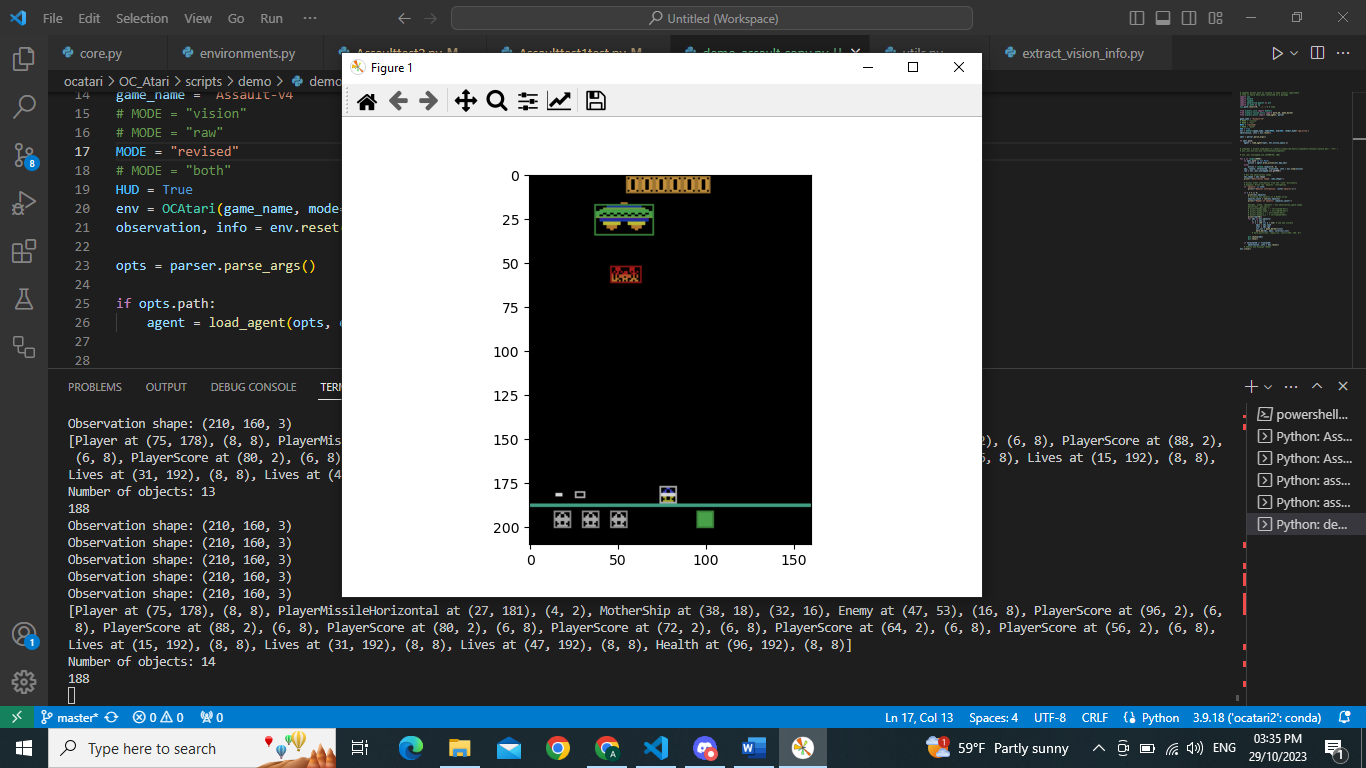
* [Player at (75, 178), (8, 8),
* PlayerMissileHorizontal at (67, 181), (4, 2),
* MotherShip at (28, 18), (32, 16),
* PlayerScore at (96, 2), (6, 8), PlayerScore at (88, 2), (6, 8), PlayerScore at (80, 2), (6, 8), PlayerScore at (72, 2), (6, 8), PlayerScore at (64, 2), (6, 8), PlayerScore at (56, 2), (6, 8),
* Lives at (15, 192), (8, 8), Lives at (31, 192), (8, 8), Lives at (47, 192), (8, 8),
* Health at (96, 192), (8, 8)]

First 2 digits being the coordinates, the second two being their dimensions.

Very similar to using “vision”. player score has 6 objects 1 for each digit. Each life is also counted as an object, but for some reason live are counted backwards.



Similar behavior however; once a new object appears, it adds it.



***4)USING “BOTH”:***

As we can see here the initial observation was made when we had

Number of objects: 13

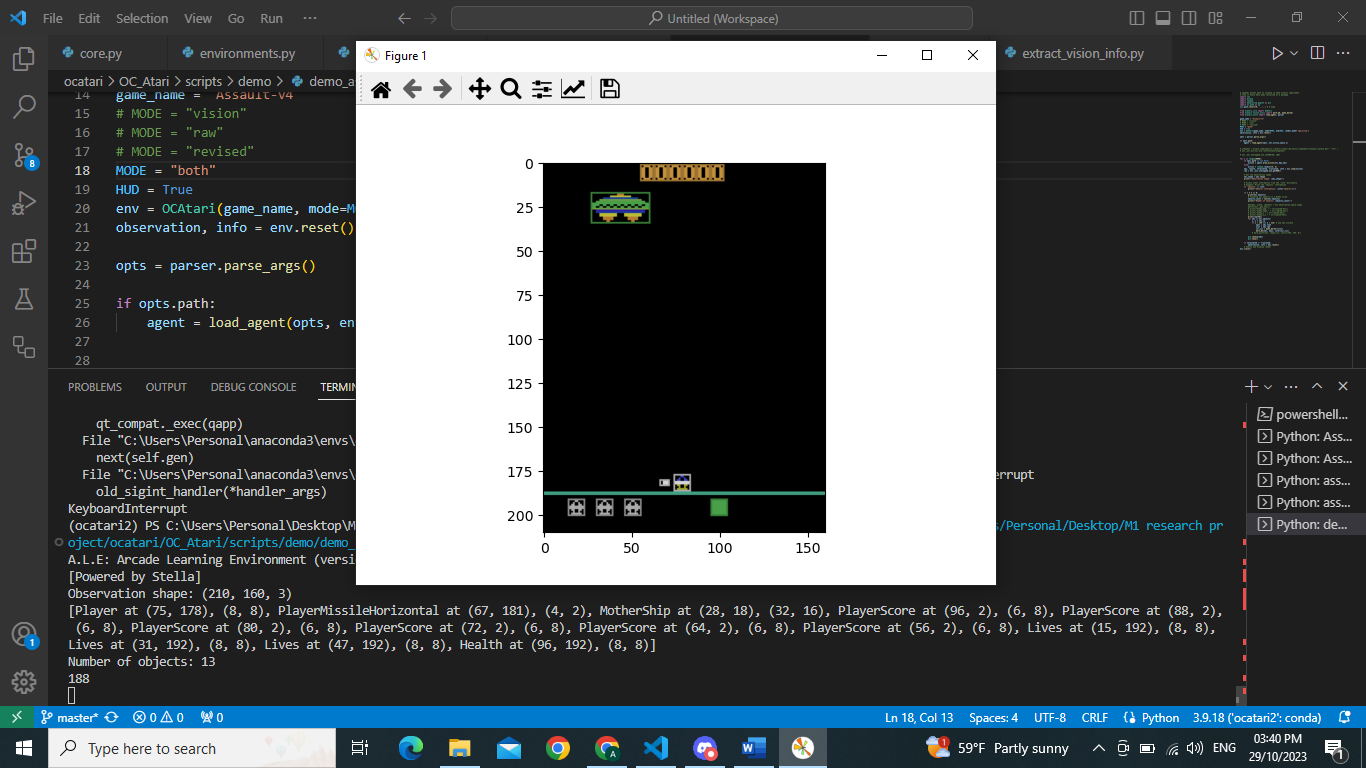
Observation shape: (210, 160, 3)

* [Player at (75, 178), (8, 8), PlayerMissileHorizontal at (67, 181), (4, 2), MotherShip at (28, 18), (32, 16), PlayerScore at (96, 2), (6, 8), PlayerScore at (88, 2), (6, 8), PlayerScore at (80, 2), (6, 8), PlayerScore at (72, 2), (6, 8), PlayerScore at (64, 2), (6, 8), PlayerScore at (56, 2), (6, 8), Lives at (15, 192), (8, 8),

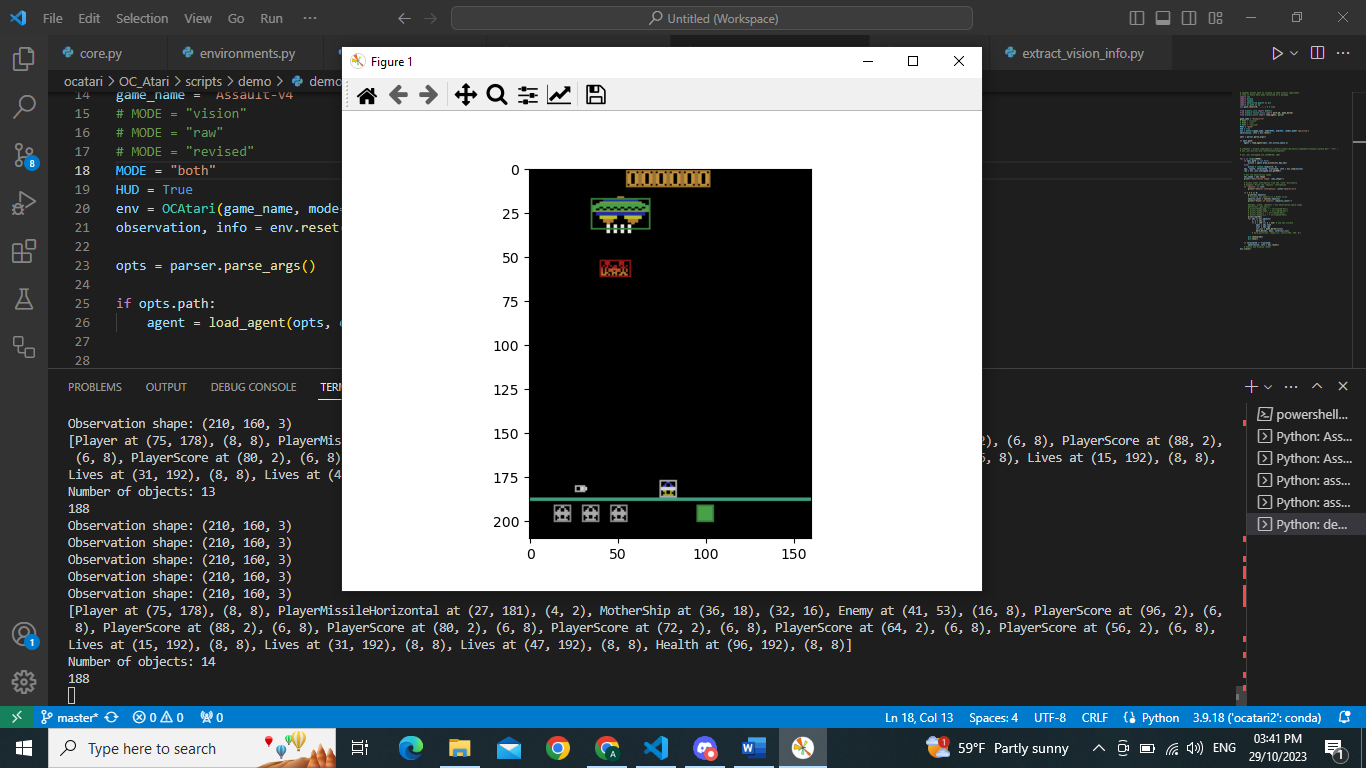
Lives at (31, 192), (8, 8), Lives at (47, 192), (8, 8), Health at (96, 192), (8, 8)]

Number of objects: 13First 2 digits being the coordinates, the second two being their dimensions.

Very similar to using “vision”. player score has 6 objects 1 for each digit. Each life is also counted as an object, but for some reason live are counted backwards.

  
Number of objects: 14

* [Player at (75, 178), (8, 8),
* PlayerMissileHorizontal at (27, 181), (4, 2),
* MotherShip at (36, 18), (32, 16),
* Enemy at (41, 53), (16, 8),
* PlayerScore at (96, 2), (6, 8), PlayerScore at (88, 2), (6, 8), PlayerScore at (80, 2), (6, 8), PlayerScore at (72, 2), (6, 8), PlayerScore at (64, 2), (6, 8), PlayerScore at (56, 2), (6, 8),
* Lives at (15, 192), (8, 8), Lives at (31, 192), (8, 8), Lives at (47, 192), (8, 8),
* Health at (96, 192), (8, 8)]



How they extract:  
  
***OC\_Atari\ocatari\core.py***

if mode == "vision":

            self.detect\_objects = detect\_objects\_vision

            self.step = self.\_step\_vision

        elif mode == "raw":

            self.detect\_objects = detect\_objects\_raw

            self.step = self.\_step\_ram

        elif mode == "revised":

            self.max\_objects = get\_max\_objects(self.game\_name, self.hud)

            self.detect\_objects = detect\_objects\_revised

            self.step = self.\_step\_ram

        elif mode == "both":

            self.detect\_objects\_v = detect\_objects\_vision

            self.detect\_objects\_r = detect\_objects\_revised

            self.objects\_v = init\_objects(self.game\_name, self.hud)

            self.step = self.\_step\_test

1. ***vision: (in ocatari\vision\assault.py)***  
   for example:

\_detect\_objects\_assault(objects, obs, hud=False):

1. It searches for the player character, identified by the color corresponding to the player. This is done using the **find\_mc\_objects** function.

find\_mc\_objects(image, colors, size=None, tol\_s=10, position=None, tol\_p=2,

                    min\_distance=10, closing\_active=True, closing\_dist=3,

                    minx=0, miny=0, maxx=160, maxy=210, all\_colors=True)

(The find\_mc\_objects function detects and isolates objects in images based on their colors, and it can be customized with various parameters to control the detection process, filter results, and prevent duplicate detections.)

If the player character's position meets specific criteria (e.g., in assault, y-coordinate > 170 and x-coordinate > 4), it is categorized as a player object and added to the list of objects.

1. It looks for player missiles by checking for colors associated with player missiles.

Depending on the color, it distinguishes between horizontal and vertical player missiles and creates corresponding objects. (we have already all the possible color values for each element)

1. Searches for enemy ships by analyzing the colors associated with enemy ships. It checks the position and size of the detected enemy ships and categorizes them as enemy objects.

The specific enemy ship color determines the type of enemy (different colors correspond to different enemy ship types).

1. Detects enemy missiles by looking for colors associated with enemy missiles.

Categorizes enemy missiles based on color and position.

1. Identifies mother ships by analyzing the colors associated with mother ships.

It considers the position and size of the detected mother ships.

Different colors correspond to different mother ship types.

1. If HUD elements (such as the player's score, lives, and health indicators) need to be detected, the function searches for their respective colors.

Detected elements are categorized and added as objects.

1. ***“RAW” (in ocatari\ram\assault.py)***

\_detect\_objects\_assault\_raw(info, ram\_state)

1. The function calculates the player's score by combining values stored in different RAM addresses. It multiplies the values at RAM addresses 0, 1, and 2 by specific factors and sums them up to get the player's score.
2. The player's horizontal (X) position is extracted from RAM address 16.
3. The X position of the player's missile is extracted from RAM address 39, and the Y position from RAM address 67. These values represent the position of the player's missile on the screen.

Etc … (same procedure for every element)

1. Information about the player's sprite (RAM address 30) and the number of lives (RAM address 101) is extracted. This data relates to the player character and the number of lives remaining.
2. ***Revised: (in ocatari\ram\assault.py)***

\_detect\_objects\_assault\_revised(objects, ram\_state, hud=False)

The \_detect\_objects\_assault\_revised function is designed to extract information about various game objects in the OCAtari game "Assault" when running in the "revised" mode, primarily using RAM state information.

1. It detects the player object's position based on RAM state information.

Calculates the player's x-coordinate (x) based on specific formulas and adjusts for screen wrapping if necessary.

Appends a Player object to the objects list with the detected position.

1. Calculates the missile's position (x, y) and dimensions (w, h) based on RAM state information and specific formulas.

Appends a PlayerMissileVertical object to the objects list if the missile is detected.

1. Detects the player's horizontal missile (if present).

It keeps track of the missile's position (horizontal\_pos) and updates it based on the player's actions.

Appends it if detected.

1. Detects the mother ship's position and color change.

Calculates the mother ship's position based on RAM state information.

Adjusts for screen wrapping if necessary.

Changes the color if specified by RAM state.

Appends a MotherShip object to the objects list.

1. Detects **up to three** enemy objects.

Calculates the position, size, and color of each enemy based on RAM state information.

Appends Enemy objects to the objects list for each detected enemy.

**Handles the case where two enemies appear as a single object.**

1. Detects the enemy missile's position, size, and color based on RAM state information.

Appends it if detected. (1 missile per enemy?)

1. If the hud flag is set to True, it detects and appends HUD elements, including score, lives, and health, to the objects list based on RAM state information.
2. ***BOTH:***

if mode == "both":

            self.detect\_objects\_v = detect\_objects\_vision

            self.detect\_objects\_r = detect\_objects\_revised

            self.objects\_v = init\_objects(self.game\_name, self.hud)

As we can see here “both” mode uses a combination of vision and revised.

Taking objects from both, following about the same process.

The \_step\_test method is used to interact with the environment and allows you to choose whether to use visual or modified game state information for interaction.

NB: Function to get max objects that appear on screen at once:

print(\_get\_max\_objects(hud=False))

MAX\_NB\_OBJECTS = {'Player': 1, 'PlayerMissileVertical': 1, 'PlayerMissileHorizontal': 1, 'MotherShip': 1,

                  'Enemy': 9, 'EnemyMissile': 1}

MAX\_NB\_OBJECTS\_HUD = {'PlayerScore': 6, 'Lives': 3, 'Health': 1}

***OC\_Atari\scripts\ReverseEngineeringHelper.py***

Helps rebuild another demo