```
import gym
import matplotlib.pyplot as plt
from IPython import display as ipythondisplay
```

```
pip install gym pyvirtualdisplay
```

```
/usr/local/lib/python3.10/dist-packages/ipykernel/ipkernel.py:283: DeprecationWarning: `should_run_async` will not call `transform_cell`
  and should_run_async(code)
Looking in indexes: https://pypi.org/simple, https://us-python.pkg.dev/colab-wheels/public/simple/
Requirement already satisfied: gym in /usr/local/lib/python3.10/dist-packages (0.25.2)
Collecting pyvirtualdisplay
  Downloading PyVirtualDisplay-3.0-py3-none-any.whl (15 kB)
Requirement already satisfied: numpy>=1.18.0 in /usr/local/lib/python3.10/dist-packages (from gym) (1.22.4)
Requirement already satisfied: cloudpickle>=1.2.0 in /usr/local/lib/python3.10/dist-packages (from gym) (2.2.1)
Requirement already satisfied: gym-notices>=0.0.4 in /usr/local/lib/python3.10/dist-packages (from gym) (0.0.8)
Installing collected packages: pyvirtualdisplay
Successfully installed pyvirtualdisplay-3.0
```

```
!apt-get install -y xvfb python-opengl > /dev/null 2>&1
```

```
from pyvirtualdisplay import Display
display = Display(visible=0, size=(400, 300))
display.start()
```

```
<pyvirtualdisplay.display.Display at 0x7fe39e7c7a00>
```

```
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
)
```

```
/usr/local/lib/python3.10/dist-packages/ipykernel/ipkernel.py:283: DeprecationWarning: `should_run_async` will not call `transform_cell`
  and should_run_async(code)
```

```
# 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
```

```
  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, 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)],
 3: [(1.0, 10, 0.0, False)]},
15: {0: [(1.0, 15, 0, True)],
 1: [(1.0, 15, 0, True)],
 2: [(1.0, 15, 0, True)],
 3: [(1.0, 15, 0, True)]}}
```

```
# 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)
```

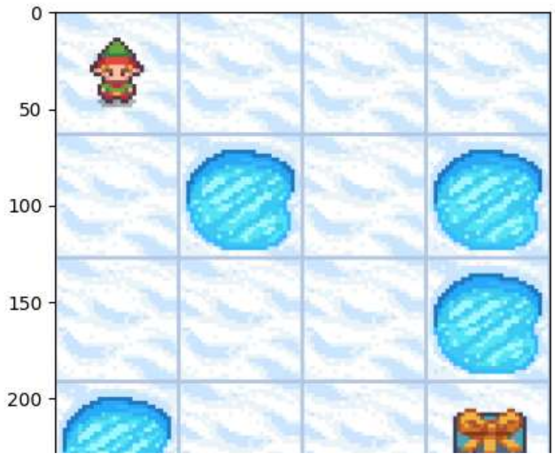
```
# The gym environment has ...sample() functions that allow us to sample
# from the above spaces:
for g in range(1,10,1):
  print("sample from S:",env.observation_space.sample()," ... ","sample from A:",env.action_space.sample())
```

```
sample from S: 14  ...  sample from A: 3
sample from S: 3  ...  sample from A: 1
sample from S: 2  ...  sample from A: 3
sample from S: 15  ...  sample from A: 1
sample from S: 9  ...  sample from A: 1
sample from S: 3  ...  sample from A: 3
sample from S: 2  ...  sample from A: 3
sample from S: 11  ...  sample from A: 0
sample from S: 7  ...  sample from A: 1
```

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

env.reset()
prev_screen = env.render(mode='rgb_array')
plt.imshow(prev_screen)
```

```
/usr/local/lib/python3.10/dist-packages/gym/core.py:43: DeprecationWarning: WARN: The argument mode in
See here for more information: https://www.gymlibrary.ml/content/api/
  deprecation(
<matplotlib.image.AxesImage at 0x7fe39e6da380>
```




```
print(env.action_space.sample)
```

```
<bound method Discrete.sample of Discrete(4)>
```

```
# 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):
  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)
  screen = env.render(mode='rgb_array')
  plt.imshow(screen)
```

```
Enter the action as an integer from 0 to 4  (or exit):
0
/usr/local/lib/python3.10/dist-packages/gym/core.py:43: DeprecationWarning: WARN: The argument mode
See here for more information: https://www.gymlibrary.ml/content/api/
  deprecation(
--> The result of taking action 0 is:
     S= 0
     R= 0.0
     p= {'prob': 1.0}
Enter the action as an integer from 0 to 4  (or exit):
1
--> The result of taking action 1 is:
     S= 4
     R= 0.0
     p= {'prob': 1.0}
Enter the action as an integer from 0 to 4  (or exit):
2
--> The result of taking action 2 is:
     S= 5
     R= 0.0
     p= {'prob': 1.0, 'TimeLimit.truncated': False}
Enter the action as an integer from 0 to 4  (or exit):
3
--> The result of taking action 3 is:
     S= 5
     R= 0
     p= {'prob': 1.0, 'TimeLimit.truncated': False}
Enter the action as an integer from 0 to 4  (or exit):
4
---------------------------------------------------------------------------
KeyError                                  Traceback (most recent call last)
<ipython-input-23-f35ee03cb1e0> in <cell line: 5>()
      9     break
     10   action=int(userInput)
---> 11   (observation, reward, compute, probability) = env.step(action)
     12   print("--> The result of taking action",action,"is:")
     13   print("     S=",observation)

4 frames

/usr/local/lib/python3.10/dist-packages/gym/envs/toy_text/frozen_lake.py in step(self, a)
    245
```

```
# 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()
exitCommand=False
import random
while not(exitCommand):
  action=random.randint(0,env.action_space.n-1)
  (observation, reward, done, info) = env.step(action)
  print("--> The result of taking action",action,"is:")
  print("     S=",observation)
  print("     R=",reward)
  print("     done=",done)
  print("     info=",info)

  screen = env.render(mode='rgb_array')
  plt.imshow(screen)
  if done:
    exitCommand=True
```

```
--> The result of taking action 2 is:
     S= 1
     R= 0.0
     done= False
     info= {'prob': 1.0}
--> The result of taking action 0 is:
     S= 0
     R= 0.0
     done= False
     info= {'prob': 1.0}
--> The result of taking action 0 is:
     S= 0
     R= 0.0
     done= False
     info= {'prob': 1.0}
--> The result of taking action 0 is:
     S= 0
     R= 0.0
     done= False
     info= {'prob': 1.0}
--> The result of taking action 2 is:
     S= 1
     R= 0.0
     done= False
     info= {'prob': 1.0}
--> The result of taking action 3 is:
     S= 1
     R= 0.0
     done= False
     info= {'prob': 1.0}
--> The result of taking action 0 is:
     S= 0
     R= 0.0
     done= False
     info= {'prob': 1.0}
--> The result of taking action 2 is:
     S= 1
     R= 0.0
     done= False
     info= {'prob': 1.0}
--> The result of taking action 0 is:
     S= 0
     R= 0.0
     done= False
     info= {'prob': 1.0}
--> The result of taking action 2 is:
     S= 1
     R= 0.0
     done= False
     info= {'prob': 1.0}
--> The result of taking action 2 is:
     S= 2
     R= 0.0
     done= False
     info= {'prob': 1.0}
--> The result of taking action 0 is:
     S= 1
     R= 0.0
     done= False
     info= {'prob': 1.0}
--> The result of taking action 1 is:
     S= 5
     R= 0.0
     done= True
     info= {'prob': 1.0, 'TimeLimit.truncated': False}
```


```
# 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.
env.reset()
exitCommand=False
import random
def random_policy(observation):
    num_actions = env.action_space.n
    action = random.randint(0, num_actions - 1)
```

```
      action = random.randint(0, num_actions - 1)
      return action
  policy={}
  while not(exitCommand):
    action=random.randint(0,env.action_space.n-1)
    (observation, reward, done, info) = env.step(action)
    print("--> The result of taking action",action,"is:")
    print("     S=",observation)
    print("     R=",reward)
    print("     done=",done)
    print("     info=",info)
    action = random_policy(env.observation_space)

    observation, reward, done, info = env.step(action)

    # Store the selected action in the policy dictionary
    policy[observation] = action

    screen = env.render(mode='rgb_array')
    plt.imshow(screen)
    if done:
      exitCommand=True
  print("Final Policy:")
  print(policy)
```

```
--> The result of taking action 3 is:
```

```
# 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.
#import numpy as np
import numpy as np
nS=env.observation_space.n
nA=env.action_space.n
def c4_policy_evaluation(env, policy, gamma=0.9, theta=1e-8, max_iterations=1000):
    V = np.zeros(nS)  # Initialize value function V with zeros
    delta = np.zeros(nS)
    iterations = 0
    for _ in range(max_iterations):
        for s in range(nS):
            v = V[s]
            new_v = 0
            for a, action_prob in enumerate(policy[s]):
                for prob, next_state, reward, done in env.P[s][a]:
                    new_v += action_prob * (reward + gamma * V[next_state])
            V[s] = new_v
            delta[s] = np.abs(v - V[s])
            iterations+=1
        if np.max(delta) < theta:
            break
    return V,iterations

env.reset()
# Randomly initialize the policy function
random_policy = np.random.rand(nS, nA)
random_policy /= np.sum(random_policy, axis=1, keepdims=True)  # Normalize to get valid probabilities

# Compute the value function using C-4 Policy Evaluation
value_function, num_iterations = c4_policy_evaluation(env, random_policy)

print("Random Policy:")
print(random_policy)
print("\nValue function:")
print(value_function)
print("\nNumber of iterations:", num_iterations)
```

```
Random Policy:
[[0.31164474 0.50402386 0.08228162 0.10204977]
 [0.02957195 0.37551012 0.17729286 0.41762506]
 [0.35009591 0.14646156 0.05166569 0.45177684]
 [0.17454711 0.43370204 0.23394326 0.15780759]
 [0.30230288 0.37468976 0.01867339 0.30433397]
 [0.11515957 0.4718268  0.29047828 0.12253535]
 [0.28593586 0.32445296 0.23075314 0.15885805]
 [0.1801543  0.44831985 0.04936031 0.32216554]
 [0.05500314 0.54812129 0.34729915 0.04957642]
 [0.14614144 0.27497607 0.22219818 0.3566843 ]
 [0.32167919 0.19331414 0.34926326 0.13574341]
 [0.28951248 0.33716078 0.09743714 0.27588961]
 [0.03568991 0.27800716 0.37018548 0.31611746]
 [0.29393116 0.38703277 0.11111704 0.20791903]
 [0.25940017 0.04583775 0.36730092 0.32746115]
 [0.49758372 0.17900747 0.22329459 0.10011422]]

Value function:
[0.00636526 0.0021344  0.00728706 0.00176815 0.00845917 0.
 0.02707887 0.         0.01308998 0.03859801 0.08916556 0.
 0.         0.07695674 0.42925413 0.        ]

Number of iterations: 544
```

```
# (Optional): Repeat the above for q.

import numpy as np
nS=env.observation_space.n
nA=env.action_space.n
def Q_c4_policy_evaluation(env, policy, gamma=0.9, theta=1e-8, max_iterations=1000):
    Q = np.zeros(nS,nA)  # Initialize value function V with zeros
    delta = np.zeros(nS)
    iterations = 0
    for _ in range(max_iterations):
```

```
        for s in range(nS):
            for a in range(nA):
                q = Q[s][a]
                new_q = 0
                for prob, next_state, reward, done in env.P[s][a]:
                  new_q += prob * (reward + gamma * np.sum(policy[next_state] * Q[next_state]))
                  Q[s][a] = new_q
                  delta = max(delta, abs(q - Q[s][a]))
        iterations+=1
        if np.max(delta) < theta:
            break
    return Q,iterations

env.reset()
# Randomly initialize the policy function
random_policy = np.random.rand(nS, nA)
random_policy /= np.sum(random_policy, axis=1, keepdims=True)  # Normalize to get valid probabilities

# Compute the value function using C-4 Policy Evaluation
value_function, num_iterations = c4_policy_evaluation(env, random_policy)
print("I am writing the answer for optional question, not sure if it is correct")
print("Random Policy:")
print(random_policy)
print("\nValue function:")
print(value_function)
print("\nNumber of iterations:", num_iterations)
# 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")
```

```
I am writing the answer for optional question, not sure if it is correct
Random Policy:
[[0.29883886 0.35858972 0.02030124 0.32227017]
 [0.27855813 0.31389356 0.24973409 0.15781422]
 [0.49068198 0.07249161 0.33198918 0.10483723]
 [0.09760603 0.28214692 0.29270977 0.32753728]
 [0.18429816 0.32896122 0.1158044  0.37093622]
 [0.05911513 0.33957763 0.43615259 0.16515465]
 [0.2168417  0.20361778 0.23383899 0.34570153]
 [0.2354272  0.22575522 0.22611231 0.31270528]
 [0.06419204 0.3982263  0.1044225  0.43315915]
 [0.28912988 0.13934815 0.41374935 0.15777262]
 [0.26611786 0.26876368 0.24593291 0.21918555]
 [0.18984937 0.35342981 0.32519173 0.13152908]
 [0.15700729 0.48539175 0.34163076 0.0159702 ]
 [0.32073347 0.09779396 0.29902835 0.28244423]
 [0.21830621 0.26060958 0.20124245 0.31984175]
 [0.35408295 0.34548555 0.04038443 0.26004707]]

Value function:
[0.00243035 0.001244   0.00203781 0.00040521 0.0032506  0.
 0.01801119 0.         0.00641775 0.05085912 0.09482453 0.
 0.         0.11066511 0.32697232 0.        ]

Number of iterations: 736
```