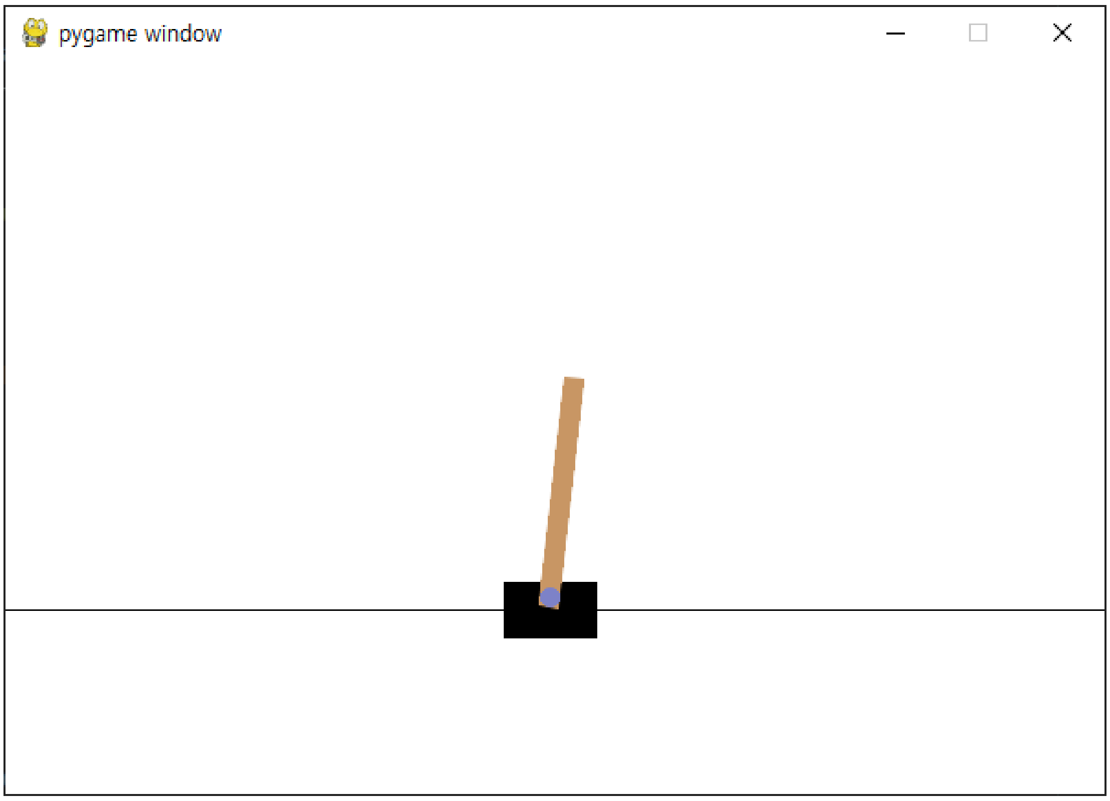
In [2]:
```
from collections import deque
import random
import numpy as np
import gym

class ReplayBuffer:
    def __init__(self, buffer_size, batch_size):
        self.buffer = deque(maxlen=buffer_size)
        self.batch_size = batch_size

    def add(self, state, action, reward, next_state, done):
        data = (state, action, reward, next_state, done)
        self.buffer.append(data)

    def __len__(self):
        return len(self.buffer)

    def get_batch(self):
        data = random.sample(self.buffer, self.batch_size)

        state = np.stack([x[0] for x in data])
        action = np.array([x[1] for x in data])
        reward = np.array([x[2] for x in data])
        next_state = np.stack([x[3] for x in data])
        done = np.array([x[4] for x in data]).astype(np.int32)
        return state, action, reward, next_state, done

env = gym.make('CartPole-v1', render_mode='human') # Using CartPole-v1 as it's m
                                                  # and seen in the previous requ
                                                  # The image image_0df8ea.png sh
                                                  # but the logic is generally co
replay_buffer = ReplayBuffer(buffer_size=10000, batch_size=32)

for episode in range(10):
    state_tuple = env.reset()
    state = state_tuple[0]
    done = False

    while not done:
        action = 0
        next_state_tuple = env.step(action)
        next_state, reward, terminated, truncated, info = next_state_tuple
        done = terminated or truncated

        replay_buffer.add(state, action, reward, next_state, done)
        state = next_state

if len(replay_buffer) >= replay_buffer.batch_size: # Check if buffer has enough
    state, action, reward, next_state, done = replay_buffer.get_batch()
    print(state.shape)
    print(action.shape)
    print(reward.shape)
    print(next_state.shape)
    print(done.shape)
else:
    print(f"Not enough samples in replay buffer ({len(replay_buffer)}) to get a

env.close()
```

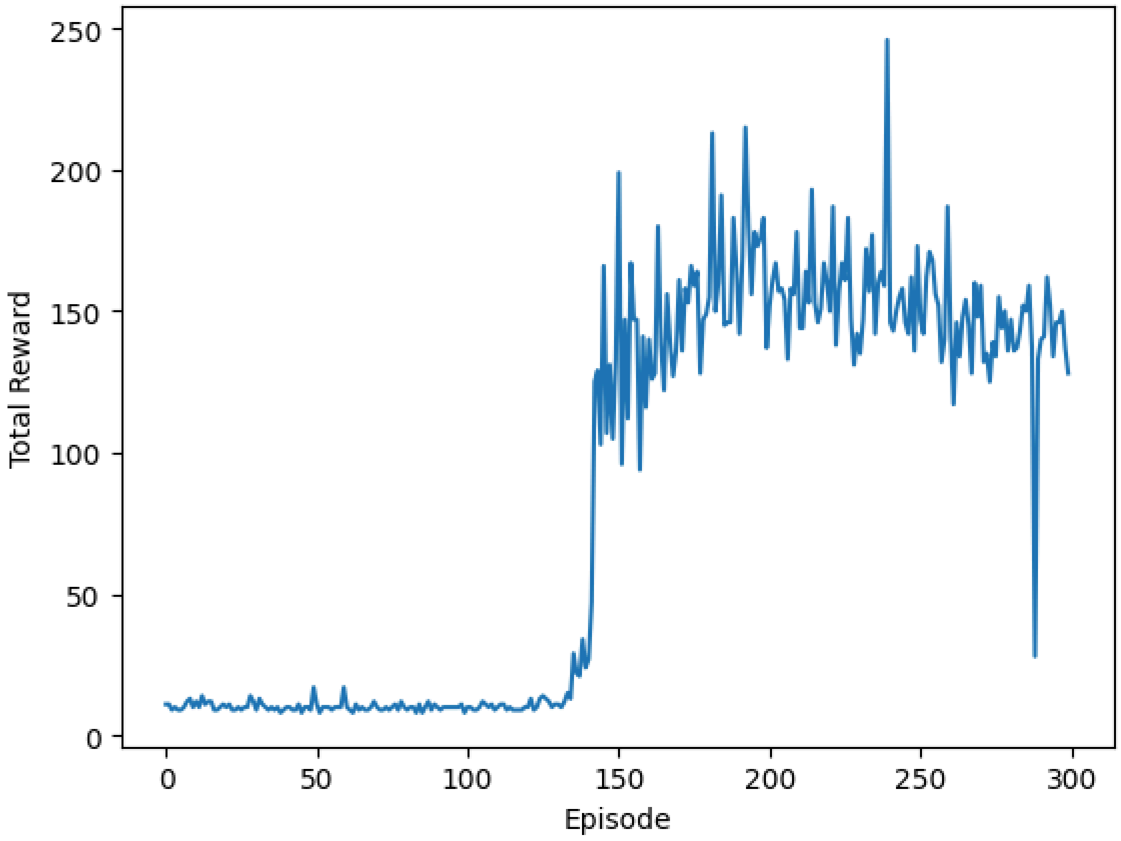
```
(32, 4)
(32,)
(32,)
(32, 4)
(32,)
```



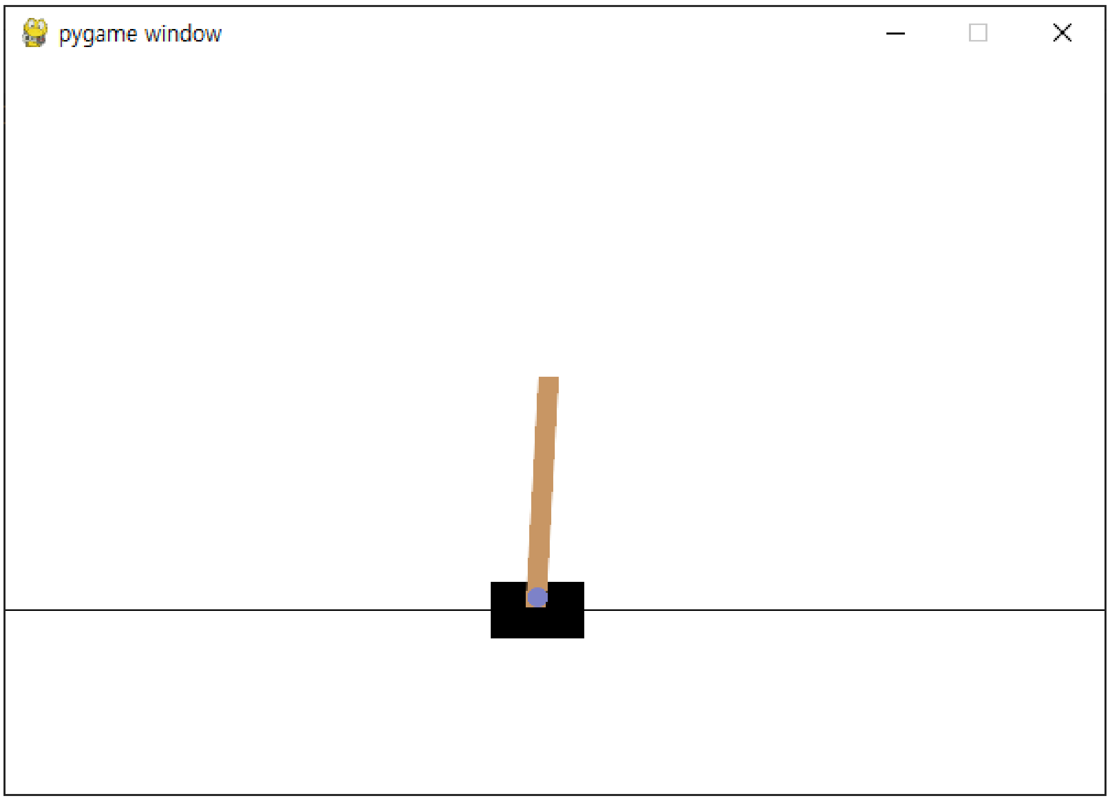
실습 2 dqn2.py

In [7]:
```python
from collections import deque
import random
import matplotlib.pyplot as plt
import numpy as np
import gym
from dezero import Model
from dezero import optimizers
import dezero.functions as F
import dezero.layers as L
import copy

class ReplayBuffer:
    def __init__(self, buffer_size, batch_size):
        self.buffer = deque(maxlen=buffer_size)
        self.batch_size = batch_size

    def add(self, state, action, reward, next_state, done):
        data = (state, action, reward, next_state, done)
        self.buffer.append(data)

    def __len__(self):
        return len(self.buffer)

    def get_batch(self):
        data = random.sample(self.buffer, self.batch_size)
        state = np.stack([x[0] for x in data])
        action = np.array([x[1] for x in data])
        reward = np.array([x[2] for x in data])
        next_state = np.stack([x[3] for x in data])
        done = np.array([x[4] for x in data]).astype(np.int32)
        return state, action, reward, next_state, done

class QNet(Model):
    def __init__(self, action_size):
        super().__init__()
        self.l1 = L.Linear(128)
        self.l2 = L.Linear(128)
        self.l3 = L.Linear(action_size)

    def forward(self, x):
        x = F.relu(self.l1(x))
        x = F.relu(self.l2(x))
        x = self.l3(x)
        return x

class DQNAgent:
    def __init__(self):
        self.gamma = 0.98
        self.lr = 0.0005
        self.epsilon = 0.1
        self.buffer_size = 10000
        self.batch_size = 32
        self.action_size = 2

        self.replay_buffer = ReplayBuffer(self.buffer_size, self.batch_size)
        self.qnet = QNet(self.action_size)
        self.qnet_target = QNet(self.action_size)
        self.optimizer = optimizers.Adam(self.lr)
        self.optimizer.setup(self.qnet)

    def get_action(self, state):
        if np.random.rand() < self.epsilon:
            return np.random.choice(self.action_size)
        else:
            state = state[np.newaxis, :]
            qs = self.qnet(state)
            return qs.data.argmax()

    def update(self, state, action, reward, next_state, done):
        self.replay_buffer.add(state, action, reward, next_state, done)
        if len(self.replay_buffer) < self.batch_size:
            return

        state, action, reward, next_state, done = self.replay_buffer.get_batch()
        qs = self.qnet(state)
        q = qs[np.arange(self.batch_size), action]

        next_qs = self.qnet_target(next_state)
        next_q = next_qs.max(axis=1)
        next_q.unchain()
        target = reward + (1 - done) * self.gamma * next_q

        loss = F.mean_squared_error(q, target)

        self.qnet.cleargrads()
        loss.backward()
        self.optimizer.update()

    def sync_qnet(self):
        self.qnet_target = copy.deepcopy(self.qnet)

episodes = 300
sync_interval = 20
env = gym.make('CartPole-v1', render_mode='rgb_array')
agent = DQNAgent()
reward_history = []

for episode in range(episodes):
    state_tuple = env.reset()
    if isinstance(state_tuple, tuple) and len(state_tuple) == 2 and isinstance(s
        state = state_tuple[0]
    else: # for older gym or unexpected format
        state = state_tuple

    done = False
    total_reward = 0

    while not done:
        action = agent.get_action(state)
        step_result = env.step(action)
        next_state, reward, terminated, truncated, info = step_result
        done = terminated or truncated

        agent.update(state, action, reward, next_state, done)
        state = next_state
        total_reward += reward

    if episode % sync_interval == 0:
        agent.sync_qnet()

    reward_history.append(total_reward)
    if episode % 10 == 0:
        print("episode :{}, total reward : {}".format(episode, total_reward))

plt.xlabel('Episode')
plt.ylabel('Total Reward')
plt.plot(range(len(reward_history)), reward_history)
plt.show()

env2 = gym.make('CartPole-v0', render_mode='human')
agent.epsilon = 0
state_tuple_eval = env2.reset()
if isinstance(state_tuple_eval, tuple) and len(state_tuple_eval) == 2 and isinst
    state = state_tuple_eval[0]
else: # for older gym or unexpected format
    state = state_tuple_eval

done = False
total_reward = 0

while not done:
    action = agent.get_action(state)
    step_result_eval = env2.step(action)
    next_state, reward, terminated, truncated, info = step_result_eval
    done = terminated or truncated
    state = next_state
    total_reward += reward
    render_result = env2.render() # Included as per image, behavior depends on g

print("Total Reward :", total_reward)
env.close()
env2.close()
```

```
episode :0, total reward : 11.0
episode :10, total reward : 12.0
episode :20, total reward : 10.0
episode :30, total reward : 9.0
episode :40, total reward : 10.0
episode :50, total reward : 11.0
episode :60, total reward : 10.0
episode :70, total reward : 10.0
episode :80, total reward : 9.0
episode :90, total reward : 10.0
episode :100, total reward : 10.0
episode :110, total reward : 10.0
episode :120, total reward : 10.0
episode :130, total reward : 11.0
episode :140, total reward : 27.0
episode :150, total reward : 199.0
episode :160, total reward : 140.0
episode :170, total reward : 161.0
episode :180, total reward : 155.0
episode :190, total reward : 142.0
episode :200, total reward : 152.0
episode :210, total reward : 144.0
episode :220, total reward : 150.0
episode :230, total reward : 135.0
episode :240, total reward : 146.0
episode :250, total reward : 147.0
episode :260, total reward : 143.0
episode :270, total reward : 159.0
episode :280, total reward : 147.0
episode :290, total reward : 140.0
```



```
Total Reward : 160.0
```



```
In [ ]:
```