```
import gym


gym.envs.register(
    id='FrozenLakeNotSlippery-v0',
    entry_point='gym.envs.toy_text:FrozenLakeEnv',
    kwargs={'map_name' : '4x4', 'is_slippery': False},
    max_episode_steps=100,
    reward_threshold=0.74
)


# Create the gridworld-like environment
env=gym.make('FrozenLakeNotSlippery-v0')
# Let's look at the model of the environment (i.e., P):
env.env.P
# Question: what is the data in this structure saying? Relate this to the course
# presentation of P
```

```
{0: {0: [(1.0, 0, 0.0, False)],
  1: [(1.0, 4, 0.0, False)],
  2: [(1.0, 1, 0.0, False)],
  3: [(1.0, 0, 0.0, False)]},
 1: {0: [(1.0, 0, 0.0, False)],
  1: [(1.0, 5, 0.0, True)],
  2: [(1.0, 2, 0.0, False)],
  3: [(1.0, 1, 0.0, False)]},
 2: {0: [(1.0, 1, 0.0, False)],
  1: [(1.0, 6, 0.0, False)],
  2: [(1.0, 3, 0.0, False)],
  3: [(1.0, 2, 0.0, False)]},
 3: {0: [(1.0, 2, 0.0, False)],
  1: [(1.0, 7, 0.0, True)],
  2: [(1.0, 3, 0.0, False)],
  3: [(1.0, 3, 0.0, False)]},
 4: {0: [(1.0, 4, 0.0, False)],
  1: [(1.0, 8, 0.0, False)],
  2: [(1.0, 5, 0.0, True)],
  3: [(1.0, 0, 0.0, False)]},
 5: {0: [(1.0, 5, 0, True)],
  1: [(1.0, 5, 0, True)],
  2: [(1.0, 5, 0, True)],
  3: [(1.0, 5, 0, True)]},
 6: {0: [(1.0, 5, 0.0, True)],
  1: [(1.0, 10, 0.0, False)],
  2: [(1.0, 7, 0.0, True)],
  3: [(1.0, 2, 0.0, False)]},
 7: {0: [(1.0, 7, 0, True)],
  1: [(1.0, 7, 0, True)],
  2: [(1.0, 7, 0, True)],
  3: [(1.0, 7, 0, True)]},
 8: {0: [(1.0, 8, 0.0, False)],
  1: [(1.0, 12, 0.0, True)],
  2: [(1.0, 9, 0.0, False)],
  3: [(1.0, 4, 0.0, False)]},
 9: {0: [(1.0, 8, 0.0, False)],
  1: [(1.0, 13, 0.0, False)],
  2: [(1.0, 10, 0.0, False)],
  3: [(1.0, 5, 0.0, True)]},
 10: {0: [(1.0, 9, 0.0, False)],
  1: [(1.0, 14, 0.0, False)],
  2: [(1.0, 11, 0.0, True)],
  3: [(1.0, 6, 0.0, False)]},
 11: {0: [(1.0, 11, 0, True)],
  1: [(1.0, 11, 0, True)],
  2: [(1.0, 11, 0, True)],
  3: [(1.0, 11, 0, True)]},
 12: {0: [(1.0, 12, 0, True)],
  1: [(1.0, 12, 0, True)],
  2: [(1.0, 12, 0, True)],
  3: [(1.0, 12, 0, True)]},
 13: {0: [(1.0, 12, 0.0, True)],
  1: [(1.0, 13, 0.0, False)],
  2: [(1.0, 14, 0.0, False)],
  3: [(1.0, 9, 0.0, False)]},
 14: {0: [(1.0, 13, 0.0, False)],
  1: [(1.0, 14, 0.0, False)],
  2: [(1.0, 15, 1.0, True)],
```

Question: What is the data in this structure saying? Relate this to the course presentation of P

First this is a 4x4 frozen lake environment, with a start point on the top left and end point on the bottom right, therefore the 16 keys in the first level dictionary are the 16 states respectively, and the 4 keys in the second level dictionary are the 4 possible actions can be taken at each state, that is:

0 = Left,

1 = Down,

2 = Right,

3 = Up

Therefore the data structure is basically saying we have 16 states, which is the state space S in the lecture, each state has 4 actions, which is the action space A in the lecture.

In the tuple of each action's value, the first element is the probability of reaching a specific next state and obtain a specific reward given the current state and action, which should be $p(s', r|s, a)$ from our lecture. As we can see this is always 1.0 in our case, that's because we set is_slippery to false.

Second element in the tuple is the next state s', third element is the reward r at next state, and last element is a boolean representation of whether the game is over (the game is over if we reach a state that is "G" as the goal or "H" as the hole).

To conclude, P is basically the complete model of environment in finite MDP.

```
# Now let's investigate the observation space (i.e., S using our nomenclature),
# and confirm we see it is a discrete space with 16 locations
print(env.observation_space)
```

```
Discrete(16)
```

```
stateSpaceSize = env.observation_space.n
print(stateSpaceSize)
```

```
16
```

```
# Now let's investigate the action space (i.e., A) for the agent->environment
# channel
print(env.action_space)
```

```
Discrete(4)
```

```
# The gym environment has ...sample() functions that allow us to sample
# from the above spaces:
for g in range(1,10,1):
  print("sample from S:",env.observation_space.sample()," ... ","sample from A:",env.action_s
```

```
sample from S: 0  ...  sample from A: 3
sample from S: 12  ...  sample from A: 1
sample from S: 2  ...  sample from A: 0
sample from S: 14  ...  sample from A: 0
sample from S: 5  ...  sample from A: 0
sample from S: 5  ...  sample from A: 3
sample from S: 10  ...  sample from A: 3
sample from S: 11  ...  sample from A: 2
sample from S: 1  ...  sample from A: 3
```

```
# The enviroment also provides a helper to render (visualize) the environment
env.reset()
env.render()
```

```
SFFF
FHFH
FFFH
HFFG
```

```
# We can act as the agent, by selecting actions and stepping the environment
# through time to see its responses to our actions
```

```
    env.reset()
    exitCommand=False
    while not(exitCommand):
      env.render()
      print("Enter the action as an integer from 0 to",env.action_space.n," (or exit): ")
      userInput=input()
      if userInput=="exit":
        break
      action=int(userInput)
      (observation, reward, compute, probability) = env.step(action)
      print("--> The result of taking action",action,"is:")
      print("     S=",observation)
      print("     R=",reward)
      print("     p=",probability)

      env.render()
```

```
  (Right)
SFFF
FHFH
FFFH
HFFG
  (Right)
SFFF
FHFH
FFFH
HFFG
Enter the action as an integer from 0 to 4  (or exit):
1
--> The result of taking action 1 is:
     S= 13
     R= 0.0
     p= {'prob': 1.0}
  (Down)
SFFF
FHFH
FFFH
HFFG
  (Down)
SFFF
FHFH
FFFH
HFFG
Enter the action as an integer from 0 to 4  (or exit):
2
--> The result of taking action 2 is:
     S= 14
     R= 0.0
     p= {'prob': 1.0}
  (Right)
SFFF
FHFH
FFFH
HFFG
  (Right)
```

```
SFFF
FHFH
FFFH
HFFG
Enter the action as an integer from 0 to 4  (or exit):
2
--> The result of taking action 2 is:
     S= 15
     R= 1.0
     p= {'prob': 1.0}
  (Right)
SFFF
FHFH

FFFH
HFFG
  (Right)
SFFF
FHFH
FFFH
HFFG
Enter the action as an integer from 0 to 4  (or exit):
```

```
# Question: draw a table indicating the correspondence between the action
# you input (a number) and the logic action performed.
# Question: draw a table that illustrates what the symbols on the render image
# mean?
# Question: Explain what the objective of the agent is in this environment?
```

Input Number	Action
0	Left
1	Down
2	Right
3	Up

Symbol	Meaning
S	Start point, safe
F	Frozen lake, safe
H	Hole, fall into the freezing water, game over
G	Goal, win, game over

Goal: Agent's goal is to reach goal position without falling into one of the holes.

```
# Practical: Code up an AI that will employ random action selection in order
# to drive the agent. Test this random action selection agent with the
# above environment (i.e., code up a loop as I did above, but instead
# of taking input from a human user, take it from the AI you coded).


env.reset()
```

```
  while not(end):
    env.render()
    action=env.action_space.sample()
    (observation, reward, end, probability) = env.step(action)
    print("--> The result of taking action",action,"is:")
    print("     S=",observation)
    print("     R=",reward)
    print("     p=",probability)
    env.render()
    if end:
      print(f'--> Game over, {"Agent reached goal" if observation==15 else \
                            "Agent fell in water"}')
```

```
SFFF
FHFH
FFFH
HFFG
--> The result of taking action 2 is:
     S= 1
     R= 0.0
     p= {'prob': 1.0}
  (Right)
SFFF
FHFH
FFFH
HFFG
  (Right)
SFFF
FHFH
FFFH
HFFG
--> The result of taking action 3 is:
     S= 1
     R= 0.0
     p= {'prob': 1.0}
  (Up)
SFFF
FHFH
FFFH
HFFG
  (Up)
SFFF
FHFH
FFFH
HFFG
--> The result of taking action 1 is:
     S= 5
     R= 0.0
     p= {'prob': 1.0}
  (Down)
SFFF
FHFH
FFFH
```

```
    HFFG
    --> Game over. Agent fell in water
```

```
# Now towards dynamic programming. Note that env.env.P has the model
# of the environment.
#
# Question: How would you represent the agent's policy function and value function?
# Practical: revise the above AI solver to use a policy function in which you
# code the random action selections in the policy function. Test this.
# Practical: Code the C-4 Policy Evaluation (Prediction) algorithm. You may use
# either the inplace or ping-pong buffer (as described in the lecture). Now
# randomly initialize your policy function, and compute its value function.
# Report your results: policy and value function. Ensure your prediction
# algo reports how many iterations it took.
#
# (Optional): Repeat the above for q.
#
# Policy Improvement:
# Question: How would you use P and your value function to improve an arbitrary
# policy, pi, per Chapter 4?
# Practical: Code the policy iteration process, and employ it to arrive at a
# policy that solves this problem. Show your testing results, and ensure
# it reports the number of iterations for each step: (a) overall policy
# iteration steps and (b) evaluation steps.
# Practical: Code the value iteration process, and employ it to arrive at a
# policy that solves this problem. Show your testing results, reporting
# the iteration counts.
# Comment on the difference between the iterations required for policy vs
# value iteration.
#
# Optional: instead of the above environment, use the "slippery" Frozen Lake via
# env = gym.make("FrozenLake-v0")
```

Question: How would you represent the agent's policy function and value function?

1. policy table pi(s): a dictionary that maps state to action
2. value function v(s): a dictionary that maps state to a value

```
# Practical: revise the above AI solver to use a policy function in which you
# code the random action selections in the policy function. Test this.

# This is global variable of how env.env.p[state] for all states is structured
PROBABILITY = 0
NEXT_STATE = 1
REWARD = 2
DONE = 3
```

```
def initialize_policy_function(env=env):
    import random
    return {s: random.randint(0, env.action_space.n - 1)
            for s in range(env.observation_space.n)}

def initialize_value_function(env=env, random=True):
  if not random:
    return {s: 0 if s != 15 else 1
           for s in range(env.observation_space.n)}
  else:
    import random
    return {s: random.random()
          for s in range(env.observation_space.n)}

def next_state(s, a, env=env.env.P):
  return env[s][a][0][NEXT_STATE]

def next_state_reward(s, a, env=env.env.P):
  return env[s][a][0][REWARD]


# Practical: Code the C-4 Policy Evaluation (Prediction) algorithm. You may use
# either the inplace or ping-pong buffer (as described in the lecture).

def policy_evaluation(pi,
                      v,
                      env = env.env.P,
                      gamma=0.9,
                      threshold=0.01):
  '''
  Returns the re-evaluated state value table.

  Parameters:
      pi (dict): maps state to action
      v (dict): maps state to value
      env (dict{dict}): environment details
      gamma (float): update rate
      threshold (float): evaluation stops when change of value table is smaller
                          than this threshold

  Returns:
      v (dict): updated value table
  '''
  ite = 0
  while True:
    ite += 1
    delta = 0
    for s in env.keys():
      value = v[s]
      a = pi[s]
      v[s] = next_state_reward(s, a, env) + gamma * v[next_state(s, a, env)]
      delta = max(delta, abs(value - v[s]))
    if delta < threshold:
      print(f"Took {ite} iterations to evaluate policy")
      return v


# Now randomly initialize your policy function, and compute its value function.
# Report your results: policy and value function. Ensure your prediction
# algo reports how many iterations it took.
pi = initialize_policy_function()
v = initialize_value_function()
policy_evaluation(pi, v)
```

```
     Took 23 iterations to evaluate policy
     {0: 0.002387411491748006,
      1: 0.04418139627823871,
      2: 0.012799008179449432,
      3: 0.0026981953782613853,
      4: 0.0021486703425732056,
      5: 0.08130058654091034,
      6: 0.017511931608000728,
      7: 0.016388004031956293,
      8: 0.03892307205914997,
      9: 0.019457701786667475,
      10: 0.017511931608000728,
      11: 0.07648011181920712,
      12: 0.024930007909316952,
      13: 0.019457701786667475,
      14: 0.019457701786667475,
      15: 0.06190190128293902}
```

Question: How would you use P and your value function to improve an arbitrary policy, π, per Chapter 4?

Answer:

1. Initialize pi and v, both randomly
2. Run policy iteration, until we find a policy that avoid agent from falling into freezing water, at this point we have our policy that is at least no worse than optimal policy, then we can let the agent use this policy to play the game

```
# Policy improvement function
def policy_improvement(pi,
                      v,
                      env = env.env.P,
                      gamma=0.9):
  '''
  Returns the improved policy table.
```

```
    Parameters:
        pi (dict): maps state to action
        v (dict): maps state to value
        env (dict{dict}): environment details
        gamma (float): update rate

    Returns:
        pi (dict): updated policy table
    '''
    ite = 0
    while True:
      ite += 1
      policy_stable = True
      for s in env.keys():
        a = pi[s]
        # Map each action to a value that is the value of next state so we can
        # find argmax a that maximize value(next_state)
        look_up_table = {a_: next_state_reward(s, a_, env) +
                             gamma * v[next_state(s, a_, env)]
                             for a_ in env[s].keys()}
        a_ = max(look_up_table, key=look_up_table.get)
        # instead of check a_ == a I modified it to check if old action results
        # in same next state value as new action because two actions may have same
        # next state value, use this condition we can avoid meaningless swap
        if look_up_table[a_] > look_up_table[a]:
          pi[s] = a_
          policy_stable = False
      if policy_stable:
        print(f"Took {ite} iterations to improve policy")
        return pi


  # Practical: Code the policy iteration process, and employ it to arrive at a
  # policy that solves this problem. Show your testing results, and ensure
  # it reports the number of iterations for each step: (a) overall policy
  # iteration steps and (b) evaluation steps.

  def run_agent(pi, render=False):
    '''
    Returns if current policy function let the agent reaches goal

    Parameters:
       pi (dict): policy table
       render (bool): whether to show agent's steps, can be disabled when training

    Returns:
       reached_goal (bool): returns true if agent reaches goal, false if agent
                            fall into hole
    '''
    env.reset()
    if render:
        env.render()
    end=False
    s = 0
    while not(end):
      action=pi[s]
      (observation, reward, end, probability) = env.step(action)
      s = observation
      if render:
        env.render()
      if end:
        win = observation == 15
        if render:
          print(f"-- Agent {'reached goal' if win else 'fell into a hole'} --")
        return win

  # Practical: Code the policy iteration process, and employ it to arrive at a
  # policy that solves this problem.
  def policy_iteration():
    '''
    Returns optimal policy function and value function

    Returns:
       pi (dict): optimal policy function
       v (dict): optimal value function
    '''
    pi = initialize_policy_function()
    v = initialize_value_function()
    ite = 0
    while not run_agent(pi):
      print(f"--- Iteration {ite} ---")
      ite += 1
      policy_evaluation(pi, v)
      policy_improvement(pi, v)
    print(f"--- Done, took {ite} iterations to find optimal policy ---")
    return pi, v


  # Show your testing results, and ensure it reports the number of iterations for
  # each step: (a) overall policy iteration steps and (b) evaluation steps.
  pi, v = policy_iteration()
```

```
     --- Iteration 0 ---
     Took 23 iterations to evaluate policy
     Took 2 iterations to improve policy
     --- Iteration 1 ---
     Took 2 iterations to evaluate policy
     Took 2 iterations to improve policy
     --- Iteration 2 ---
     Took 3 iterations to evaluate policy
     Took 2 iterations to improve policy
     --- Iteration 3 ---
     Took 2 iterations to evaluate policy
     Took 2 iterations to improve policy
     --- Done, took 4 iterations to find optimal policy ---
```

```
# Show optimal policy
run_agent(pi, render=True)
```

```
SFFF
FHFH
FFFH
HFFG
  (Right)
SFFF
FHFH
FFFH
HFFG
  (Right)
SFFF
FHFH
FFFH
HFFG
  (Down)
SFFF
FHFH
FFFH
HFFG
  (Down)
SFFF
FHFH
FFFH
HFFG
  (Down)
SFFF
FHFH
FFFH
HFFG
  (Right)
SFFF
FHFH
FFFH
HFFG
-- Agent reached goal --
True
```

Optional: Repeat for q

q(s,a): A dictionary of dictionary, first level of dictionary maps state to possible actions, and second level of dictionary maps action to value

```
def initialize_q():
  import random
  return {s: {a: random.random() for a in range(env.action_space.n)}
          for s in range(env.observation_space.n)}
```

```
initialize_q()


# use q instead
# Policy improvement function
def policy_improvement(q, v, env = env.env.P, gamma=0.9):
  '''
  Returns the improved policy table.

  Parameters:
      pi (dict): maps state to action
      v (dict): maps state to value
      env (dict{dict}): environment details
      gamma (float): update rate

  Returns:
      pi (dict): updated policy table
  '''
  ite = 0
  while True:
    ite += 1
    policy_stable = True
    for s in env.keys():
      a = policy_function(s, pi)
      # Map each action to a value that is the value of next state so we can
      # find argmax a that maximize value(next_state)
      look_up_table = {a_: next_state_reward(s, a_, env) +
                       gamma * v[next_state(s, a_, env)]
                       for a_ in env[s].keys()}
      a_ = max(look_up_table, key=look_up_table.get)
      # instead of check a_ == a I modified it to check if old action results
      # in same next state value as new action because two actions may have same
      # next state value, use this condition we can avoid meaningless swap
      if look_up_table[a_] > look_up_table[a]:
        pi[s] = a_
        policy_stable = False
    if policy_stable:
      print(f"Took {ite} iterations to improve policy")
      return pi


# Practical: Code the value iteration process, and employ it to arrive at a
# policy that solves this problem. Show your testing results, reporting
# the iteration counts.
# Comment on the difference between the iterations required for policy vs
# value iteration.
#
import random
v = {s: random.random() if s != 15 else 0 for s in
     range(env.observation_space.n)}
def value_iteration(v, env=env.env.P, theta=0.01, gamma=0.9):
  ite = 0
  # Value Iteration, update value table until diff is smaller than threshold
  # Value iteration, update value table until diff is smaller than threshold
  while True:
    ite += 1
    delta = 0
    for s in env.keys():
      value = v[s]
      lookup_table = {a: next_state_reward(s, a, env) + gamma *
                      v[next_state(s, a, env)] for a in env[s].keys()}
      v[s] = max(lookup_table.values())
      delta = max(delta, abs(value - v[s]))
    if delta < theta:
      break
  print(f'Value iteration done after {ite} iterations.')

  # Create policy based on current value table
  pi = {}
  for s in env.keys():
    lookup_table = {a: next_state_reward(s, a, env) + gamma *
                      v[next_state(s, a, env)] for a in env[s].keys()}
    best_action = max(lookup_table, key=lookup_table.get)
    pi[s] = best_action
  return pi

pi = value_iteration(v)
```

```
Value iteration done after 23 iterations.
```

```
run_agent(pi, render=True)
```

```
SFFF
FHFH
FFFH
HFFG
  (Down)
SFFF
FHFH
FFFH
HFFG
  (Down)
SFFF
FHFH
FFFH
HFFG
  (Right)
SFFF
FHFH
FFFH
HFFG
  (Down)
SFFF
FHFH
FFFH
HFFG
```

```
  (Right)
SFFF
FHFH
FFFH
HFFG
  (Right)
SFFF
FHFH
FFFH
HFFG
-- Agent reached goal --
True
```

Comment on the difference between the iterations required for policy vs value iteration.

It took 30 iterations for policy iteration to converge but took 23 iterations for value iteration to converge, therefore it's better to use value iteration because it merges faster.

```
# Optional: instead of the above environment, use the "slippery" Frozen Lake via
# env = gym.make("FrozenLake-v0")
env = gym.make("FrozenLake-v0")

PROBABILITY = 0
NEXT_STATE = 1
REWARD = 2
DONE = 3



import random

v = {s: random.random() if s != 15 else 0 for s in
     range(env.observation_space.n)}

def value_iteration(v, P=env.env.P, theta=0.01, gamma=0.3):
  ite = 0
  # Value Iteration, update value table until diff is smaller than threshold
  while True:
    ite += 1
    delta = 0
    lookup_table = {}
    for s in P.keys():
      value = v[s]
      k = [sum([ s_[PROBABILITY] * (s_[REWARD] + gamma * s_[NEXT_STATE]) for s_ in P[s][a]])
      v[s] = max([sum([ s_[PROBABILITY] * (s_[REWARD] + gamma * s_[NEXT_STATE]) for s_ in P[s
      delta = max(delta, abs(value - v[s]))
    if delta < theta:
      break
    print(f'Value iteration done after {ite} iterations.')

    # Create policy based on current value table
    pi = {}
    for s in P.keys():
      lookup_table = {a: sum([ s_[PROBABILITY] * (s_[REWARD] + gamma * s_[NEXT_STATE]) for s_ i
                      for a in P[s].keys()}
      best_action = max(lookup_table, key=lookup_table.get)
      pi[s] = best_action
    return pi

pi = value_iteration(v)
```

```
    Value iteration done after 2 iterations.
```

```
run = True
while run:
  run = not run_agent(pi, render=True)
```

```
    HFFG
      (Right)
    SFFF
    FHFH
    FFFH
    HFFG
      (Right)
    SFFF
    FHFH
    FFFH
    HFFG
      (Right)
    SFFF

    FHFH
    FFFH
    HFFG
    -- Agent fell into a hole --

    SFFF
    FHFH
    FFFH
    HFFG
      (Down)
    SFFF
    FHFH
    FFFH
    HFFG
      (Down)
    SFFF
    FHFH
    FFFH
    HFFG
      (Down)
    SFFF
    FHFH
```

```
FFFH
HFFG
  (Down)
SFFF
FHFH
FFFH
HFFG
  (Down)
SFFF
FHFH
FFFH
HFFG
  (Down)
SFFF
FHFH
FFFH
HFFG
  (Down)
SFFF
FHFH
FFFH
HFFG
  (Down)
SFFF
```