Position Class:

This defines a class called Position, which is designed to store position data, primarily in the form of (x, y) coordinates. The class has two attributes, x and y, representing the x and y values of a position vector. The constructor method \_\_init\_\_() initializes these attributes with default values of 0 if no arguments are provided during instantiation. The \_\_str\_\_() method is defined to provide a string representation of a Position object in the format "(x, y)". This method converts the x and y attributes to strings and concatenates them within parentheses to form the desired representation. Overall, this class allows for easy manipulation and representation of 2D position data.

Class SpaceType:  
 We define a enumeration class named SpaceType, which represents different types of spaces within a system. The class inherits from the Enum class provided by the enum module in Python. Within the SpaceType class, there are three enumeration members: EMPTY, DROP\_OFF, and PICK\_UP. Each member represents a distinct type of space and is associated with a string value. Specifically, EMPTY denotes an empty space, DROP\_OFF represents a space designated for dropping off items, and PICK\_UP signifies a space designated for picking up items. These enumeration members are defined with string values "E", "D", and "P" respectively, which serve as identifiers for each type of space. This enumeration allows for a clear and concise way to represent and differentiate between different types of spaces within a system, making the code more readable and maintainable. Additionally, using an enumeration ensures that only predefined values can be assigned to variables representing space types, reducing the likelihood of errors in the code.

Class Space:

We define a class called Space, designed to represent individual spaces. Each Space object has attributes including its type, indicating whether it's empty, designated for picking up items, or for dropping off items, represented by the SpaceType enumeration. Additionally, it tracks the number of blocks currently on the space (num\_blocks), and the maximum number of blocks allowed on the space (max\_blocks). The constructor method \_\_init\_\_() initializes these attributes, with default values provided for type (defaulting to an empty space), num\_blocks (defaulted to 0), and max\_blocks (defaulted to 5). It also performs validation to ensure num\_blocks does not exceed max\_blocks and that neither of them is negative. The method \_init\_reward() is used to set the reward attribute based on the type of the space. This reward represents the benefit or penalty associated with occupying or interacting with the space. The \_\_str\_\_() method is defined to provide a string representation of a Space object in the format "(SpaceType, num\_blocks, max\_blocks)". This method converts the attributes to strings and concatenates them within parentheses to form the desired representation. This class allows for the representation and management of individual spaces within a game or simulation environment, with customizable attributes and behavior.

Class Environment:

We define a class named Environment, tailored to represent a 2D board consisting of individual spaces, like to those found in tabletop or grid-based games. This class encapsulates attributes essential for defining the environment, including the number of rows (n) and columns (m) in the 2D grid, as well as the board itself (pd\_world), represented as a 2D list of Space objects. The \_\_init\_\_() method initializes instances of the Environment class, with default values of 5 assigned to n and m, creating an empty grid of spaces. The set() method allows for placing a Space object at a specified position on the board, while the at() method retrieves the Space object at a given position. Additionally, the within\_bounds() method verifies whether a position lies within the bounds of the environment. Lastly, the \_\_str\_\_() method generates a string representation of the environment by iterating through each space on the board and appending its type to a string, resulting in a visual depiction of the grid. This Environment class serves as a foundational component for modeling and simulating grid-based environments.

Class Direction:  
 We define an enumeration class called Direction, serving to represent various directions and actions within a system. Enumerations in Python offer a way to create symbolic names bound to unique, constant integer values. In this context, the Direction enum encompasses seven members: NORTH, SOUTH, EAST, and WEST representing cardinal directions, along with PICKUP, DROPOFF, and IDLE representing specific actions. Each member is associated with a distinct integer value, starting from 0 for NORTH and incrementing by one for each subsequent member. By using this enumeration, developers can employ clear and intuitive names to reference different directions or actions within their code, enhancing readability and maintainability.

Classes ActorType and Actor:

The ActorType enumeration class defines symbolic names for different colors of actors within a system. It includes members such as RED, BLUE, and BLACK, each mapped to a single-character string representing the respective color. This enumeration provides a convenient and readable way to represent actor colors in the codebase.

The Actor class represents individual actors that can exist within an environment. It encapsulates attributes such as type, position, and has\_box, representing the color, position in a 2D space, and whether the actor is currently holding a box, respectively. The \_\_init\_\_ method initializes instances of the Actor class with a specified type and an optional position, defaulting to the origin (0, 0) if not provided. Upon initialization, the actor's has\_box attribute is set to False to indicate that it is not holding a box.

Together, these components offer a flexible and organized approach to representing actors of different colors within a 2D environment, facilitating efficient management and manipulation of actor data.

Class State: **MAY NEED TO EXPAND MORE ON THE FUCNTIONS**

We define class called State, which represents a specific snapshot of an environment containing actors and various spatial data. The class includes several attributes to capture the state of the environment, such as positions and data for dropoff locations (dropoff\_pos and dropoff\_data), pickup locations (pickup\_pos and pickup\_data), actor positions (actor\_pos), actor distances (actor\_dist), and a list of actors (actors). The \_\_init\_\_ method initializes instances of the State class by initializing these attributes based on the provided environment and actors. The is\_terminal method checks if the state is terminal, meaning no actors are holding boxes, no boxes are left to pick up, and all dropoff locations are filled to their maximum capacity. Private methods \_init\_env\_locations and \_init\_actors initialize dropoff and pickup locations as well as actors and their distances, respectively. The update method updates the state based on actor actions, such as picking up or dropping off boxes. Finally, the \_\_str\_\_ method generates a string representation of the state by concatenating the positions and data of dropoff locations, pickup locations, and actor positions. This State class provides a comprehensive representation of the state of an environment with actors, facilitating efficient management and manipulation of state data.

Class QTable: **Can prob expand if needed**

The QTable class encapsulates the functionality necessary for managing the Q-values of state-action pairs, which are pivotal in reinforcement learning algorithms like Q-learning. Upon instantiation, the class initializes a nested dictionary named table using defaultdict, allowing efficient storage of Q-values indexed by states and actions. Methods like get\_q and set\_q facilitate retrieval and setting of Q-values for specific state-action pairs. Additionally, utility methods such as get\_max\_q and get\_max\_action enable the retrieval of the maximum Q-value and the corresponding action for a given state, respectively. Furthermore, the print\_table method offers the capability to export the entire Q-table to a CSV file, providing a means to analyze and visualize the Q-values externally. With these functionalities, the QTable class serves as a versatile tool for managing and manipulating Q-values, essential for the learning and decision-making processes within reinforcement learning algorithms.

Def Transition:

The transition function facilitates the movement of an agent within an environment by updating its position based on the provided action. Initially, the function extracts the current x and y coordinates from the current\_pos object, representing the agent's position. Then, through a series of conditional statements, the function determines the next position of the agent depending on the action specified. For instance, if the action is Direction.NORTH, the agent's position is incremented by one unit in the y-direction to move upwards. Similarly, actions like Direction.SOUTH, Direction.WEST, and Direction.EAST correspond to movements downwards, leftwards, and rightwards, respectively. If the action is Direction.PICKUP, Direction.DROPOFF, or Direction.IDLE, indicating no movement, the agent's position remains unchanged. Notably, in this simplified implementation, there's no explicit boundary checking or handling of edge cases like collision with walls. The function assumes an infinite grid where all states are considered valid. Finally, the function returns the calculated next\_pos, reflecting the agent's position after executing the specified action.

Def QLearning:

The Q\_learning function serves as the implementation of the Q-learning algorithm, facilitating the learning process of an agent within a given environment. It operates based on several steps: first, it extracts the current position of the agent and retrieves information about the space at that position from the environment. Subsequently, it utilizes a transition function to determine the next position of the agent based on the action chosen. Following this, the function calculates the immediate reward associated with the chosen action, considering the type of space the agent is interacting with. With the current state and action, it retrieves the current Q-value from the Q-table, representing the expected cumulative reward for taking that action from the current state. Upon simulating the next state by updating the current state with the chosen action, the function invokes a policy function to determine the next action to be taken by the agent. It then iterates through possible actions in the next state to identify the maximum Q-value, reflecting the agent's anticipation of future rewards. Utilizing the Q-learning equation, which balances existing knowledge with newly acquired information, the function updates the Q-value for the current state-action pair in the Q-table. Finally, it updates the current state with the next position and action, ensuring that the agent's state representation aligns with its progression through the environment. Through these sequential steps, the Q\_learning function enables the agent to learn and improve its decision-making process over time, ultimately leading to optimal behavior in the given environment.

Def SARSA:

The SARSA function implements the SARSA (State-Action-Reward-State-Action) algorithm, a type of reinforcement learning method that updates Q-values based on the agent's interactions with the environment. The function begins by extracting the current position of the agent and retrieving information about the space at that position. It then retrieves the current Q-value for the state-action pair from the Q-table. Next, using a transition function, it determines the next position of the agent based on the chosen action. Based on the type of space and action, the function calculates the immediate reward associated with the action. It updates the current state with the next position and action, simulating the agent's transition to the next state. Through a policy function, it determines the next action to be taken by the agent and retrieves the corresponding Q-value for the next state-action pair. Utilizing the SARSA equation, the function calculates the updated Q-value, incorporating the immediate reward and the estimated future rewards discounted by a factor gamma. Finally, it updates the Q-value in the Q-table for the current state-action pair and returns the next action to be taken by the agent.

Class Policy:

The Policy class encapsulates various decision-making strategies for an agent navigating an environment in reinforcement learning scenarios. Each method within the class represents a distinct policy: PRANDOM selects actions randomly, PGREEDY prioritizes actions with the highest Q-values, and PEXPLOIT balances between exploiting known actions and exploring new ones based on ε-greedy principles. These methods check the feasibility of actions such as picking up or dropping off boxes and determine valid actions based on the agent's current position and environment constraints. By providing these different policies, the Policy class empowers agents to adapt their decision-making processes, making efficient learning and the development of optimal behaviors over time.

Class Run:

The Run class controls the execution of various reinforcement learning algorithms, including Q-Learning and SARSA, along with different exploration and exploitation strategies. In its methods, such as explore\_q, explore\_sarsa, and greedy\_q, the class iterates over a predetermined number of episodes, allowing the agent to interact with the environment, select actions based on the specified policies, and update the Q-table accordingly. Through exploration strategies like random action selection and exploitation strategies like selecting actions based on the highest Q-values, the class enables the agent to learn and refine its behavior over time. The methods handle terminal states by resetting the environment and recalculating actions as necessary, ensuring continuous learning and adaptation.

Def Main:  
 The main function starts by initializing random seeds for reproducibility and then sets up a grid environment with specific drop-off and pick-up locations. Actors are placed within this environment, each assigned a color and starting position. The code then proceeds to conduct Q-learning exploration, followed by training using a greedy exploitation strategy, and SARSA exploration. For each phase, the resulting Q-tables are saved to CSV files for later analysis. Overall, this main function orchestrates the execution of reinforcement learning algorithms and provides a structured framework for evaluating their performance in solving tasks within a defined environment.