In [1]:
```python
#import libraries
import numpy as np
```

In [2]:
```python
#define the shape of the environment (i.e., its states)
environment_rows = 11
environment_columns = 11

#Create a 3D numpy array to hold the current Q-values for each state and
#The array contains 11 rows and 11 columns (to match the shape of the env
#The "action" dimension consists of 4 layers that will allow us to keep t
#each state (see next cell for a description of possible actions).
#The value of each (state, action) pair is initialized to 0.
q_values = np.zeros((environment_rows, environment_columns, 4))
```

In [3]:
```python
#define actions
#numeric action codes: 0 = up, 1 = right, 2 = down, 3 = left
actions = ['up', 'right', 'down', 'left']
```

In [4]:
```python
#Create a 2D numpy array to hold the rewards for each state.
#The array contains 11 rows and 11 columns (to match the shape of the env
rewards = np.full((environment_rows, environment_columns), -100.)
rewards[0, 5] = 100. #set the reward for the packaging area (i.e., the go

#define aisle locations  for rows 1 through 9
aisles = {} #store locations in a dictionary
aisles[1] = [i for i in range(1, 10)]
aisles[2] = [1, 7, 9]
aisles[3] = [i for i in range(1, 8)]
aisles[3].append(9)
aisles[4] = [3, 7]
aisles[5] = [i for i in range(11)]
aisles[6] = [5]
aisles[7] = [i for i in range(1, 10)]
aisles[8] = [3, 7]
aisles[9] = [i for i in range(11)]

#print(aisles)

#set the rewards for all aisle locations
for row_index in range(1, 10):
  for column_index in aisles[row_index]:
    rewards[row_index, column_index] = -1.

#print rewards matrix
for row in rewards:
  print(row)
```

```
[-100. -100. -100. -100. -100.  100. -100. -100. -100. -100. -100.]
[-100.   -1.   -1.   -1.   -1.   -1.   -1.   -1.   -1.   -1. -100.]
[-100.   -1. -100. -100. -100. -100. -100.   -1. -100.   -1. -100.]
[-100.   -1.   -1.   -1.   -1.   -1.   -1.   -1. -100.   -1. -100.]
[-100. -100. -100.   -1. -100. -100. -100.   -1. -100. -100. -100.]
[-1. -1. -1. -1. -1. -1. -1. -1. -1. -1. -1.]
[-100. -100. -100. -100. -100.   -1. -100. -100. -100. -100. -100.]
[-100.   -1.   -1.   -1.   -1.   -1.   -1.   -1.   -1.   -1. -100.]
[-100. -100. -100.   -1. -100. -100. -100.   -1. -100. -100. -100.]
[-1. -1. -1. -1. -1. -1. -1. -1. -1. -1. -1.]
[-100. -100. -100. -100. -100. -100. -100. -100. -100. -100. -100.]
```

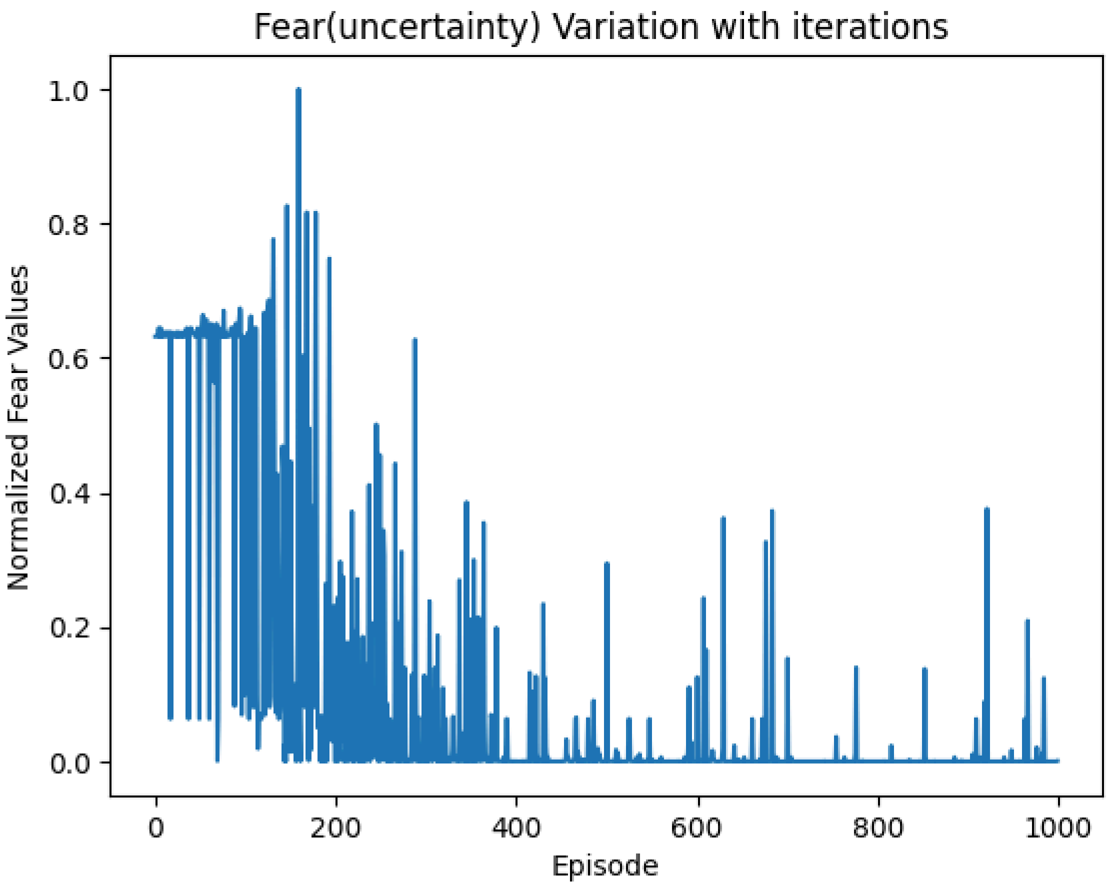
In [5]:

```
#define a function that determines if the specified location is a termina
def is_terminal_state(current_row_index, current_column_index):
  #if the reward for this location is -1, then it is not a terminal state
  if rewards[current_row_index, current_column_index] == -1.:
    return False
  else:
    return True

#define a function that will choose a random, non-terminal starting locat
def get_starting_location():
  #get a random row and column index
  current_row_index = np.random.randint(environment_rows)
  current_column_index = np.random.randint(environment_columns)
  #continue choosing random row and column indexes until a non-terminal s
  #(i.e., until the chosen state is a 'white square').
  while is_terminal_state(current_row_index, current_column_index):
    current_row_index = np.random.randint(environment_rows)
    current_column_index = np.random.randint(environment_columns)
  return current_row_index, current_column_index

#define an epsilon greedy algorithm that will choose which action to take
def get_next_action(current_row_index, current_column_index, epsilon):
  #if a randomly chosen value between 0 and 1 is less than epsilon,
  #then choose the most promising value from the Q-table for this state.
  if np.random.random() < epsilon:
    return np.argmax(q_values[current_row_index, current_column_index])
  else: #choose a random action
    return np.random.randint(4)

#define a function that will get the next location based on the chosen ac
def get_next_location(current_row_index, current_column_index, action_ind
  new_row_index = current_row_index
  new_column_index = current_column_index
  if actions[action_index] == 'up' and current_row_index > 0:
    new_row_index -= 1
  elif actions[action_index] == 'right' and current_column_index < enviro
    new_column_index += 1
  elif actions[action_index] == 'down' and current_row_index < environmen
    new_row_index += 1
  elif actions[action_index] == 'left' and current_column_index > 0:
    new_column_index -= 1
  return new_row_index, new_column_index

#Define a function that will get the shortest path between any location w
#the agent is allowed to travel and the item packaging location.
def get_shortest_path(start_row_index, start_column_index):
  #return immediately if this is an invalid starting location
  if is_terminal_state(start_row_index, start_column_index):
```

```python
        return []
    else: #if this is a 'legal' starting location
        current_row_index, current_column_index = start_row_index, start_colu
        shortest_path = []
        shortest_path.append([current_row_index, current_column_index])
        #continue moving along the path until we reach the goal
        while not is_terminal_state(current_row_index, current_column_index):
            #get the best action to take
            action_index = get_next_action(current_row_index, current_column_in
            #move to the next location on the path, and add the new location to
            current_row_index, current_column_index = get_next_location(current
            shortest_path.append([current_row_index, current_column_index])
        return shortest_path
```

In [6]:
```python
import matplotlib.pyplot as plt

#define training parameters
epsilon = 0.9 #the percentage of time when we should take the best action
discount_factor = 0.9 #discount factor for future rewards
learning_rate = 0.9 #the rate at which the agent should learn
fear_values=[]
hope_values=[]
#run through 1000 training episodes
for episode in range(1000):
    #get the starting location for this episode
    row_index, column_index = get_starting_location()
    sum_td=0

    #continue taking actions (i.e., moving) until we reach a terminal sta
    #(i.e., until we reach the item packaging area or crash into an item
    while not is_terminal_state(row_index, column_index):
        #choose which action to take (i.e., where to move next)
        action_index = get_next_action(row_index, column_index, epsilon)

        #perform the chosen action, and transition to the next state (i.e
        old_row_index, old_column_index = row_index, column_index #store
        row_index, column_index = get_next_location(row_index, column_ind

        #receive the reward for moving to the new state, and calculate th
        reward = rewards[row_index, column_index]
        old_q_value = q_values[old_row_index, old_column_index, action_in
        temporal_difference = reward + (discount_factor * np.max(q_values
        sum_td+=abs(temporal_difference)
        #update the Q-value for the previous state and action pair
        new_q_value = old_q_value + (learning_rate * temporal_difference)
        q_values[old_row_index, old_column_index, action_index] = new_q_v

    fear_values.append(sum_td)


normalized_fear_values = (fear_values - np.min(fear_values)) / (np.max(fe
hope_values= [1 if x == 0 else 1/x for x in normalized_fear_values]
# Normalize hope values
normalized_hope_values = (hope_values - np.min(hope_values)) / (np.max(ho


# Plotting normalized fear values
plt.plot(normalized_fear_values)
plt.xlabel('Episode')
plt.ylabel('Normalized Fear Values')
```

```python
plt.title('Fear(uncertainty) Variation with iterations ')
plt.show()
```



In [ ]: