```python
import project_utils
import robot_class


def main():
    # Main function
    # Edit number of episodes, 'random' or 'path' protocol, and layout 'a' or 'b'
    robot = robot_class.Robot(episodes=1000, protocol='random', layout='a')
    # Initialize robot

    robot.start_procedure()
    # Run episodes

    project_utils.report_printout(robot)
    # Print results


if __name__ == '__main__':
    # Python best practices
    main()

```

```python
import random
import numpy as np


def order_gen(warehouse):
    if warehouse == 'a':
        orders = ['A', 'B', 'C', 'D', 'E', 'F', 'G', 'H', 'I', 'J']
    else:
        orders = ['A', 'B', 'C', 'D', 'E', 'F', 'G', 'H',
                  'I', 'J', 'K', 'M', 'N', 'O', 'P', 'Q']

    random.shuffle(orders)
# Shuffle order of orders
    order_set = [orders[x] for x in range(random.randint(1, len(orders)))]
# Random orders
    return order_set


def map_initialize(warehouse):
    if warehouse == 'a':
        coord_dict = {
            'A': [1, 1],
            'B': [2, 2],
            'C': [1, 3],
            'D': [2, 0],
            'E': [0, 2],
            'F': [2, 4],
```

```
26                 'G': [4, 1],
27                 'H': [5, 4],
28                 'J': [3, 5],
29                 'I': [4, 2]
30             }
31         else:
32             coord_dict = {
33                 'A': [2, 0],
34                 'B': [2, 1],
35                 'C': [0, 3],
36                 'D': [0, 1],
37                 'E': [0, 2],
38                 'F': [2, 2],
39                 'G': [0, 4],
40                 'H': [2, 3],
41                 'I': [0, 5],
42                 'J': [2, 4],
43                 'K': [4, 2],
44                 # 'L': [],
45                 'M': [4, 1],
46                 'N': [4, 5],
47                 'O': [5, 3],
48                 'P': [4, 0],
49                 'Q': [5, 4],
50             }
51         empty_arr = np.array([['*' for i in range(6)] for j in range(6)])
52
```

```
53      # Each time loop runs "letter is different key from dict.
54      for letter, (x_pos, y_pos) in coord_dict.items():    # "letter" represents key
    of dictionary.
55          empty_arr[y_pos][x_pos] = letter                # Key Value ("letter") is
    placed into empty_arr using x and y
56
57      return empty_arr
58
59
60  def fake_shelf(warehouse):
61      if warehouse == 'a':
62          return random.choice(['A', 'B', 'C', 'D', 'E', 'F', 'G', 'H', 'I', 'J'])
63      else:
64          return random.choice(['A', 'B', 'C', 'D', 'E', 'F', 'G', 'H', 'I',
65                                'J', 'K', 'L', 'M', 'N, O', 'P', 'Q'])
66
```

```python
import project_utils
import random

import warehouse as wh
import numpy as np


class Robot:
    def __init__(self, episodes, protocol, layout):
        # Episode tracking
        self.episode = 0
        self.episodes = episodes

        # Order tracking
        self.warehouse = layout
        self.orders = []
        self.items = []
        self.complete = False

        # Score tracking
        self.score = 0
        self.max_score = 0
        self.min_score = 0
        self.avg_score = 0
        self.baseline_score = 0
        self.baseline_score_max = 0
        self.baseline_score_min = 0
```

```
28          self.baseline_score_avg = 0
29
30          # Environment navigation
31          self.false_positive_rate = 10
32          self.false_negative_rate = 10
33          self.surroundings = {
34              'up':       '*',
35              'down':     '*',
36              'left':     '*',
37              'right':    '*',
38          }
39          self.protocol = protocol
40          self.adjacent = False
41          self.next_dir = 'none'
42          self.moves = {
43              'up':       np.array([-1, 0]),
44              'down':     np.array([1, 0]),
45              'left':     np.array([0, -1]),
46              'right':    np.array([0, 1]),
47          }
48
49          # Paths for path protocol (experimental)
50          self.first_lap = [
51              'right',
52              'down',
53              'down',
54              'down',
```

```
55                  'down',
56                  'right',
57                  'right',
58                  'right',
59                  'up',
60                  'up',
61                  'up',
62                  'up',
63              ]
64              self.forward_path = [
65                  'down',
66                  'down',
67                  'down',
68                  'down',
69                  'right',
70                  'right',
71                  'right',
72                  'up',
73                  'up',
74                  'up',
75                  'up',
76              ]
77              self.back_path = [
78                  'down',
79                  'down',
80                  'down',
81                  'down',
```

```
 82              'down',
 83              'left',
 84              'left',
 85              'left',
 86              'up',
 87              'up',
 88              'up',
 89              'up',
 90              'up',
 91          ]
 92
 93          # Position tracking
 94          self.environment = [[]]
 95          self.position = []
 96          self.current_path = [[0, 0]]
 97          self.worst_path = []
 98          self.best_path = []
 99
100      def start_procedure(self):
101          for i in range(self.episodes
    ):                                                  # Perform episodes
102              self.new_episode
                                                                # Start new episode
    ()
103
104      def new_episode(self):
105          self.orders = wh.order_gen(self.warehouse
    )                                       # Generate orders
```

```
106         self.complete = False
                                                        # Initialize episode values
107         self.items = []
108         self.position = np.array([0, 0])
109         self.current_path = [[0, 0]]
110         self.score = 0
111         self.episode += 1
112
113         self.environment = wh.map_initialize(self.warehouse
    )                           # Create map
114
115         self.movement_protocol(self.environment)
116
117     def movement_protocol(self, environment):
118         if self.protocol == 'path'
    :                                                 # For path protocol:
119             self.search_pattern(environment, path=self.first_lap
    )                   # Start first lap
120             while not self.complete
    :                                                # Continue loop until
121                 self.search_pattern(environment, path=self.back_path
    )               # all items are found
122                 if not self.complete
    :                                               # bug fix
123                     self.search_pattern(environment, path=self.forward_path)
124         elif self.protocol == 'random'
    :                                             # For random protocol:
```

```
125             self.search_pattern(environment
    )                                           # no path is passed to search
126
127     def search_pattern(self, environment, path=None
    ):                                      # Step through environment
128         if path:
129             for step in path
    :                                                       # Path contains "steps"
130                 self.move(step, environment
    )                                           # Move method called
131                 self.look_around(environment
    )                                         # View surroundings
132                 self.check_orders(environment
    )                                        # Check surrounding shelves
133                 if self.complete:
134                     break
135         else:
136             move_options = list(self.moves.keys
    ())                                      # Up, Down, Left, Right
137             while not self.complete
    :                                                 # Until all items are found:
138                 self.adjacent = False
                                                     # Robot hasn't sensed a target
139                 self.next_dir = 'none'
                                                    # Robot hasn't decided a direction
140                 self.look_around(environment
    )                                         # View surroundings
```

```
141                 self.check_orders(environment
    )                                       # Check surrounding shelves
142
143                 if self.adjacent
    :                                                   # If an item is identified,
144                     self.move(self.next_dir, environment
    )                           # move to the spot,
145                     item = environment[self.position[0]][self.position[1]]
146                     if item in self.orders and item not in self.items
    :               # and if it is actually a target,
147                         self.items.append(item
    )                                   # pick up target,
148                         environment[self.position[0]][self.position[1]] = '*'
            # remove shelf from targets,
149                     self.complete = self.check_complete
    ()                          # check if order list is complete.
150                 else
    :                                                       # If no target is
    identified,
151                     random_dir = self.random_direction(move_options
    )                # get pseudo-random direction,
152                     self.move(random_dir, environment
    )                           # move to that direction.
153             self.finish_episode()
154
155     def move(self, direction, environment
    ):                                      # Accepts a step (direction)
```

```
156             self.position += self.moves[direction
    ]                                       # Add step to robot position
157             if environment[self.position[0]][self.position[1]] in self.orders
    :               # Assess score from new position
158                 self.score += 3
159             else:
160                 self.score -= 1
161             self.current_path.append(list(self.position
    ))                                  # Store new position
162
163         def random_direction(self, move_options):
164             direction_options
     = []                                                   # Start with no options
165             safe_options
     = []                                                       # and no safe
    options.
166             for proposed_dir in move_options
    :                                           # For a given direction,
167                 proposed_pos = self.position + self.moves[proposed_dir
    ]                   # get the resultant position.
168                 if 0 <= proposed_pos[0] <= 5 and 0 <= proposed_pos[1] <= 5
    :                 # If that is a valid position,
169                     safe_options.append(proposed_dir
    )                                       # add it to safe options,
170                     indicator = True
171                     for element in self.current_path
    :                                       # Of the visited cells,
```

```
172                     if all(element == proposed_pos
    ):                                  # if the proposed move is not new,
173                         indicator = False
                                                # it is a visited cell.
174                 if indicator
    :                                                   # If it is unvisited,
175                     direction_options.append(proposed_dir
    )                           # add it to the options too
176         if bool(direction_options
    ):                                                  # If there are unvisited cells,
177             return random.choice(direction_options
    )                                   # move to one of them.
178         else
    :                                                                   # If all
    cells have been visited,
179             return random.choice(safe_options
    )                                       # move to a random valid position
180
181     def look_around(self, environment
    ):                                              # Read from "sensors"
182         sensor_position = {
183             'up': self.position + self.moves['up'],
184             'down': self.position + self.moves['down'],
185             'left': self.position + self.moves['left'],
186             'right': self.position + self.moves['right']
187         }
188
```

```
189         # self.check_error
    ()                                                  # Error rates from
    prompt
190
191         for sensor in self.surroundings:
192             try
    :                                                               # Wall finding
    , shows empty cell
193                 if 0 <= sensor_position[sensor][0] <= 5 and 0 <= sensor_position[
    sensor][1] <= 5:
194                     self.surroundings[sensor] = environment[sensor_position[sensor
    ][0]][sensor_position[sensor][1]]
195                 else:
196                     self.surroundings[sensor] = '*'
197             except IndexError:
198                 self.surroundings[sensor] = '*'
199
200         self.check_error()
201
202     def check_error(self):
203         false_pos = random.randint(1, 100
    )                                           # Generate each type of error
204         false_neg = random.randint(1, 100
    )                                           # Error rates from prompt
205
206         if false_pos <= self.false_positive_rate
    :                                   # False positive
```

```
207             self.surroundings[random.choice(list(self.surroundings))] = 'fake'
          # Places a random fake "shelf"
208
209         if false_neg <= self.false_negative_rate
    :                                       # False negative
210             self.surroundings[random.choice(list(self.surroundings))] = '*'
           # Places a random fake "empty"
211
212     def check_orders(self, environment):
213         if self.protocol == 'path'
    :                                                # For path protocol:
214             for item in self.orders
    :                                               # If a required item from the
215                 if item in self.surroundings.values() and item not in self.items
    :   # order list is adjacent,
216                     self.retrieve(item, environment
    )                                   # Retrieve it
217             if 'fake' in self.surroundings
    :                                           # Same process for fake but is
218                 self.retrieve('fake', environment
    )                                     # handled in retrieve method
219         elif self.protocol == 'random':
220             for direction, value in self.surroundings.items
    ():                           # For random protocol:
221                 if not self.adjacent
    :                                               # If a target has not been found,
222                     if value == 'fake'
```

```
222 :                                                  # If there is a false positive
223                             fake = wh.fake_shelf(self.warehouse
    )                            # Generate a fake shelf
224                             self.surroundings[direction] = fake
225                             value = fake
226                         for item in self.orders
    :                                                  # Otherwise, check for targets
227                             if item == value and item not in self.items
    :                   # If there is a valid target
228                                 proposed_pos = self.position + self.moves[direction]
229                                 if 0 <= proposed_pos[0] <= 5 and 0 <= proposed_pos[1
    ] <= 5:
230                                     self.adjacent = True
                                        # Change indicator to True,
231                                     self.next_dir = direction
                                    # Choose that target,
232                                     break
                                                       # and stop looking
233
234     def retrieve(self, item, environment):
235         for direction, shelf in self.surroundings.items
    ():                              # From surroundings,
236             if shelf == item
    :                                                      # find shelf containing
    target
237                 if item != 'fake'
    :                                                  # For real items,
```

```
238                     self.move(direction, environment
    )                                   # move to shelf,
239                     self.items.append(item
    )                                           # grab item,
240                     environment[self.position[0]][self.position[1]] = '*'
                # do not reward future visits,
241                     if self.check_complete
    ():                                         # check if orders completed,
242                         self.finish_episode
    ()                                        # finish episode if completed
243                     back = project_utils.direction_flip(direction
    )                  # if orders remain, move back
244                     self.move(back, environment)
245                 else:
246                     fake_shelf = wh.fake_shelf(self.warehouse)
247                     if fake_shelf in self.orders and fake_shelf not in self.items
    :  # If the fake is a target shelf,
248                         self.move(direction, environment
    )                           # try and fail to get the item
249                         back = direction * -1
250                         self.move(back, environment)
251
252     def check_complete(self):
253         for item in self.orders
    :                                                   # Check order list
254             if item not in self.items
    :                                               # If an order has not yet been added
```

```
255                 return False
                                                                # not complete.
256         return True
                                                                    # If all items are
     found, complete.
257
258     def finish_episode(self
    ):                                                           # Update metrics
259         self.avg_score = (self.avg_score * (self.episode - 1) + self.score) / self.
    episode
260         self.baseline_score = 4 * len(self.orders) - 35
                                  # Brute force score, per Dr. Pears
261         self.baseline_score_avg = (self.baseline_score_avg * (self.episode - 1) +
    self.baseline_score) / self.episode
262         if self.episode == 1
    :                                                           # Initial values after
    episode 1
263             self.max_score = self.score
264             self.baseline_score_max = self.baseline_score
265             self.min_score = self.score
266             self.baseline_score_min = self.baseline_score
267             self.best_path = self.current_path
268             self.worst_path = self.current_path
269         else
    :                                                                       # Update
    values if needed
270             if self.score > self.max_score
```

```
270 :                                           # Best score and path
271                 self.max_score = self.score
272                 self.baseline_score_max = self.baseline_score
273                 self.best_path = self.current_path
274             if self.score < self.min_score
    :                                           # Worst score and path
275                 self.min_score = self.score
276                 self.baseline_score_min = self.baseline_score
277                 self.worst_path = self.current_path
278         self.complete = True
279
```

```python
import pygame
import sys

import warehouse as wh

from pygame.locals import *

# Globals:
BLACK = (0, 0, 0)
WHITE = (200, 200, 200)
GRAY = (100, 100, 100)
GREEN = (47, 237, 155)
RED = (224, 80, 61)
WINDOW_HEIGHT = 600
WINDOW_WIDTH = 600

BLOCK_SIZE = int(WINDOW_WIDTH / 6
)                                                    # Set the size of the grid
block
SCREEN = pygame.display.set_mode((WINDOW_WIDTH, WINDOW_HEIGHT
))                           # Prepare screen object


def display_results(robot):
    pygame.init
()                                                                  # Pygame
initialize
```

```
23      for window in [0, 1
    ]:                                                          # For best and
    worst paths
24          screen_change = True
25
26          # Window title info
27          info = ', press any key to continue...'
28          caption = f'Best Score: {robot.max_score}{info}' if window == 0 else f'Worst
     Score: {robot.min_score}{info}'
29          pygame.display.set_caption(f'{caption}')
30
31          # More setup
32          SCREEN.fill(BLACK)
33          font = pygame.font.SysFont(name='arial', size=20)
34
35          # My stuff
36          draw_grid
    ()                                                                  # Draw grid
37          draw_shelves(font, robot.warehouse
    )                                               # Draw warehouse layout
38          path = robot.best_path if window == 0 else robot.worst_path
                        # Pick path to display,
39          draw_path(path
    )                                                           # and draw said
    path.
40
41          # Typical pygame loop
```

```
42          while screen_change:
43              for event in pygame.event.get():
44                  if event.type == pygame.QUIT:
45                      pygame.quit()
46                      sys.exit()
47                  if event.type == KEYDOWN
    :                                                  # Next screen
48                      screen_change = False
49
50              pygame.display.update()
51
52
53  def draw_shelves(font, warehouse):
54      environment = wh.map_initialize(warehouse
    )                                                # To be displayed
55      for i in range(len(environment)):
56          for j in range(len(environment[0])):
57              cell = environment[i][j
    ]                                                        # Get each value
58              color = GRAY if cell == '*' else GREEN
                                                   # Shelves are green
59              img = font.render(cell, True, color
    )                                               # Render text
60              x = (j + .25) * BLOCK_SIZE
                                                           # Position to display
61              y = (i + .25) * BLOCK_SIZE
62              SCREEN.blit(img, (x, y
```

```
62 ))                                                    # Place on screen object
63
64
65 def draw_path(path):
66     last_coord = [.5 * BLOCK_SIZE, .5 * BLOCK_SIZE
   ]                                          # Start from origin
67     for coord in path:
68         next_coord = [(coord[1] + .5) * BLOCK_SIZE, (coord[0] + .5) * BLOCK_SIZE
   ]       # Get endpoint
69         pygame.draw.line(SCREEN, RED, last_coord, next_coord, width=5
   )                  # Draw line from last to next
70         last_coord = next_coord
                                                             # Current becomes last
   point
71
72
73 def draw_grid
   ():                                                                  # Draw
   grid
74     for x in range(0, WINDOW_WIDTH, BLOCK_SIZE):
75         for y in range(0, WINDOW_HEIGHT, BLOCK_SIZE):
76             rect = pygame.Rect(x, y, BLOCK_SIZE, BLOCK_SIZE)
77             pygame.draw.rect(SCREEN, WHITE, rect, 1)
78
79
80 def report_printout(robot):
81     print('\n################################\n')
```

```
 82      print(f'Average Score: {round(robot.avg_score, 3)}'
    )                                    # Print results
 83      print(f'Avg. Brute Force Score: {round(robot.baseline_score_avg, 3)}\n')
 84
 85      print(f'Maximum Score: {robot.max_score}')
 86      print(f'Corresponding Brute Force Score: {robot.baseline_score_max}\n')
 87
 88      print(f'Minimum Score: {robot.min_score}')
 89      print(f'Corresponding Brute Force Score: {robot.baseline_score_min}')
 90      print('\n################################\n')
 91
 92      display_results(robot
    )                                                        # Display paths
 93
 94
 95  def direction_flip(direction
    ):                                                     # Ugly solution
 96      if direction == 'up':
 97          return 'down'
 98      elif direction == 'down':
 99          return 'up'
100      elif direction == 'left':
101          return 'right'
102      elif direction == 'right':
103          return 'left'
104      else:
105          return None
```