```
import itertools
import numpy as np
import pandas as pd
import random
```

```
# States of a gridworld
N = 5
M = 5

# state space
state_space = list(itertools.product(range(N), range(M)))

# action space
action_space = [(0,1), (0,-1), (1,0), (-1,0)]
```

```
list(itertools.product(range(N), range(M)))
```

```
[(0, 0),
 (0, 1),
 (0, 2),
 (0, 3),
 (0, 4),
 (1, 0),
 (1, 1),
 (1, 2),
 (1, 3),
 (1, 4),
 (2, 0),
 (2, 1),
 (2, 2),
 (2, 3),
 (2, 4),
 (3, 0),
 (3, 1),
 (3, 2),
 (3, 3),
 (3, 4),
 (4, 0),
 (4, 1),
 (4, 2),
 (4, 3),
 (4, 4)]
```

Rules

	0	1	2	3	4
0					
1					
2					
3					
4					END

Terminal state = (4,4)

- Cannot go outside the grid.
- Once the teminal state is reached, the episode ends.
- Rewards
  - Transition to terminal state gives one step reward of +10 and every other transition gives a reward of -1.

```
terminal_states = [(4,4)]

def transition_probability(start_state, action, end_state):
  if start_state in terminal_states:
    return 0

  expected_state = tuple(np.array(start_state) + np.array(action))
  if expected_state == end_state:
    return 1

  if expected_state not in state_space and start_state == end_state:
    return 1

  return 0

def reward(start_state, action, end_state):
  if end_state in terminal_states:
    return 10
  else:
    return -1

def random_policy(start_state, action):
  if action in [(1,0), (0,1)]:
    return 0.5
  else:
    return 0
```

```
transition_probability((0,0), (-1,0), (0,0))
```

```
1
```

Policy evaluation

```
gamma = 1

v = dict(zip(state_space, np.zeros(N*M)))

iter = 0
while iter< 1000:
  for s in state_space:
    term1 = 0
    for a in action_space:
      term2 = 0
      for s_prime in state_space:
        term2+= transition_probability(s, a, s_prime) * (reward(s, a, s_prime) + gamma*v[s_prime])
      term1 += random_policy(s, a) * term2
    v[s] = term1.round(3)
  iter+=1
```

```
np.array(list(v.values())).reshape(N,M)
```

```
array([[0.811, 1.811, 2.498, 2.873, 2.998],
       [1.811, 3.124, 4.124, 4.748, 4.998],
       [2.498, 4.124, 5.499, 6.499, 6.999],
       [2.873, 4.748, 6.499, 8.   , 9.   ],
       [2.998, 4.998, 6.999, 9.   , 0.   ]])
```

```
# Random policy

# random among 4 actions

'''
array([[-95.785, -93.786, -90.348, -86.592, -84.048],
       [-93.786, -91.227, -86.668, -81.382, -77.505],
       [-90.348, -86.668, -79.716, -70.764, -63.085],
       [-86.592, -81.382, -70.764, -54.875, -36.986],
       [-84.048, -77.505, -63.085, -36.986,   0.   ]])
'''

# random among 2 actions - down or right

'''
array([[0.811, 1.811, 2.498, 2.873, 2.998],
       [1.811, 3.124, 4.124, 4.748, 4.998],
       [2.498, 4.124, 5.499, 6.499, 6.999],
       [2.873, 4.748, 6.499, 8.   , 9.   ],
       [2.998, 4.998, 6.999, 9.   , 0.   ]])
'''
```

```
'\narray([[0.811, 1.811, 2.498, 2.873, 2.998],\n       [1.811, 3.124, 4.124, 4.748, 4.998],\n       [2.498, 4.124, 5.499, 6.499,
6.999],\n       [2.873, 4.748, 6.499, 8.   , 9.   ],\n       [2.998, 4.998, 6.999, 9.   , 0.   ]])\n'
```

Policy improvement

Value iteration

```
gamma = 1

v = dict(zip(state_space, np.zeros(N*M)))

iter = 0
while iter< 1000:
  for s in state_space:
    max = -np.inf
    for a in action_space:
      term2 = 0
      for s_prime in state_space:
        term2+= transition_probability(s, a, s_prime) * (reward(s, a, s_prime) + gamma*v[s_prime])
      if term2 > max:
        max = term2
    v[s] = max
  iter+=1

np.array(list(v.values())).reshape(N,M)
```

```
array([[ 3.,  4.,  5.,  6.,  7.],
       [ 4.,  5.,  6.,  7.,  8.],
       [ 5.,  6.,  7.,  8.,  9.],
       [ 6.,  7.,  8.,  9., 10.],
       [ 7.,  8.,  9., 10.,  0.]])
```

```
#Hi AIML2023, Here is a task for the next lab session.
#Extend the notebook discussed today to come up with optimal state values through value iteration method for the follwing cases:
# Case 1. States (1,2), (1,3) are dummy states. Meaning the agent cannot transition to these states. Modify the transition probability i
#required and come up with optimal state values by policy improvemet (value iteration) for each of the above cases.

def transition_probability1(start_state, action, end_state):
    if start_state in terminal_states or start_state in [(1,2), (1,3)]:
        return 0

    expected_state = tuple(np.array(start_state) + np.array(action))
    if expected_state == end_state:
        return 1

    if expected_state not in state_space and start_state == end_state:
        return 1

    return 0

def reward(start_state, action, end_state):
    if end_state in terminal_states:
        return 10
    elif start_state in [(1,2), (1,3)]:
        return 0
    else:
        return -1
```

```
terminal_states = [(4,4)]
transition_probability1((0,0), (0,-1), (0,0))
```

```
1
```

```
#Case 2. States (1,2), (1,3) are damping states, they slow down the agent if it reaches these states.
#Assign a one-step-reward of "-5" for transition from damping states.

def transition_probability2(start_state, action, end_state):
    if start_state in terminal_states or start_state in [(1,2), (1,3)]:
        return 0

    expected_state = tuple(np.array(start_state) + np.array(action))
    if expected_state == end_state:
        return 1


terminal_states = [(5,4)]
transition_probability2((0,0), (0,-1), (0,0))
```

```
1
```

```
#Case 3. States (1,2), (1, 3) are holes.
#Meaning the agent cannot come out of these states and episode ends if the agent steps into holes.
def transition_probability3(start_state, action, end_state):
    if start_state in terminal_states or start_state in [(1,2), (1,3)]:
        return 0

    expected_state = tuple(np.array(start_state) + np.array(action))
    if expected_state == end_state:
        return 1

    if expected_state not in state_space and start_state == end_state:
        return 1

    return 0

def reward(start_state, action, end_state):
    if end_state in terminal_states:
        return 10
    elif start_state in [(1,2), (1,3)]:
        return -10  # Assign a negative reward for stepping into holes
    else:
        return -1
```

```
terminal_states = [(4,4)]
transition_probability3((0,0), (0,-1), (0,0))
```

```
1
```