```
# Function to read the maze from filename
def readMaze(filename):
   f = open(filename, "r")
   if f.mode == "r":
       walls = []      ## defines locations of list walls as (row, col)
       foods = []      ## defines locations of list foods as (row, col)
       pacmanPos = 0   ## define pacman position as tuple (row, col)
       y = 10
       while True:
           str = f.readline()      ## read one line from the file
           if str=="": break       ## stop loop if an empty (or end of file)
string is reached.
```

```
            x = 10
            for k in str:
                if k == '*':    ## star indicate a wall
                    walls.append((x, y))  # append (row, col) to walls list
                if k == '.':    ## period indicates a food
                    foods.append((x, y))    # append (row, col) to foods list
                if k == 'P':    ## letter P indicates player
                    pacmanPos = (x, y)  # set pacman position to (row, col)
                x += Problem.xStep
            y += Problem.yStep
        Problem.xMax = x            # save row in the problem static data class f
later use
        Problem.yMax = y            # save col in the problem static data class f
```

```
later use

       Problem.walls = walls   # save walls in the problem static data class for

later use

       return Problem(foods, pacmanPos)  # declare class Problem with foods and

pacman position

class Problem():

   walls = 0

   xMax = 0

   yMax = 0

   xStep = 40

   yStep = 40

   directions = {'u': (0, -yStep), 'd': (0, yStep), 'l': (-xStep, 0), 'r': (xSte

0)} # direction as dictionary
```

```
   def __init__(self, foods, pacmanPos):
       self.foods = foods
       self.pacmanPos = pacmanPos
   def isGoal (self, currentPos):  ## goal is true when current position of pacm
reaches the food
       if currentPos == self.foods[0]: return True
       return False
   def startState (self):  ## start state is pacman position
       return self.pacmanPos
   def legalActions (self, currentPos):    ## return legal actions for the curre
position
       x, y = currentPos
       actions = []
```

```
        for action in Problem.directions.keys(): ##  select an action: u, d, l, r
            dx, dy = Problem.directions[action]
            newPos  = (x + dx, y + dy)  # compute new position for that action
            x1, y1 = newPos
            # if new position is out of the maze boundaries, then skip that new
position
            if x1 < 10 or y1 > Problem.yMax: continue
            if y1 < 10 or y1 > Problem.yMax: continue
            # if the new position is in the walls list, then skip that new positi
            if newPos in Problem.walls: continue
            # save the action in the actions list.
            actions.append(action)
        return actions
```

```
   # method to compute the next position after applying the action on the curren
position
   def successor (self, action, currentPos):
       dx, dy = Problem.directions[action]
       x, y = currentPos
       newPos = (x + dx, y + dy)
       return newPos
```