1) Introduction:-

Pacman debuted in the 1980s as an arcade classic that features Pacman navigating mazes, gobbling dots, and dodging ghosts. Beyond its entertainment value, Pacman has played a pivotal role in the development of artificial intelligence, serving as a testing ground for AI algorithms. Our goal is to use Reinforcement learning to be able to play Pacman and successfully consume all the pellets while getting the highest score possible.

2) How Does the Game Work?

Pacman navigates through a maze with ghosts either chasing Pacman or scattering, with chasing happening 80% of the time while scattering is done 20% of the time, the goal of the game is for Pacman to be able to consume all the pellets within the maze without being killed by the ghosts while also trying to get the highest score possible.

3) How Does the Score Work?

Rewards	Points
Running into an active ghost	-250
Eating a scared ghost	+100
Winning (eating last dot)	+250
Eating a single dot	+1
Not eating anything	-1

4) What Files did we use?

- autoPlay & autoTrain: both files just continuously plays pacman
- game: main code for handling game operations
- ghostAgents: Agents responsible for handling ghosts
- graphicsDisplay: Responsible for displaying GUI
- graphicsUtils: Responsible for handling GUI utilities
- keyboardAgents: Responsible for keyboard inputs for when the user plays
- Layout: Responsible for handling the maze layout
- pacman: Responsible for running the game
- pacmanAgents: Agents that will play pacman
- util: responsible for handling the main Utilities

autoPlay
autoTrain
game
ghostAgents
graphicsDisplay
graphicsUtils
keyboardAgents
layout
multiAgents
oldWeights
pacman
pacmanAgents
util

5) autoPlay.py File:-

```python
import subprocess  # Importing the subprocess module to run external commands
import time
import numpy as np

# Setting the number of episodes to run
episodes = 100

# Looping through the episodes
for i in range(0, episodes):
    print("Running Episode", i)  # Printing the current episode number

    # Running the external command to execute a Pacman game and capturing the output
    result = subprocess.run("python pacman.py -p ReflexAgent -l mediumClassic --frameTime 0", stdout=subprocess.PIPE)

    # Converting the stdout to a string
    test = str(result.stdout)

    # Initializing the win flag as 'Lose'
    winFlag = 'Lose'

    # Checking if the word 'victorious' exists in the output string
    if (test.find('victorious') != -1):
        winFlag = 'Win'  # Changing the win flag to 'Win' if Pacman is victorious

    # Appending the win flag to a text file named 'WinRatio.txt'
    with open('WinRatio.txt', 'a') as f:
        f.write(winFlag)  # Writing the win flag
        f.write('\n')  # Adding a new line after each win/loss entry
```

- This file simply just loops through multiple episodes through the command “python pacman.py -p ReflexAgent -k 2 --frameTime 0” which runs the game and checks whether pacman won or lost each game and writes it in a text file called WinRatio.txt.

6) keyboardAgents.py File:-

```
from game import Agent
from game import Directions
import random
class KeyboardAgent(Agent):
    WEST_KEY = 'a'
    EAST_KEY = 'd'
    NORTH_KEY = 'w'
    SOUTH_KEY = 's'
    STOP_KEY = 'q'
    def __init__(self, index=0):
        self.lastMove = Directions.STOP
        self.index = index
        self.keys = []

    def getAction(self, state):
        from graphicsUtils import keys_waiting
        from graphicsUtils import keys_pressed
        keys = keys_waiting() + keys_pressed()
        if keys != []:
            self.keys = keys

        legal = state.getLegalActions(self.index)
        move = self.getMove(legal)

        if move == Directions.STOP:
            if self.lastMove in legal:
                move = self.lastMove

        if (self.STOP_KEY in self.keys) and Directions.STOP in legal:
            move = Directions.STOP

        if move not in legal:
            move = random.choice(legal)

        self.lastMove = move
        return move

    def getMove(self, legal):
        move = Directions.STOP
        if (self.WEST_KEY in self.keys or 'Left' in self.keys) and Directions.WEST in legal:
            move = Directions.WEST
        if (self.EAST_KEY in self.keys or 'Right' in self.keys) and Directions.EAST in legal:
            move = Directions.EAST
        if (self.NORTH_KEY in self.keys or 'Up' in self.keys) and Directions.NORTH in legal:
            move = Directions.NORTH
        if (self.SOUTH_KEY in self.keys or 'Down' in self.keys) and Directions.SOUTH in legal:
            move = Directions.SOUTH
        return move
```

Here we assigned the usual wasd to each direction with q to stop and to handle which button is pressed for when the user is the one who plays the game

W => North

A => East

S => South

D => East

Q => Stop

7) layout.py File:-

```
class Layout:
    """
    A Layout manages the static information about the game board.
    """
    def __init__(self, layoutText):
        # Initializing width and height based on the layoutText dimensions.
        self.width = len(layoutText[0])
        self.height = len(layoutText)
        # Grid to represent walls, food, and agents' positions.
        self.walls = Grid(self.width, self.height, False)
        self.food = Grid(self.width, self.height, False)
        self.capsules = []  # List to store capsule positions.
        self.agentPositions = []  # List to store agents' positions.
        self.numGhosts = 0  # Counter for the number of ghosts.
        # Processing layout text to initialize the layout.
        self.processLayoutText(layoutText)
        self.layoutText = layoutText
        self.totalFood = len(self.food.asList())

    def getNumGhosts(self):
        # Returns the number of ghosts in the layout.
        return self.numGhosts
```

- First we start with a Layout class, and initialize the needed variables
- Function getNumGhosts just returns the number of ghosts in the layout

```
    def initializeVisibilityMatrix(self):
        # Initializing visibility matrix for each position on the layout.
        global VISIBILITY_MATRIX_CACHE
        if reduce(str.__add__, self.layoutText) not in VISIBILITY_MATRIX_CACHE:
            from game import Directions
            # Vectors and corresponding directions for visibility calculation.
            vecs = [(-0.5, 0), (0.5, 0), (0, -0.5), (0, 0.5)]
            dirs = [Directions.NORTH, Directions.SOUTH,
                    Directions.WEST, Directions.EAST]
            # Initializing visibility grid.
            vis = Grid(self.width, self.height, {Directions.NORTH: set(), Directions.SOUTH: set(
            ), Directions.EAST: set(), Directions.WEST: set(), Directions.STOP: set()})
            for x in range(self.width):
                for y in range(self.height):
                    if self.walls[x][y] == False:
                        for vec, direction in zip(vecs, dirs):
                            dx, dy = vec
                            nextx, nexty = x + dx, y + dy
                            # Adding visible positions in each direction.
                            while (nextx + nexty) != int(nextx) + int(nexty) or not self.walls[int(nextx)][int(nexty)]:
                                vis[x][y][direction].add((nextx, nexty))
                                nextx, nexty = x + dx, y + dy
            self.visibility = vis
            VISIBILITY_MATRIX_CACHE[reduce(str.__add__, self.layoutText)] = vis
        else:
            self.visibility = VISIBILITY_MATRIX_CACHE[reduce(
                str.__add__, self.layoutText)]
```

- Here's how the initalizeVisibilityMatrix function works:-
  - It first checks if the visibility matrix for the current layout configuration is already cached. If it is, it retrieves the cached matrix and assigns it to the visibility attribute of the current object.
  - If the visibility matrix is not cached, it proceeds to calculate it.

- For each position (x, y) on the layout:
  - If the position is not a wall (i.e., it's traversable), it calculates the visible positions in each direction (north, south, east, west).
  - It iterates over a set of vectors and corresponding directions representing the four cardinal directions (north, south, east, west).
  - For each direction, it calculates the visible positions by extending the line of sight until it encounters a wall or reaches the edge of the layout.
  - It adds these visible positions to the visibility grid.
- After calculating the visibility matrix, it caches it for future use based on the layout configuration.
- Finally, it assigns the visibility matrix to the visibility attribute of the current object.

```python
    def isWall(self, pos):
        # Check if a position is a wall.
        x, col = pos
        return self.walls[x][col]

    def getRandomLegalPosition(self):
        # Return a random legal position on the layout.
        x = random.choice(list(range(self.width)))
        y = random.choice(list(range(self.height)))
        while self.isWall((x, y)):
            x = random.choice(list(range(self.width)))
            y = random.choice(list(range(self.height)))
        return (x, y)

    def getRandomCorner(self):
        # Return a random corner position on the layout.
        poses = [(1, 1), (1, self.height - 2), (self.width - 2, 1),
                 (self.width - 2, self.height - 2)]
        return random.choice(poses)

    def getFurthestCorner(self, pacPos):
        # Return the furthest corner position from a given position.
        poses = [(1, 1), (1, self.height - 2), (self.width - 2, 1),
                 (self.width - 2, self.height - 2)]
        dist, pos = max([(manhattanDistance(p, pacPos), p) for p in poses])
        return pos

    def isVisibleFrom(self, ghostPos, pacPos, pacDirection):
        # Check if a ghost is visible from Pacman's position and direction.
        row, col = [int(x) for x in pacPos]
        return ghostPos in self.visibility[row][col][pacDirection]

    def __str__(self):
        # Return the string representation of the layout.
        return "\n".join(self.layoutText)

    def deepCopy(self):
        # Create a deep copy of the layout.
        return Layout(self.layoutText[:])
```

- isWall just returns the positions of where the walls are
- getRandomLegalPosition where returns a legal position at random
- getFurthestCorner just returns the furthest corner from pacman's position

- isVisibleFrom checks whether the ghosts are visible for pacman from his position
- __str__ just returns a string representation of the layout
- deepCopy just creates a copy of the layout

```
    def processLayoutText(self, layoutText):
        """
        Coordinates are flipped from the input format to the (x,y) convention here

        The shape of the maze.  Each character
        represents a different type of object.
         % - Wall
         . - Food
         o - Capsule
         G - Ghost
         P - Pacman
        Other characters are ignored.
        """
        maxY = self.height - 1
        for y in range(self.height):
            for x in range(self.width):
                layoutChar = layoutText[maxY - y][x]
                # Processing each character in the layout text.
                self.processLayoutChar(x, y, layoutChar)
        self.agentPositions.sort()
        self.agentPositions = [(i == 0, pos) for i, pos in self.agentPositions]

    def processLayoutChar(self, x, y, layoutChar):
        # Process each character in the layout text and update layout attributes accordingly.
        if layoutChar == '%':
            self.walls[x][y] = True
        elif layoutChar == '.':
            self.food[x][y] = True
        elif layoutChar == 'o':
            self.capsules.append((x, y))
        elif layoutChar == 'P':
            self.agentPositions.append((0, (x, y)))
        elif layoutChar in ['G']:
            self.agentPositions.append((1, (x, y)))
            self.numGhosts += 1
        elif layoutChar in ['1', '2', '3', '4']:
            self.agentPositions.append((int(layoutChar), (x, y)))
            self.numGhosts += 1
```

- Here’s how processLayoutText function works:-
  - It iterates over each row (y) and column (x) of the layout text.
  - For each position (x, y), it retrieves the corresponding character from the layout text (layoutChar).
  - It processes each character in the layout text using the processLayoutChar method, which assigns properties to the maze elements based on the characters:
    - %: Wall
    - .: Food
    - o: Capsule
    - G: Ghost
    - P: Pacman

- Other characters are ignored.
- After processing all characters, it sorts the agent positions and converts them into a tuple of a boolean value indicating whether it's Pacman (True for Pacman, False for Ghost) and the position coordinates.
- Finally, it assigns the sorted and converted agent positions back to the agentPositions attribute of the layout.

- processLayoutChar function initializes the layout object based on the layout text, setting up walls, food, capsules, and agent positions (Pacman and ghosts) according to the characters in the layout text.

```
def getLayout(name, back=2):
    # Load layout from file.
    if name.endswith('.lay'):
        layout = tryToLoad('layouts/' + name)
        if layout == None:
            layout = tryToLoad(name)
    else:
        layout = tryToLoad('layouts/' + name + '.lay')
        if layout == None:
            layout = tryToLoad(name + '.lay')
    if layout == None and back >= 0:
        curdir = os.path.abspath('.')
        os.chdir('..')
        layout = getLayout(name, back - 1)
        os.chdir(curdir)
    return layout


def tryToLoad(fullname):
    # Attempt to load a layout from a file.
    if(not os.path.exists(fullname)):
        return None
    f = open(fullname)
    try:
        return Layout([line.strip() for line in f])
    finally:
        f.close()
```

- getLayout function used to load maze layouts from files. It accepts the name of the layout file as input and attempts to load it. If the file extension is not provided, it tries both with and without the ".lay" extension. It recursively searches for the layout file in parent directories if it's not found in the current directory. Finally, it returns the loaded layout or None if the layout file is not found.
- tryToLoad just loads files.

8) ghostAgents.py File:-

- We have 2 agents for ghosts

```
class RandomGhost(GhostAgent):

    # A ghost that chooses a legal action uniformly at random.


    def getDistribution(self, state):
        # Returns a distribution over legal actions.

        dist = util.Counter()
        for a in state.getLegalActions(self.index):
            dist[a] = 1.0
        dist.normalize()
        return dist
```

- RandomGhost agent chooses moves uniformly at random

```
class DirectionalGhost(GhostAgent):
    # A ghost that prefers to rush Pacman, or flee when scared.
    def __init__(self, index, prob_attack=0.8, prob_scaredFlee=0.8):
        # Initializes the ghost agent with probabilities for attacking and fleeing.
        self.index = index
        self.prob_attack = prob_attack
        self.prob_scaredFlee = prob_scaredFlee

    def getDistribution(self, state):
        # Returns a distribution over legal actions.

        ghostState = state.getGhostState(self.index)
        legalActions = state.getLegalActions(self.index)
        pos = state.getGhostPosition(self.index)
        isScared = ghostState.scaredTimer > 0

        speed = 1
        if isScared:
            speed = 0.5

        actionVectors = [Actions.directionToVector(
            a, speed) for a in legalActions]
        newPositions = [(pos[0]+a[0], pos[1]+a[1]) for a in actionVectors]
        pacmanPosition = state.getPacmanPosition()

        distancesToPacman = [manhattanDistance(
            pos, pacmanPosition) for pos in newPositions]
        if isScared:
            bestScore = max(distancesToPacman)
            bestProb = self.prob_scaredFlee
        else:
            bestScore = min(distancesToPacman)
            bestProb = self.prob_attack
        bestActions = [action for action, distance in zip(
            legalActions, distancesToPacman) if distance == bestScore]

        dist = util.Counter()
        for a in bestActions:
            dist[a] = bestProb / len(bestActions)
        for a in legalActions:
            dist[a] += (1-bestProb) / len(legalActions)
        dist.normalize()
        return dist
```

- Here’s how this agent works
  - It retrieves relevant information from the state:
    - Ghost state and whether it's scared.
    - Legal actions the ghost can take.
    - Ghost's current position.
    - Pacman's position.
    - It adjusts the speed of the ghost based on whether it's scared or not.
  - It calculates new potential positions for the ghost based on the legal actions it can take.
  - It calculates the distances from these potential positions to Pacman's position.
  - If the ghost is scared, it determines the best score by maximizing distances to Pacman. Otherwise, it minimizes distances.
  - It identifies the best actions based on the best score calculated.
  - It constructs a probability distribution over the legal actions:
    - Assigns a higher probability to the best actions.
    - Normalizes the distribution.
  - It returns the probability distribution over legal actions.

9) pacmanAgents.py File:-

```
class LeftTurnAgent(game.Agent):
    "An agent that turns left at every opportunity"

    # Define method to get the action of the agent
    def getAction(self, state):
        # Get legal actions available to the agent
        legal = state.getLegalPacmanActions()
        # Get the current direction of the agent
        current = state.getPacmanState().configuration.direction
        # If the agent is currently stopped, set the direction to north
        if current == Directions.STOP:
            current = Directions.NORTH
        # Calculate the direction to the left of the current direction
        left = Directions.LEFT[current]
        # If the left direction is legal, turn left
        if left in legal:
            return left
        # If the current direction is legal, continue straight
        if current in legal:
            return current
        # If turning right is legal, turn right
        if Directions.RIGHT[current] in legal:
            return Directions.RIGHT[current]
        # If turning left from the left direction is legal, perform a U-turn
        if Directions.LEFT[left] in legal:
            return Directions.LEFT[left]
        # If none of the above actions are possible, stop
        return Directions.STOP
```

- LeftTurnAgent essentially follows a left-hand rule: it always tries to turn left whenever it can. If it can't turn left, it goes straight, and if it can't go straight either, it turns right or performs a U-turn. If none of these actions are possible, it stops. This idea is inspired by the idea of that any maze can be solved by holding the left wall

```
# Define a class for a greedy agent that chooses actions based on a provided evaluation function
class GreedyAgent(Agent):
    def __init__(self, evalFn="scoreEvaluation"):
        self.evaluationFunction = util.lookup(evalFn, globals())
        assert self.evaluationFunction != None

    # Define method to get the action of the agent
    def getAction(self, state):
        # Generate candidate actions
        legal = state.getLegalPacmanActions()
        # Remove the STOP action if present
        if Directions.STOP in legal:
            legal.remove(Directions.STOP)

        # Generate successor states for each legal action
        successors = [(state.generateSuccessor(0, action), action)
                      for action in legal]
        # Evaluate each successor state using the evaluation function
        scored = [(self.evaluationFunction(state), action)
                  for state, action in successors]
        # Find the best score among the evaluated successor states
        bestScore = max(scored)[0]
        # Select actions that lead to the best score
        bestActions = [pair[1] for pair in scored if pair[0] == bestScore]
        # Choose a random action among the best actions
        return random.choice(bestActions)

# Define a simple evaluation function that returns the score of the state
def scoreEvaluation(state):
    return state.getScore()
```

- This agent operates by evaluating all possible successor states based on the provided evaluation function and selecting the action that maximizes the evaluation score. If multiple actions lead to the same best score, it chooses randomly among them.

- getAction method: This method defines how the agent selects its action given the current state of the game.
    - It starts by generating all legal actions available to the agent in the current state, excluding the STOP action if present.
    - For each legal action, it generates the successor state resulting from applying that action to the current state.
    - It evaluates each successor state using the evaluation function, producing a list of (score, action) tuples.
    - It then identifies the best score among the evaluated successor states.
    - Finally, it selects actions that lead to the best score and randomly chooses one among them to return as the agent's action.
- scoreEvaluation just returns the score of the state

10) util.py File:-

```
class FixedRandom:
    def __init__(self):
        # Initialize with a fixed state for reproducibility
        fixedState = (3, (2147483648, 507801126, 683453281, 310439348, 2597246090,
                          2209084787, 2267831527, 979920060, 3098657677, 37650879, 807947081, 3974896263,
                          881243242, 3100634921, 1334775171, 3965168385, 746264660, 4074750168, 500078808,
                          776561771, 702988163, 1636311725, 2559226045, 157578202, 2498342920, 2794591496,
                          4130598723, 496985844, 2944563015, 3731321600, 3514814613, 3362575829, 3038768745,
                          2206497038, 1108748846, 1317460727, 3134077628, 988312410, 1674063516, 746456451,
                          3958482413, 1857117812, 708750586, 1583423339, 3466495450, 1536929345, 1137240525,
                          3875025632, 2466137587, 1235845595, 4214575620, 3792516855, 657994358, 1241843248,
                          1695651859, 3678946666, 1929922113, 2351044952, 2317810202, 2039319015, 460787996,
                          3654096216, 4068721415, 1814163703, 2904112444, 1386111013, 574629867, 2654529343,
                          3833135042, 2725328455, 552431551, 4006991378, 1331562057, 3710134542, 303171486,
                          1203231078, 2670768975, 54570816, 2679609001, 578983064, 1271454725, 3230871056,
                          2496832891, 2944938195, 1608828728, 367886575, 2544708204, 103775539, 1912402393,
                          1098482180, 2738577070, 3091646463, 1505274463, 2079416566, 659100352, 839995305,
                          1696257633, 274389836, 3973303017, 671127655, 1061109122, 517486945, 1379749962,
                          3421383928, 3116950429, 2165882425, 2346928266, 2892678711, 2936066049, 1316407868,
                          2873411858, 4279682888, 2744351923, 3290373816, 1014377279, 955200944, 4220990860,
                          2386098930, 1772997650, 3757346974, 1621616438, 2877097197, 442116595, 2010480266,
                          2867861469, 2955352695, 605335967, 2222936009, 2067554933, 4129906358, 1519608541,
                          1195006590, 1942991038, 2736562236, 279162408, 1415982909, 4099901426, 1732201505,
                          2934657937, 860563237, 2479235483, 3081651097, 2244720867, 3112631622, 1636991639,
                          3860393305, 2312061927, 48780114, 1149090394, 2643246550, 1764050647, 3836789087,
                          3474859076, 4237194338, 1735191073, 2150369208, 92164394, 756974036, 2314453957,
                          323969533, 4267621035, 283649842, 810004843, 727855536, 1757827251, 3334960421,
                          3261035106, 38417393, 2660980472, 1256633965, 2184045390, 811213141, 2857482069,
                          2237770878, 3891003138, 2787806886, 2435192790, 2249324662, 3507764896, 995388363,
                          856944153, 619213904, 3233967826, 3703465555, 3286531781, 3863193356, 2992340714,
                          413696855, 3865185632, 1704163171, 3043634452, 2225424707, 2199018022, 3506117517,
                          3311559776, 3374443561, 1207829628, 668793165, 1822020716, 2082656160, 1160606415,
                          3034757648, 741703672, 3094328738, 459332691, 2702383376, 1610239915, 4162939394,
                          557861574, 3805706338, 3832520705, 1248934879, 3250424034, 892335058, 74323433,
                          3209751608, 3213220797, 3444035873, 3743886725, 1783837251, 610968664, 580745246,
                          4041979504, 201684874, 2673219253, 1377283008, 3497299167, 2344209394, 2304982920,
                          3081403782, 2599256854, 3184475235, 3373055826, 695186388, 2423332338, 222864327,
                          1258227992, 3627871647, 3487724980, 4027953808, 3053320360, 533627073, 3026232514,
                          2340271949, 867277230, 868513116, 2158535651, 2487822909, 3428235761, 3067196046,
                          3435119657, 1908441839, 788668797, 3367703138, 3317763187, 908264443, 2252100381,
                          764223334, 4127108988, 384641349, 3377374722, 1263833251, 1958694944, 3847832657,
                          1253909612, 1096494446, 555725445, 2277045895, 3340096504, 1383318686, 4234428127,
                          1072582179, 94169494, 1064509968, 2681151917, 2681864920, 734708852, 1338914021,
                          1270409500, 1789469116, 4191988204, 1716329784, 2213764829, 3712538840, 919910444,
                          1318414447, 3383806712, 3054941722, 3378649942, 1205735655, 1268136494, 2214009444,
                          2532395133, 3232230447, 230294038, 342599089, 772808141, 4096882234, 3146662953,
                          2784264306, 1860954704, 2675279609, 2984212876, 2466966981, 2627986059, 2985545332,
                          2578042598, 1458940786, 2944243755, 3959506256, 1509151382, 325761900, 942251521,
```

- FixedRandom class is first initialized with a tuple of all the fixedStates

```
        self.random = random.Random()
        self.random.setstate(fixedState)

"""
 Data structures useful for implementing SearchAgents
"""
class Stack:
    def __init__(self):
        self.list = []

    def push(self, item):
        "Push 'item' onto the stack"
        self.list.append(item)

    def pop(self):
        "Pop the most recently pushed item from the stack"
        return self.list.pop()

    def isEmpty(self):
        "Returns true if the stack is empty"
        return len(self.list) == 0


def manhattanDistance(xy1, xy2):
    "Returns the Manhattan distance between points xy1 and xy2"
    return abs(xy1[0] - xy2[0]) + abs(xy1[1] - xy2[1])
```

- creates a new instance of the random.Random() class, which provides functionality for generating random numbers. Then, it sets the state of this random number generator to the fixed state using the setstate() method. This ensures that subsequent calls to the random methods (e.g., random() or randint()) will produce the same sequence of pseudo-random numbers every time the program runs.
- Class Stack is just for creating a stack that'll be used later in the searching algorithm and in some debugging
- manhanttanDistance function just calculates the ManhattanDistance between two points

```
class Counter(dict):
    """
    A counter keeps track of counts for a set of keys.

    The counter class is an extension of the standard python
    dictionary type.  It is specialized to have number values
    (integers or floats), and includes a handful of additional
    functions to ease the task of counting data.  In particular,
    all keys are defaulted to have value 0.  Using a dictionary:

    a = {}
    print a['test']

    would give an error, while the Counter class analogue:

    >>> a = Counter()
    >>> print a['test']
    0

    returns the default 0 value. Note that to reference a key
    that you know is contained in the counter,
    you can still use the dictionary syntax:

    >>> a = Counter()
    >>> a['test'] = 2
    >>> print a['test']
    2

    This is very useful for counting things without initializing their counts,
    see for example:

    >>> a['blah'] += 1
    >>> print a['blah']
    1

    The counter also includes additional functionality useful in implementing
    the classifiers for this assignment.  Two counters can be added,
    together.  See below for details.  They can also be normalized and their
    total count and arg max can be extracted.
    """

    def __getitem__(self, idx):
        self.setdefault(idx, 0)
        return dict.__getitem__(self, idx)
```

- The idea and the goal of this class is explained in the multi-line comment, so let's explain how it's implemented
- In the method __getitem__ we first ensures that if the key idx does not exist in the dictionary, it sets the default value of 0 for that key then returns the value associated with the key idx in the dictionary.

```
    def incrementAll(self, keys, count):
        """
        Increments all elements of keys by the same count.

        >>> a = Counter()
        >>> a.incrementAll(['one','two', 'three'], 1)
        >>> a['one']
        1
        >>> a['two']
        1
        """
        for key in keys:
            self[key] += count

    def totalCount(self):
        """
        Returns the sum of counts for all keys.
        """
        return sum(self.values())

    def normalize(self):
        """
        Edits the counter such that the total count of all
        keys sums to 1.  The ratio of counts for all keys
        will remain the same. Note that normalizing an empty
        Counter will result in an error.
        """
        total = float(self.totalCount())
        if total == 0:
            return
        for key in list(self.keys()):
            self[key] = self[key] / total


def raiseNotDefined():
    fileName = inspect.stack()[1][1]
    line = inspect.stack()[1][2]
    method = inspect.stack()[1][3]

    print("*** Method not implemented: %s at line %s of %s" %
          (method, line, fileName))
    sys.exit(1)
```

- incrementAll method just increments all elements of the key by the same count
- totalCount method returns the sum of the counts for all keys
- normalize method adjusts the counts of all keys in a counter so that the total count of all keys sums to 1. It divides each count by the total count and updates the counter accordingly. If the counter is empty, it returns without making any changes.

- raiseNotDefined method is used as a placeholder for methods that have not been implemented yet. When called, it prints a message indicating which method is not implemented, along with the file name and line number where the call to raiseNotDefined occurred. Then it exits the program with a status code of 1, indicating an error.

```
def normalize(vectorOrCounter):
    """
    Normalize a vector or counter by dividing each value by the sum of all values
    """
    normalizedCounter = Counter()
    if type(vectorOrCounter) == type(normalizedCounter):
        counter = vectorOrCounter
        total = float(counter.totalCount())
        if total == 0:
            return counter
        for key in list(counter.keys()):
            value = counter[key]
            normalizedCounter[key] = value / total
        return normalizedCounter
    else:
        vector = vectorOrCounter
        s = float(sum(vector))
        if s == 0:
            return vector
        return [el / s for el in vector]


def sample(distribution, values=None):
    if type(distribution) == Counter:
        items = sorted(distribution.items())
        distribution = [i[1] for i in items]
        values = [i[0] for i in items]
    if sum(distribution) != 1:
        distribution = normalize(distribution)
    choice = random.random()
    i, total = 0, distribution[0]
    while choice > total:
        i += 1
        total += distribution[i]
    return values[i]


def getProbability(value, distribution, values):
    """
    Gives the probability of a value under a discrete distribution
    defined by (distributions, values).
    """
    total = 0.0
    for prob, val in zip(distribution, values):
        if val == value:
            total += prob
    return total
```

- The function normalize takes either a vector or a counter as input and returns a normalized version of it.
    - If the input is a counter, it calculates the sum of all counts (totalCount) and then divides each count by the total sum to normalize it. It creates a new counter with normalized counts and returns it.
    - If the input is a vector, it calculates the sum of all elements in the vector. Then, it divides each element by the sum to normalize it. It returns a new list containing the normalized values.

- The sample function randomly selects a value from a given distribution. It accepts either a Counter object or a list representing the probabilities of values. It ensures the distribution sums to 1 and then selects a value based on the probabilities.
    - If the input distribution is a Counter, it extracts the values and probabilities from the Counter and stores them in values and distribution lists respectively.
    - If the sum of probabilities in the distribution is not equal to 1, it normalizes the distribution.
    - It generates a random number choice between 0 and 1.
    - It iterates through the distribution to find the index i where the cumulative sum of probabilities exceeds choice. This index corresponds to the selected value.
    - It returns the value corresponding to the selected index i.

- getProbability function calculates the probability of a given value occurring in a discrete distribution defined by distribution and values. It iterates through each pair of probability and corresponding value in the distribution. If the value matches the provided value, it adds the probability associated with that value to the total probability. Finally, it returns the total probability for the specified value.

```python
def chooseFromDistribution(distribution):
    "Takes either a counter or a list of (prob, key) pairs and samples"
    if type(distribution) == dict or type(distribution) == Counter:
        return sample(distribution)
    r = random.random()
    base = 0.0
    for prob, element in distribution:
        base += prob
        if r <= base:
            return element

def nearestPoint(pos):
    """
    Finds the nearest grid point to a position (discretizes).
    """
    (current_row, current_col) = pos

    grid_row = int(current_row + 0.5)
    grid_col = int(current_col + 0.5)
    return (grid_row, grid_col)
```

- chooseFromDistribution function selects an item from a distribution. It can take either a counter or a list of (probability, item) pairs. If the input is a counter or a dictionary, it uses the sample function to sample from it. Otherwise, if the input is a list of (probability, item) pairs, it iterates through each pair, accumulating the probabilities until the cumulative probability exceeds a randomly chosen value r. When this happens, it returns the corresponding item.
- nearestPoint function takes a position represented by coordinates and discretizes it to find the nearest grid point. It achieves this by rounding the current row and column coordinates to the nearest integer using the int() function. This process effectively maps continuous positions to discrete grid points

```python
class TimeoutFunction:
    def __init__(self, function, timeout):
        self.timeout = timeout
        self.function = function

    def handle_timeout(self, signum, frame):
        raise TimeoutFunctionException()

    def __call__(self, *args, **keyArgs):
        # If we have SIGALRM signal, use it to cause an exception if and
        # when this function runs too long.  Otherwise check the time taken
        # after the method has returned, and throw an exception then.
        if hasattr(signal, 'SIGALRM'):
            old = signal.signal(signal.SIGALRM, self.handle_timeout)
            signal.alarm(self.timeout)
            try:
                result = self.function(*args, **keyArgs)
            finally:
                signal.signal(signal.SIGALRM, old)
            signal.alarm(0)
        else:
            startTime = time.time()
            result = self.function(*args, **keyArgs)
            timeElapsed = time.time() - startTime
            if timeElapsed >= self.timeout:
                self.handle_timeout(None, None)
        return result
```

- TimeoutFunction class is designed to wrap another function and enforce a time limit on its execution. It takes two parameters during initialization: function, which is the function to be executed, and timeout, which specifies the maximum time allowed for the function to run before it's forcefully terminated.
- Handle_timeout method is a signal handler designed to respond to a timeout event triggered by the operating system. It takes two parameters: signum, which represents the signal number, and frame, which represents the current stack frame at the time of the signal. When called, this function raises a TimeoutFunctionException, indicating that the function being executed has exceeded its allotted time and the timeout condition has been reached
- __call__ method serves as the main entry point for the TimeoutFunction class. It allows instances of the class to be called like functions. When invoked, it first checks if the SIGALRM signal is available, which is used for setting an alarm clock. If available, it sets up a signal handler to catch the SIGALRM signal, which is triggered when a timeout occurs. Then it sets the alarm to the specified timeout duration. Next, it executes the wrapped function (self.function) with the provided arguments and keyword arguments (*args and **keyArgs). If the operation completes before the timeout, it cancels the alarm and returns the result. If the operation takes longer than the timeout, it raises a TimeoutFunctionException. If the SIGALRM signal is not available, it manually measures the execution time using the time module and raises a timeout exception if needed. Finally, it returns the result of the wrapped function. This mechanism ensures that functions can be executed with a timeout limit, preventing them from running indefinitely.

```
_ORIGINAL_STDOUT = None
_ORIGINAL_STDERR = None
_MUTED = False


def mutePrint():
    global _ORIGINAL_STDOUT, _ORIGINAL_STDERR, _MUTED
    if _MUTED:
        return
    _MUTED = True

    _ORIGINAL_STDOUT = sys.stdout
    #_ORIGINAL_STDERR = sys.stderr
    sys.stdout = WritableNull()
    #sys.stderr = WritableNull()


def unmutePrint():
    global _ORIGINAL_STDOUT, _ORIGINAL_STDERR, _MUTED
    if not _MUTED:
        return
    _MUTED = False

    sys.stdout = _ORIGINAL_STDOUT
    #sys.stderr = _ORIGINAL_STDERR

```

- mutePrint function globally redirects standard output (stdout) to a null device, effectively suppressing any printed output. It does this by replacing sys.stdout with an instance of a custom class WritableNull, which is essentially a file-like object that discards anything written to it. Additionally, it keeps track of the original stdout stream in _ORIGINAL_STDOUT and sets _MUTED flag to True to indicate that the printing has been muted.
- unmutePrint function reverses the action performed by mutePrint() by restoring the original standard output streams. If the printing is currently muted (_MUTED is True), it sets _MUTED to False to indicate that printing is no longer muted. Then it restores the original stdout stream by assigning _ORIGINAL_STDOUT back to sys.stdout

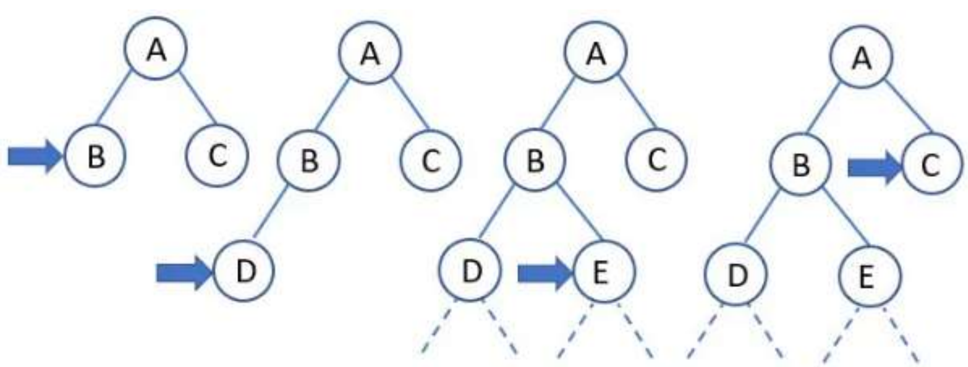
11) multiagent.py File:-

```python
class ReflexAgent(Agent):
    """
    A reflex agent chooses an action at each choice point by examining
    its alternatives via a state evaluation function.
    """

    def goalTest(self, gs, pos, flag):
        # Testing for goals
        if(flag == 0):
            if(gs.hasFood(pos[0], pos[1])):
                return True
            return False
        if(flag == 1):
            gpos = gs.getGhostPositions()
            for gp in gpos:
                if(gp == pos):
                    return True
            return False


    def DLS(self, currentNode, stack, explored, layer, limit, found, flag):
        # Depth Limited Search
        explored.append(currentNode)
        if(self.goalTest(currentNode.parent.state, currentNode.state.getPacmanPosition(), flag)):
            stack.push(currentNode)
            return stack, explored, True
        if(layer == limit):
            return stack, explored, False
        stack.push(currentNode)
        actions = currentNode.state.getLegalActions()
        for a in actions:
            newState = currentNode.state.generatePacmanSuccessor(a)
            newNode = Node(newState, currentNode, a, 1)
            if newNode in explored:
                continue
            stack, explored, found = self.DLS(newNode, stack, explored, layer+1, limit, found, flag)
            if(found):
                return stack, explored, True
        stack.pop()
        return stack, explored, False
```

This file simply only has ReflexAgent class which is an agent for pacman and the goalTest tests for the goal state

- here is how the goalTest function works:-
  - If flag is 0, the goal state is achieved if there is food at the specified position.
  - If flag is 1, the goal state is achieved if there is a ghost at the specified position.

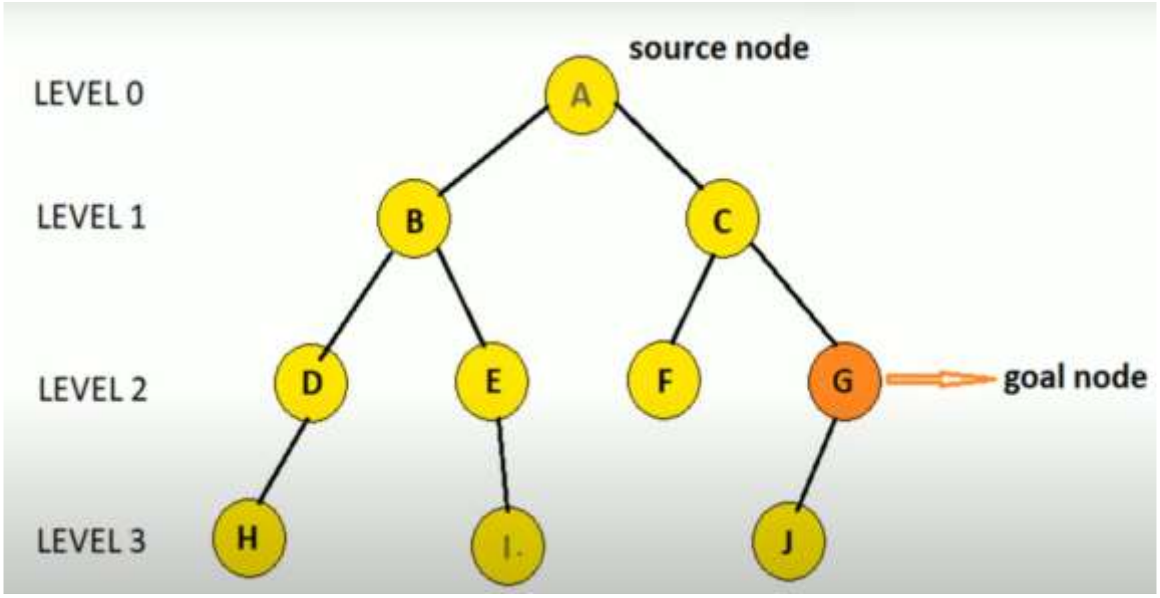
- Here is how DLS works:-



- If we fix the depth limit to 2, DLS can be carried out similarly to the DFS until the goal node is found to exist in the tree's search domain.
- This is implemented using a stack in a similar way to DFS

```python
    def IDS(self, sgs, limit, flag):
        # Iterative Deepening Search
        found = False
        current_limit = 0
        while(not found and current_limit <= limit):
            current_limit = current_limit + 1
            startNode = Node(sgs, None, None, 0)
            startNode.parent = startNode
            stack = Stack()
            explored = []
            stack, explored, found = self.DLS(startNode, stack, explored, 1, current_limit, False, flag)

        actions = []
        while(not stack.isEmpty()):
            node = stack.pop()
            actions.append(node.action)

        if not actions:
            return actions, found

        actions.reverse()
        actions.pop(0)  # Removes start node from actions

        return actions, found


    def getAction(self, gameState):
        """
        Choose an action based on evaluation function.
        """
        legalMoves = gameState.getLegalActions()

        weights = np.loadtxt("weights.csv", delimiter=",")

        scores = []
        for action in legalMoves:
            s = self.evaluationFunction(gameState, action, weights)
            scores.append(s)

        bestScore = max(scores)
        bestIndices = [index for index in range(len(scores)) if scores[index] == bestScore]
        chosenIndex = random.choice(bestIndices) # Pick randomly among the best

        return legalMoves[chosenIndex]
```

- Let’s show how the IDS works with an example:-



- Let's say we have this tree with the goal node being G, we'll start with depth limit of 0, and then it will only search A which is not the goal node so we increment the depth limit and now it's 1 and then it will do the following search A => B =>C then it hasn't reached the goal state so it will increment the depth limit again and now it's 2 then it will do the following search A=>B=>D=>E=>C=>F=>G and it reached it's goal state so then it will stop.

- Here's how the getAction function works:-

- This function implements a simple decision-making strategy for the AI agent based on evaluating each possible action and selecting the one deemed most favourable according to a set of predefined or learned weights in the weights.csv file.

```
    def CalcGhostPos(self, cgs, actions):
        # Calculate ghost position
        for a in actions:
            cgs = cgs.generatePacmanSuccessor(a)
        return cgs.getPacmanPosition()

    def findAllGhosts(self, cgs):
        # Find all active and scared ghosts and then turn them into binary features
        f1 = 0  # Active ghost one step away (Binary)
        f2 = 0  # Active ghost two steps away (Binary)
        f3 = 0  # Scared ghost one step away (Binary)
        f4 = 0  # Scared ghost two steps away (Binary)
        actions, found = self.IDS(cgs, 3, 1)
        if not found:
            return f1, f2, f3, f4
        ghosts = cgs.getGhostStates()
        ghostPos = self.CalcGhostPos(cgs, actions)
        foundGhostPosition = False
        for g in ghosts:
            if(ghostPos == g.configuration.pos):
                ghost = g
                foundGhostPosition = True
                break

        if not foundGhostPosition:
            return f1, f2, f3, f4

        if(ghost.scaredTimer > 0):  # If ghost is scared
            if(len(actions) <= 1):
                f3 = 1
            if(len(actions) == 2):
                f4 = 1
        if(ghost.scaredTimer == 0): # If ghost is active
            if(len(actions) <= 1):
                f1 = 1
            if(len(actions) == 2):
                f2 = 1

        return f1, f2, f3, f4
```

- Here’s how calcGhostPos function works:-
  - The function iterates over each action in the actions sequence.
  - For each action, it generates a successor game state (cgs.generatePacmanSuccessor(a)). This means it simulates Pacman's movement according to the action a and updates the game state accordingly.
  - After applying all the actions, the function retrieves the position of Pacman using cgs.getPacmanPosition()
  - The function returns the final position of the ghost after applying the sequence of actions.

- Here's how findAllGhosts function works:-
  - This function performs a depth-limited search to identify ghosts in the game state and extracts relevant information about their status (active or scared) and proximity to Pacman, represented by binary features.
  - Binary features:
    - f1: Active ghost one step away.
    - f2: Active ghost two steps away.
    - f3: Scared ghost one step away.
    - f4: Scared ghost two steps away.

```python
    def getFeatureFive(self, cgs, sgs):
        # Eating Food (Binary)
        if(self.goalTest(cgs, sgs.getPacmanPosition(), 0)):
            return 1
        return 0

    def getFeatureSix(self, cgs):
        # Distance to closest food
        food = cgs.getFood()
        pacPos = cgs.getPacmanPosition()
        dist = []
        x_size = food.width
        y_size = food.height
        for x in range(0, x_size):
            for y in range(0, y_size):
                if(food[x][y] == True):
                    dist.append(manhattanDistance(pacPos, (x,y)))
        if not dist:
            return 0
        closestFood = min(dist)
        return 1/closestFood

    def evaluationFunction(self, currentGameState, action, weights):
        # Evaluation function for choosing actions
        successorGameState = currentGameState.generatePacmanSuccessor(action)

        f1, f2, f3, f4 = self.findAllGhosts(successorGameState)
        f5 = self.getFeatureFive(currentGameState, successorGameState)
        f6 = self.getFeatureSix(successorGameState)
        features = np.array([f1, f2, f3, f4, f5, f6])

        Q_s_a = np.dot(weights, np.transpose(features))

        return Q_s_a
```

- Here's how the getFeatureFive function works:-
  - This function essentially acts as a binary indicator of whether Pacman is currently positioned at a location with food

- Here's how the getFeatureSix function works:-
  - It calculates distance from pacman to every food in the grid and appends them in the dist list, and returns 1/closestFood if there is food on the grid otherwise it returns 0 indicating there's no food on the grid

- Here's how the evaluationFunction works:-
  - It generates the successor game state resulting from applying the action to the current game state.
  - It computes various features using helper functions like findAllGhosts, getFeatureFive, and getFeatureSix, based on the successor game state and possibly the current game state.
  - It combines these features into a feature vector.
  - It computes the dot product of the feature vector and the weights to obtain a single score, Q_s_a, representing the estimated value of taking the given action in the current state.
  - It returns this score.

12) Game.py File:-

```
class Agent:
    def __init__(self, index=0):
        self.index = index


class Directions:
    NORTH = 'North'
    SOUTH = 'South'
    EAST = 'East'
    WEST = 'West'
    STOP = 'Stop'

    LEFT = {NORTH: WEST,
            SOUTH: EAST,
            EAST:  NORTH,
            WEST:  SOUTH,
            STOP:  STOP}

    RIGHT = dict([(y, x) for x, y in list(LEFT.items())])

    REVERSE = {NORTH: SOUTH,
               SOUTH: NORTH,
               EAST: WEST,
               WEST: EAST,
               STOP: STOP}
```

- Class agent simply initializes the agent index
- Class Directions just handles the directions

```python
class Configuration:
    """
    A Configuration holds the (x,y) coordinate of a character, along with its
    traveling direction.

    The convention for positions, like a graph, is that (0,0) is the lower left corner, x increases
    horizontally and y increases vertically.  Therefore, north is the direction of increasing y, or (0,1).
    """


    def generateSuccessor(self, vector):
        """
        Generates a new configuration reached by translating the current
        configuration by the action vector.  This is a low-level call and does
        not attempt to respect the legality of the movement.

        Actions are movement vectors.
        """
        x, y = self.pos
        dx, dy = vector
        direction = Actions.vectorToDirection(vector)
        if direction == Directions.STOP:
            direction = self.direction  # There is no stop direction
        return Configuration((x + dx, y+dy), direction)
```

- generateSuccessor method calculates a new configuration based on the current configuration and a movement vector. It doesn't check if the movement is legal; rather, it simply applies the vector to the current position. The method takes the vector as input, calculates the new position by adding the vector components to the current position, and determines the new direction based on the vector. If the vector represents a stop action, it retains the current direction. Finally, it returns a new Configuration object with the updated position and direction.

```python
class AgentState:
    """
    AgentStates hold the state of an agent (configuration, speed, scared, etc).
    """

    def __init__(self, startConfiguration, isPacman):
        self.start = startConfiguration
        self.configuration = startConfiguration
        self.isPacman = isPacman
        self.scaredTimer = 0
        # state below potentially used for contest only
        self.numCarrying = 0
        self.numReturned = 0

    def __str__(self):
        if self.isPacman:
            return "Pacman: " + str(self.configuration)
        else:
            return "Ghost: " + str(self.configuration)

    def __eq__(self, other):
        if other == None:
            return False
        return self.configuration == other.configuration and self.scaredTimer == other.scaredTimer

    def __hash__(self):
        return hash(hash(self.configuration) + 13 * hash(self.scaredTimer))

    def copy(self):
        state = AgentState(self.start, self.isPacman)
        state.configuration = self.configuration
        state.scaredTimer = self.scaredTimer
        state.numCarrying = self.numCarrying
        state.numReturned = self.numReturned
        return state

    def getPosition(self):
        if self.configuration == None:
            return None
        return self.configuration.getPosition()

    def getDirection(self):
        return self.configuration.getDirection()
```

- In class AgentState we first initialize the needed variables
- __str__ method returns a string representation of the object. In this context, it is used to provide human-readable descriptions of configurations, either for Pacman or ghosts, based on the attributes of the object.
- __eq__ method defines the equality comparison between two instances of the same class. It returns True if the configurations (positions and directions) of the two instances are the same, and their scaredTimer attributes are equal. If other is None, it returns False, indicating that self is not equal to None.
- __hash__ method is used to compute the hash value of an object, which is essential for the object's usability in hash-based data structures like dictionaries and sets. In this implementation, the hash value is computed based on the hash values of the configuration attribute (representing the position and direction) and the scaredTimer attribute. The hash values are combined using addition and multiplication to generate a final hash value for the object. This ensures that objects with similar configurations and scared timers will have similar hash values, aiding in efficient storage and retrieval in hash-based collections.

- Copy method just creates a copy of the state
- getPosition method retrieves the position of the agent. It first checks if the agent's configuration is None, indicating that its position is not defined. If the configuration is not None, it calls the getPosition method of the configuration object to retrieve the position of the agent. Finally, it returns the position of the agent.

```python
class Grid:
    """
    A 2-dimensional array of objects backed by a list of lists.  Data is accessed
    via grid[x][y] where (x,y) are positions on a Pacman map with x horizontal,
    y vertical and the origin (0,0) in the bottom left corner.

    The __str__ method constructs an output that is oriented like a pacman board.
    """

    def __init__(self, width, height, initialValue=False, bitRepresentation=None):
        if initialValue not in [False, True]:
            raise Exception('Grids can only contain booleans')
        self.CELLS_PER_INT = 30

        self.width = width
        self.height = height
        self.data = [[initialValue for y in range(
            height)] for x in range(width)]
        if bitRepresentation:
            self._unpackBits(bitRepresentation)

    def __getitem__(self, i):
        return self.data[i]

    def __setitem__(self, key, item):
        self.data[key] = item

    def __str__(self):
        out = [[str(self.data[x][y])[0] for x in range(self.width)]
               for y in range(self.height)]
        out.reverse()
        return '\n'.join([''.join(x) for x in out])

    def __eq__(self, other):
        if other == None:
            return False
        return self.data == other.data

    def __hash__(self):
        # return hash(str(self))
        base = 1
        h = 0
        for l in self.data:
            for i in l:
                if i:
                    h += base
                base *= 2
        return hash(h)
```

```python
    def packBits(self):
        """
        Returns an efficient int list representation

        (width, height, bitPackedInts...)
        """
        bits = [self.width, self.height]
        currentInt = 0
        for i in range(self.height * self.width):
            bit = self.CELLS_PER_INT - (i % self.CELLS_PER_INT) - 1
            x, y = self._cellIndexToPosition(i)
            if self[x][y]:
                currentInt += 2 ** bit
            if (i + 1) % self.CELLS_PER_INT == 0:
                bits.append(currentInt)
                currentInt = 0
        bits.append(currentInt)
        return tuple(bits)

    def _cellIndexToPosition(self, index):
        x = index / self.height
        y = index % self.height
        return x, y

    def _unpackBits(self, bits):
        """
        Fills in data from a bit-level representation
        """
        cell = 0
        for packed in bits:
            for bit in self._unpackInt(packed, self.CELLS_PER_INT):
                if cell == self.width * self.height:
                    break
                x, y = self._cellIndexToPosition(cell)
                self[x][y] = bit
                cell += 1
```

- packBits method returns an efficient integer list representation of the current layout. It begins by storing the width and height of the layout in a list called bits. Then, it iterates over each cell in the layout. For each cell, it calculates the bit position within the packed integer based on the cell's index, width, and height. If the cell is present (i.e., it's not a wall), it sets the corresponding bit in the current integer. When the current integer is filled with bits representing cells or reaches the maximum capacity, it's appended to the bits list, and a new integer is started. Finally, the method returns a tuple containing the width, height, and the list of packed integers representing the layout. The _cellIndexToPosition method is a helper function that converts a cell index to its corresponding (x, y) position on the layout.
- _cellIndexToPosition method converts a one-dimensional index representing a cell in a grid to its two-dimensional (x, y) position in the grid. It does this by dividing the index by the height of the grid to determine the row (x-coordinate) and taking the remainder of the index divided by the height to determine the column (y-coordinate). Finally, it returns the (x, y) position tuple.

- _unpackBits method is responsible for filling in data from a bit-level representation into a grid. It iterates over each packed integer in the input bits, which represents the grid cells. For each packed integer, it calls the _unpackInt method to extract individual bits. It then assigns these bits to the corresponding cells in the grid, using the _cellIndexToPosition method to determine the position of each cell based on its index. The process continues until all cells in the grid are filled.

```
    def _unpackInt(self, packed, size):
        bools = []
        if packed < 0:
            raise ValueError("must be a positive integer")
        for i in range(size):
            n = 2 ** (self.CELLS_PER_INT - i - 1)
            if packed >= n:
                bools.append(True)
                packed -= n
            else:
                bools.append(False)
        return bools
```

- _unpackInt method takes a packed integer and a size parameter, which specifies the number of bits to unpack. It iterates over each bit position in the packed integer, starting from the most significant bit (MSB) to the least significant bit (LSB). For each bit position, it checks if the corresponding bit is set by comparing it with the powers of 2 from the highest to the lowest. If the bit is set, it appends True to the bools list; otherwise, it appends False. Finally, it returns the list of boolean values representing the unpacked bits.

```python
class Actions:
    """
    A collection of static methods for manipulating move actions.
    """
    # Directions
    _directions = {Directions.WEST:  (-1, 0),
                   Directions.STOP:  (0, 0),
                   Directions.EAST:  (1, 0),
                   Directions.NORTH: (0, 1),
                   Directions.SOUTH: (0, -1)}

    _directionsAsList = [('West', (-1, 0)), ('Stop', (0, 0)), ('East', (1, 0)), ('North', (0, 1)), ('South', (0, -1))]

    TOLERANCE = .001

    def reverseDirection(action):
        if action == Directions.NORTH:
            return Directions.SOUTH
        if action == Directions.SOUTH:
            return Directions.NORTH
        if action == Directions.EAST:
            return Directions.WEST
        if action == Directions.WEST:
            return Directions.EAST
        return action
    reverseDirection = staticmethod(reverseDirection)

    def vectorToDirection(vector):
        dx, dy = vector
        if dy > 0:
            return Directions.NORTH
        if dy < 0:
            return Directions.SOUTH
        if dx < 0:
            return Directions.WEST
        if dx > 0:
            return Directions.EAST
        return Directions.STOP
    vectorToDirection = staticmethod(vectorToDirection)

    def directionToVector(direction, speed=1.0):
        dx, dy = Actions._directions[direction]
        return (dx * speed, dy * speed)
    directionToVector = staticmethod(directionToVector)
```

- Class Actions we first initialize the directions as a fist and as a list
- reverseDirection method just updates the directions if it’s in reverse
- vectorToDirection method takes a 2D vector as input and returns a direction based on the vector's components. If the vertical component (dy) of the vector is positive, it returns Directions.NORTH; if dy is negative, it returns Directions.SOUTH. Similarly, if the horizontal component (dx) is negative, it returns Directions.WEST, and if dx is positive, it returns Directions.EAST. If both dx and dy are zero, indicating no movement, it returns Directions.STOP
- directionToVector method takes a direction and an optional speed as input and returns a corresponding 2D vector. It retrieves the corresponding components of the direction from the _directions dictionary attribute and scales them by the given speed. If no speed is provided, it defaults to 1.0.

```python
    def getPossibleActions(config, walls):
        possible = []
        x, y = config.pos
        x_int, y_int = int(x + 0.5), int(y + 0.5)

        # In between grid points, all agents must continue straight
        if (abs(x - x_int) + abs(y - y_int) > Actions.TOLERANCE):
            return [config.getDirection()]

        for dir, vec in Actions._directionsAsList:
            dx, dy = vec
            next_y = y_int + dy
            next_x = x_int + dx
            if not walls[next_x][next_y]:
                possible.append(dir)

        return possible

    getPossibleActions = staticmethod(getPossibleActions)

    def getLegalNeighbors(position, walls):
        x, y = position
        x_int, y_int = int(x + 0.5), int(y + 0.5)
        neighbors = []
        for dir, vec in Actions._directionsAsList:
            dx, dy = vec
            next_x = x_int + dx
            if next_x < 0 or next_x == walls.width:
                continue
            next_y = y_int + dy
            if next_y < 0 or next_y == walls.height:
                continue
            if not walls[next_x][next_y]:
                neighbors.append((next_x, next_y))
        return neighbors
    getLegalNeighbors = staticmethod(getLegalNeighbors)

    def getSuccessor(position, action):
        dx, dy = Actions.directionToVector(action)
        x, y = position
        return (x + dx, y + dy)
    getSuccessor = staticmethod(getSuccessor)
```

- getPossibleActions method determines the possible actions that can be taken from a given configuration, considering the layout's walls. It starts by initializing an empty list to store the possible actions. Then, it checks if the current position of the agent lies between grid points; if so, the agent must continue straight, and the function returns a list containing the current direction. Otherwise, it iterates over each direction and its corresponding vector. For each direction, it calculates the next position by adding the vector components to the current position. If the next position is not obstructed by a wall, the direction is added to the list of possible actions. Finally, the function returns the list of possible actions.

- getLegalNeighbors method determines the legal neighboring positions of a given position, considering the layout's walls. It takes the position of the agent and the walls layout as input. It starts by initializing an empty list to store the legal neighboring positions. Then, it iterates over each direction and its corresponding vector. For each direction, it calculates the next position by adding the vector components to the current position. If the next position is within the bounds of the walls layout and is not obstructed by a wall, it is considered a legal neighboring position and added to the list of neighbors. Finally, the function returns the list of legal neighboring positions.

- getSuccessor method computes the successor position resulting from applying a given action to a current position. It takes two arguments: the current position as a tuple (x, y) and the action to be applied. First, it converts the action into a vector using the Actions.directionToVector method. Then, it computes the new position by adding the vector components to the current position. Finally, it returns the new position as a tuple (x_new, y_new). This method is declared as a static method using the staticmethod decorator.

```
class GameStateData:

    def __init__(self, prevState=None):
        """
        Generates a new data packet by copying information from its predecessor.
        """
        if prevState != None:
            self.food = prevState.food.shallowCopy()
            self.capsules = prevState.capsules[:]
            self.agentStates = self.copyAgentStates(prevState.agentStates)
            self.layout = prevState.layout
            self._eaten = prevState._eaten
            self.score = prevState.score

        self._foodEaten = None
        self._foodAdded = None
        self._capsuleEaten = None
        self._agentMoved = None
        self._lose = False
        self._win = False
        self.scoreChange = 0

    def deepCopy(self):
        state = GameStateData(self)
        state.food = self.food.deepCopy()
        state.layout = self.layout.deepCopy()
        state._agentMoved = self._agentMoved
        state._foodEaten = self._foodEaten
        state._foodAdded = self._foodAdded
        state._capsuleEaten = self._capsuleEaten
        return state

    def copyAgentStates(self, agentStates):
        copiedStates = []
        for agentState in agentStates:
            copiedStates.append(agentState.copy())
        return copiedStates
```

- GameStateData handles the states of the game, we first initialize the needed variables
- deepCopy method creates a copy of the state
- copyAgentStates creates a copy of the agent states

```
    def _foodWallStr(self, hasFood, hasWall):
        if hasFood:
            return '.'
        elif hasWall:
            return '%'
        else:
            return ' '

    def _pacStr(self, dir):
        if dir == Directions.NORTH:
            return 'v'
        if dir == Directions.SOUTH:
            return '^'
        if dir == Directions.WEST:
            return '>'
        return '<'

    def _ghostStr(self, dir):
        return 'G'


    def initialize(self, layout, numGhostAgents):
        """
        Creates an initial game state from a layout array (see layout.py).
        """
        self.food = layout.food.copy()
        #self.capsules = []
        self.capsules = layout.capsules[:]
        self.layout = layout
        self.score = 0
        self.scoreChange = 0

        self.agentStates = []
        numGhosts = 0
        for isPacman, pos in layout.agentPositions:
            if not isPacman:
                if numGhosts == numGhostAgents:
                    continue  # Max ghosts reached already
                else:
                    numGhosts += 1
            self.agentStates.append(AgentState(
                Configuration(pos, Directions.STOP), isPacman))
        self._eaten = [False for a in self.agentStates]
```

- _foodWallStr just puts . where food are, % for wall otherwise just a space
- _pacStr just has pacman directions as string
- _ghostStr just makes G represent ghost
- Initalizie method sets up the initial state of a game based on a given layout array. It copies information about the layout, food, capsules, and agent positions from the input layout. It initializes attributes such as food, capsules, layout, score, scoreChange, agentStates, and _eaten. For each agent position specified in the layout, it creates an AgentState object with the corresponding configuration and adds it to the agentStates list. If the agent is a ghost and the maximum number of ghost agents has been reached, it skips adding more ghost agents.

```python
class Game:
    """
    The Game manages the control flow, soliciting actions from agents.
    """

    def __init__(self, agents, display, rules, startingIndex=0, muteAgents=False, catchExceptions=False):
        self.agentCrashed = False
        self.agents = agents
        self.display = display
        self.rules = rules
        self.startingIndex = startingIndex
        self.gameOver = False
        self.muteAgents = muteAgents
        self.catchExceptions = catchExceptions
        self.moveHistory = []
        self.totalAgentTimes = [0 for agent in agents]
        self.totalAgentTimeWarnings = [0 for agent in agents]
        self.agentTimeout = False
        import io
        self.agentOutput = [io.StringIO() for agent in agents]

    def getProgress(self):
        if self.gameOver:
            return 1.0
        else:
            return self.rules.getProgress(self)

    def _agentCrash(self, agentIndex, quiet=False):
        "Helper method for handling agent crashes"
        if not quiet:
            traceback.print_exc()
        self.gameOver = True
        self.agentCrashed = True
        self.rules.agentCrash(self, agentIndex)

    OLD_STDOUT = None
    OLD_STDERR = None

    def mute(self, agentIndex):
        if not self.muteAgents:
            return
        global OLD_STDOUT, OLD_STDERR
        import io
        OLD_STDOUT = sys.stdout
        OLD_STDERR = sys.stderr
        sys.stdout = self.agentOutput[agentIndex]
        sys.stderr = self.agentOutput[agentIndex]
```

- Now we’re in the class game, we firstly initialize the needed variables
- getProgress method checks if the game is over (gameOver is True), it returns a progress value of 1.0, indicating that the game is completed. Otherwise, it delegates the calculation of progress to the getProgress method of the game rules (self.rules.getProgress(self)
- _agentCrash is just for handling if any agent crashes
- Mute method is responsible for muting the output of agents during the game. It first checks if muting agents is enabled (muteAgents attribute). If muting is enabled, it redirects the standard output (sys.stdout) and standard error (sys.stderr) streams to an instance of io.StringIO stored in the agentOutput list at the specified agentIndex. This redirection effectively suppresses the output produced by the agent during its execution.

```
    def unmute(self):
        if not self.muteAgents:
            return
        global OLD_STDOUT, OLD_STDERR
        # Revert stdout/stderr to originals
        sys.stdout = OLD_STDOUT
        sys.stderr = OLD_STDERR
```

- Unmute method reverts the standard output (sys.stdout) and standard error (sys.stderr) streams back to their original values. It first checks if muting agents is enabled (muteAgents attribute). If muting is enabled, it restores the original standard output and standard error streams by assigning them to the variables OLD_STDOUT and OLD_STDERR, respectively. This allows normal printing to the console to resume after muting.

13) pacman.py File:-

```
class GameState:
    """
    A GameState specifies the full game state, including the food, capsules,
    agent configurations and score changes.

    GameStates are used by the Game object to capture the actual state of the game and
    can be used by agents to reason about the game.

    Much of the information in a GameState is stored in a GameStateData object.  We
    strongly suggest that you access that data via the accessor methods below rather
    than referring to the GameStateData object directly.

    Note that in classic Pacman, Pacman is always agent 0.
    """

    ####################################################
    # Accessor methods: use these to access state data #
    ####################################################

    # static variable keeps track of which states have had getLegalActions called
    explored = set()

    def getAndResetExplored():
        tmp = GameState.explored.copy()
        GameState.explored = set()
        return tmp
    getAndResetExplored = staticmethod(getAndResetExplored)

    def getLegalActions(self, agentIndex=0):
        """
        Returns the legal actions for the agent specified.
        """
#         GameState.explored.add(self)
        if self.isWin() or self.isLose():
            return []

        if agentIndex == 0:  # Pacman is moving
            return PacmanRules.getLegalActions(self)
        else:
            return GhostRules.getLegalActions(self, agentIndex)
```

- First we have the game state class, where we start by having a set of explored states
- getAndResetExplored method is a static method used to retrieve and reset a class-level variable called explored in the GameState class. Here's what it does:

  - It creates a copy of the explored set using the copy() method. This ensures that the original set remains unchanged even after resetting.
  - It then resets the explored set to an empty set using the set() function. This clears all previously explored states.
  - Finally, it returns the copied set of explored states.

- getLegalActions method returns the legal actions that an agent specified by the agentIndex parameter can take in the current game state. Here's a breakdown of its functionality:
  - Check for Win or Lose: It checks if the game state represents a winning or losing state. If so, it returns an empty list, indicating that there are no legal actions to take from a terminal state.
  - Determine Legal Actions:
    - If agentIndex is 0, it implies that Pacman is the agent currently making a move. In this case, it calls PacmanRules.getLegalActions(self), likely a method specific to handling legal actions for Pacman.
    - If agentIndex is not 0, it suggests that a ghost agent specified by the index is making a move. It then calls GhostRules.getLegalActions(self, agentIndex), which likely retrieves legal actions specifically tailored for ghost agents.

```python
    def generateSuccessor(self, agentIndex, action):
        """
        Returns the successor state after the specified agent takes the action.
        """
        # Check that successors exist
        if self.isWin() or self.isLose():
            raise Exception('Can\'t generate a successor of a terminal state.')

        # Copy current state
        state = GameState(self)

        # Let agent's logic deal with its action's effects on the board
        if agentIndex == 0:  # Pacman is moving
            state.data._eaten = [False for i in range(state.getNumAgents())]
            PacmanRules.applyAction(state, action)
        else:                # A ghost is moving
            GhostRules.applyAction(state, action, agentIndex)

        # Time passes
        if agentIndex == 0:
            state.data.scoreChange += -TIME_PENALTY  # Penalty for waiting around
        else:
            GhostRules.decrementTimer(state.data.agentStates[agentIndex])

        # Resolve multi-agent effects
        GhostRules.checkDeath(state, agentIndex)

        # Book keeping
        state.data._agentMoved = agentIndex
        state.data.score += state.data.scoreChange
        GameState.explored.add(self)
        GameState.explored.add(state)
        return state

    def getLegalPacmanActions(self):
        return self.getLegalActions(0)

    def generatePacmanSuccessor(self, action):
        """
        Generates the successor state after the specified pacman move
        """
        return self.generateSuccessor(0, action)
```

- generateSuccessor method calculates the successor state after an agent specified by agentIndex takes a particular action action. Below is a breakdown of its functionality:
  - Check for Terminal State: It first checks if the current state is a terminal state (win or lose). If it is, it raises an exception indicating that successors cannot be generated from terminal states.
  - Copy Current State: It then creates a copy of the current game state to make modifications without altering the original state.
  - Apply Action Effects:
    - ➢ If agentIndex is 0, indicating that Pacman is making a move, it initializes a list _eaten to track if Pacman has eaten any food pellets and applies Pacman's action using PacmanRules.applyAction(state, action).
    - ➢ If agentIndex is not 0, implying that a ghost agent is making a move, it applies the action using GhostRules.applyAction(state, action, agentIndex).
  - Time Passage:
    - ➢ If agentIndex is 0, Pacman's move incurs a time penalty by decrementing the score (state.data.scoreChange) by TIME_PENALTY.
    - ➢ If it's a ghost's move, the timer for the ghost's state is decremented.

- Resolve Multi-Agent Effects: It checks if any agents have died due to interactions, particularly relevant for ghosts, using GhostRules.checkDeath(state, agentIndex).
- Bookkeeping:
  - Updates the _agentMoved attribute of the state to indicate which agent made the move.
  - Adjusts the overall score based on any changes (state.data.score += state.data.scoreChange).
  - Adds both the current state and the generated successor state to a set called explored, which likely tracks previously visited states.
- Return: Finally, it returns the modified successor state.

- getLegalPacmanAction method retrieves the legal actions that the Pacman agent can take in the current game state. It achieves this by calling the getLegalActions method with agentIndex set to 0
- generatePacmanSuccessor method generates a successor state after the specified Pacman move. It achieves this by calling the generateSuccessor method with agentIndex set to 0, indicating that it's Pacman who is moving, and providing the action chosen by Pacman.

```python
    def getPacmanState(self):
        """
        Returns an AgentState object for pacman (in game.py)

        state.pos gives the current position
        state.direction gives the travel vector
        """
        return self.data.agentStates[0].copy()

    def getPacmanPosition(self):
        return self.data.agentStates[0].getPosition()

    def getGhostStates(self):
        return self.data.agentStates[1:]

    def getGhostState(self, agentIndex):
        if agentIndex == 0 or agentIndex >= self.getNumAgents():
            raise Exception("Invalid index passed to getGhostState")
        return self.data.agentStates[agentIndex]

    def getGhostPosition(self, agentIndex):
        if agentIndex == 0:
            raise Exception("Pacman's index passed to getGhostPosition")
        return self.data.agentStates[agentIndex].getPosition()

    def getGhostPositions(self):
        return [s.getPosition() for s in self.getGhostStates()]

    def getNumAgents(self):
        return len(self.data.agentStates)

    def getScore(self):
        return float(self.data.score)

    def getCapsules(self):
        """
        Returns a list of positions (x,y) of the remaining capsules.
        """
        return self.data.capsules

    def getNumFood(self):
        return self.data.food.count()
```

- getPacmanState or getGhostState method just returns the AgentState object for pacman or ghosts
- getPacmanPosition or getGhostPosition method just returns the current state of pacman or ghosts

```
    def getFood(self):
        """
        Returns a Grid of boolean food indicator variables.

        Grids can be accessed via list notation, so to check
        if there is food at (x,y), just call

        currentFood = state.getFood()
        if currentFood[x][y] == True: ...
        """
        return self.data.food

    def getWalls(self):
        """
        Returns a Grid of boolean wall indicator variables.

        Grids can be accessed via list notation, so to check
        if there is a wall at (x,y), just call

        walls = state.getWalls()
        if walls[x][y] == True: ...
        """
        return self.data.layout.walls

    def hasFood(self, x, y):
        return self.data.food[x][y]

    def hasWall(self, x, y):
        return self.data.layout.walls[x][y]

    def isLose(self):
        return self.data._lose

    def isWin(self):
        return self.data._win
```

- getFood method Returns a Grid of boolean food indicator variables.
- getWalls Returns a Grid of boolean wall indicator variables.

```python
SCARED_TIME = 40    # Moves ghosts are scared
COLLISION_TOLERANCE = 0.7  # How close ghosts must be to Pacman to kill
TIME_PENALTY = 1  # Number of points lost each round


class ClassicGameRules:
    """
    These game rules manage the control flow of a game, deciding when
    and how the game starts and ends.
    """

    def __init__(self, timeout=30):
        self.timeout = timeout

    def newGame(self, layout, pacmanAgent, ghostAgents, display, quiet=False, catchExceptions=False):
        agents = [pacmanAgent] + ghostAgents[:layout.getNumGhosts()]
        initState = GameState()
        initState.initialize(layout, len(ghostAgents))
        game = Game(agents, display, self, catchExceptions=catchExceptions)
        game.state = initState
        self.initialState = initState.deepCopy()
        self.quiet = quiet
        return game

    def process(self, state, game):
        """
        Checks to see whether it is time to end the game.
        """
        if state.isWin():
            self.win(state, game)
        if state.isLose():
            self.lose(state, game)

    def win(self, state, game):
        if not self.quiet:
            print("Pacman emerges victorious! Score: %d" % state.data.score)
        game.gameOver = True

    def lose(self, state, game):
        if not self.quiet:
            print("Pacman died! Score: %d" % state.data.score)
        game.gameOver = True

    def getProgress(self, game):
        return float(game.state.getNumFood()) / self.initialState.getNumFood()
```

- First, we initialized the scared_time of the ghosts and their collision_tollerance and the time_penalty
- Then we have class ClassicGameRules where it handles the rules of the game, we first initialize the timeout
- newGame method creates a new instance of a Pacman game. It initializes the game with parameters such as the game layout, Pacman and ghost agents, display module, and optional settings. It sets up the initial state of the game, initializes a Game instance to manage the gameplay, stores the initial state, and returns the created Game instance for gameplay.
- Process method just checks whether to end the game or not
- Win method just checks if pacman wins
- Lose method just checks if pacman loses
- getProgress method just returns the progress done

```python
    def agentCrash(self, game, agentIndex):
        if agentIndex == 0:
            print("Pacman crashed")
        else:
            print("A ghost crashed")

    def getMaxTotalTime(self, agentIndex):
        return self.timeout

    def getMaxStartupTime(self, agentIndex):
        return self.timeout

    def getMoveWarningTime(self, agentIndex):
        return self.timeout

    def getMoveTimeout(self, agentIndex):
        return self.timeout

    def getMaxTimeWarnings(self, agentIndex):
        return 0
```

- These functions are just for handling crashes and timeouts

```python
class PacmanRules:
    """
    These functions govern how pacman interacts with his environment under
    the classic game rules.
    """
    PACMAN_SPEED = 1

    def getLegalActions(state):
        """
        Returns a list of possible actions.
        """
        return Actions.getPossibleActions(state.getPacmanState().configuration, state.data.layout.walls)
    getLegalActions = staticmethod(getLegalActions)

    def applyAction(state, action):
        """
        Edits the state to reflect the results of the action.
        """
        legal = PacmanRules.getLegalActions(state)
        if action not in legal:
            raise Exception("Illegal action " + str(action))

        pacmanState = state.data.agentStates[0]

        # Update Configuration
        vector = Actions.directionToVector(action, PacmanRules.PACMAN_SPEED)
        pacmanState.configuration = pacmanState.configuration.generateSuccessor(
            vector)

        # Eat
        next = pacmanState.configuration.getPosition()
        nearest = nearestPoint(next)
        if manhattanDistance(nearest, next) <= 0.5:
            # Remove food
            PacmanRules.consume(nearest, state)
    applyAction = staticmethod(applyAction)
```

- Class PacmanRules just handles the rules of pacman
- getLegalActions just returns a list of possible actions
- applyAction just edits the state to reflect the results of an action taken

```python
    def consume(position, state):
        x, y = position
        # Eat food
        if state.data.food[x][y]:
            state.data.scoreChange += 10
            state.data.food = state.data.food.copy()
            state.data.food[x][y] = False
            state.data._foodEaten = position
            numFood = state.getNumFood()

            if numFood == 0 and not state.data._lose:
                state.data.scoreChange += 500
                state.data._win = True

        # Eat capsule
        if(position in state.getCapsules()):
            state.data.capsules.remove(position)
            state.data._capsuleEaten = position

            # Reset all ghosts' scared timers
            for index in range(1, len(state.data.agentStates)):
                state.data.agentStates[index].scaredTimer = SCARED_TIME
    consume = staticmethod(consume)
```

- consume method just checks if pacman ate food or a capsule and resets ghosts scared timers if pacman does eat a capsule

```python
class GhostRules:
    """
    These functions dictate how ghosts interact with their environment.
    """
    GHOST_SPEED = 1.0

    def getLegalActions(state, ghostIndex):
        """
        Ghosts cannot stop, and cannot turn around unless they
        reach a dead end, but can turn 90 degrees at intersections.
        """
        conf = state.getGhostState(ghostIndex).configuration
        possibleActions = Actions.getPossibleActions(
            conf, state.data.layout.walls)
        reverse = Actions.reverseDirection(conf.direction)
        if Directions.STOP in possibleActions:
            possibleActions.remove(Directions.STOP)
        if reverse in possibleActions and len(possibleActions) > 1:
            possibleActions.remove(reverse)
        return possibleActions
    getLegalActions = staticmethod(getLegalActions)

    def applyAction(state, action, ghostIndex):

        legal = GhostRules.getLegalActions(state, ghostIndex)
        if action not in legal:
            raise Exception("Illegal ghost action " + str(action))

        ghostState = state.data.agentStates[ghostIndex]
        speed = GhostRules.GHOST_SPEED
        if ghostState.scaredTimer > 0:
            speed /= 2.0
        vector = Actions.directionToVector(action, speed)
        ghostState.configuration = ghostState.configuration.generateSuccessor(
            vector)
    applyAction = staticmethod(applyAction)

    def decrementTimer(ghostState):
        timer = ghostState.scaredTimer
        if timer == 1:
            ghostState.configuration.pos = nearestPoint(
                ghostState.configuration.pos)
        ghostState.scaredTimer = max(0, timer - 1)
    decrementTimer = staticmethod(decrementTimer)
```

- class GhostRules simply is for handling the rules of ghosts
- getLegalActions method determines the allowable actions for a ghost in the current game state. It considers the ghost's current configuration, such as its position and direction, and the layout of the game environment to determine possible actions. The function excludes actions that would cause the ghost to stop or reverse its direction. It returns a list of permissible actions for the specified ghost.
- applyAction method updates the game state when a ghost takes a specific action. It ensures the action's legality, considering the ghost's current state and game rules. If the action is legal, it calculates the ghost's movement vector based on its speed and the chosen action, then updates the ghost's configuration accordingly in the game state.
- decrementTimer just decrements the ghost scared timer when needed

```
    def checkDeath(state, agentIndex):
        pacmanPosition = state.getPacmanPosition()
        if agentIndex == 0:  # Pacman just moved; Anyone can kill him
            for index in range(1, len(state.data.agentStates)):
                ghostState = state.data.agentStates[index]
                ghostPosition = ghostState.configuration.getPosition()

                if GhostRules.canKill(pacmanPosition, ghostPosition):
                    GhostRules.collide(state, ghostState, index)
        else:
            ghostState = state.data.agentStates[agentIndex]
            ghostPosition = ghostState.configuration.getPosition()

            if GhostRules.canKill(pacmanPosition, ghostPosition):
                GhostRules.collide(state, ghostState, agentIndex)
    checkDeath = staticmethod(checkDeath)

    def collide(state, ghostState, agentIndex):
        if ghostState.scaredTimer > 0:
            state.data.scoreChange += 200
            GhostRules.placeGhost(state, ghostState)
            ghostState.scaredTimer = 0
            # Added for first-person
            state.data._eaten[agentIndex] = True
        else:
            if not state.data._win:
                state.data.scoreChange -= 500
                state.data._lose = True
    collide = staticmethod(collide)

    def canKill(pacmanPosition, ghostPosition):
        return manhattanDistance(ghostPosition, pacmanPosition) <= COLLISION_TOLERANCE
    canKill = staticmethod(canKill)

    def placeGhost(state, ghostState):
        ghostState.configuration = ghostState.start
    placeGhost = staticmethod(placeGhost)
```

- checkDeath method determines if any agent (either Pacman or ghost) has been killed in the game. It evaluates whether the positions of agents overlap and invokes collision handling mechanisms accordingly. If the agent in question is Pacman, it checks for collisions with ghosts and handles them appropriately. If the agent is a ghost, it checks for collisions with Pacman and manages the outcome accordingly.
- Collide method determines if a collision occurred between between pacman and ghost and whether the ghost was scared or not
- canKill checks if the ghost is within collision distance with pacman
- placeGhost just initializes the ghost position

```
def default(str):
    return str + ' [Default: %default]'


def parseAgentArgs(str):
    if str == None:
        return {}
    pieces = str.split(',')
    opts = {}
    for p in pieces:
        if '=' in p:
            key, val = p.split('=')
        else:
            key, val = p, 1
        opts[key] = val
    return opts
```

- default function takes a string argument and returns a modified string indicating that it represents a default value. It appends the text "[Default: %default]" to the input string
- parseAgentArgs function takes a string argument str, which represents a comma-separated list of key-value pairs. It splits the input string based on commas to extract individual pieces. Then, it iterates over each piece. If a piece contains an equal sign (=), it splits it into a key-value pair; otherwise, it assumes the value is 1 and assigns it to the key. Finally, it returns a dictionary containing the extracted key-value pairs. This function is used for parsing and converting command-line arguments into a dictionary format for further processing or configuration.

```python
def readCommand(argv):
    """
    Processes the command used to run pacman from the command line.
    """
    from optparse import OptionParser
    usageStr = """
    USAGE:      python pacman.py <options>
    EXAMPLES:   (1) python pacman.py
                    - starts an interactive game
                (2) python pacman.py --layout smallClassic --zoom 2
                OR  python pacman.py -l smallClassic -z 2
                    - starts an interactive game on a smaller board, zoomed in
    """
    parser = OptionParser(usageStr)

    parser.add_option('-n', '--numGames', dest='numGames', type='int',
                      help=default('the number of GAMES to play'), metavar='GAMES', default=1)
    parser.add_option('-l', '--layout', dest='layout',
                      help=default(
                          'the LAYOUT_FILE from which to load the map layout'),
                      metavar='LAYOUT_FILE', default='mediumClassic')
    parser.add_option('-p', '--pacman', dest='pacman',
                      help=default(
                          'the agent TYPE in the pacmanAgents module to use'),
                      metavar='TYPE', default='KeyboardAgent')
    parser.add_option('-t', '--textGraphics', action='store_true', dest='textGraphics',
                      help='Display output as text only', default=False)
    parser.add_option('-q', '--quietTextGraphics', action='store_true', dest='quietGraphics',
                      help='Generate minimal output and no graphics', default=False)
    parser.add_option('-g', '--ghosts', dest='ghost',
                      help=default(
                          'the ghost agent TYPE in the ghostAgents module to use'),
                      metavar='TYPE', default='RandomGhost')
    parser.add_option('-k', '--numghosts', type='int', dest='numGhosts',
                      help=default('The maximum number of ghosts to use'), default=4)
    parser.add_option('-z', '--zoom', type='float', dest='zoom',
                      help=default('Zoom the size of the graphics window'), default=1.0)
    parser.add_option('-f', '--fixRandomSeed', action='store_true', dest='fixRandomSeed',
                      help='Fixes the random seed to always play the same game', default=False)
    parser.add_option('-r', '--recordActions', action='store_true', dest='record',
                      help='Writes game histories to a file (named by the time they were played)', default=False)
    parser.add_option('--replay', dest='gameToReplay',
                      help='A recorded game file (pickle) to replay', default=None)
    parser.add_option('-a', '--agentArgs', dest='agentArgs',
                      help='Comma seperated values sent to agent. e.g. "opt1=val1,opt2,opt3=val3"')
    parser.add_option('-x', '--numTraining', dest='numTraining', type='int',
                      help=default('How many episodes are training (suppresses output)'), default=0)
```

- This function is made so it would read commands inputed through the CMD

```python
def loadAgent(pacman, nographics):
    # Looks through all pythonPath Directories for the right module,
    pythonPathStr = os.path.expandvars("$PYTHONPATH")
    if pythonPathStr.find(';') == -1:
        pythonPathDirs = pythonPathStr.split(':')
    else:
        pythonPathDirs = pythonPathStr.split(';')
    pythonPathDirs.append('.')

    for moduleDir in pythonPathDirs:
        if not os.path.isdir(moduleDir):
            continue
        moduleNames = [f for f in os.listdir(
            moduleDir) if f.endswith('gents.py')]
        for modulename in moduleNames:
            try:
                module = __import__(modulename[:-3])
            except ImportError:
                continue
            if pacman in dir(module):
                if nographics and modulename == 'keyboardAgents.py':
                    raise Exception(
                        'Using the keyboard requires graphics (not text display)')
                return getattr(module, pacman)
    raise Exception('The agent ' + pacman +
                    ' is not specified in any *Agents.py.')


def replayGame(layout, actions, display):
    import pacmanAgents
    import ghostAgents
    rules = ClassicGameRules()
    agents = [pacmanAgents.GreedyAgent()] + [ghostAgents.RandomGhost(i+1)
                                             for i in range(layout.getNumGhosts())]
    game = rules.newGame(layout, agents[0], agents[1:], display)
    state = game.state
    display.initialize(state.data)

    for action in actions:
            # Execute the action
        state = state.generateSuccessor(*action)
        # Change the display
        display.update(state.data)
        # Allow for game specific conditions (winning, losing, etc.)
        rules.process(state, game)

    display.finish()
```

- loadAgent function loads an agent class specified by the pacman argument. It searches through directories in the Python path, including the current directory, to find modules ending with "Agents.py". For each module found, it attempts to import it. If successful, it checks if the specified pacman agent is defined within the module. If found, it returns the agent class. If nographics is enabled and the agent is a keyboard agent, it raises an exception because keyboard agents require graphics. If no matching agent is found, it raises an exception indicating that the specified agent is not available.
- replayGame function replays a game given the layout, a sequence of actions, and a display. It imports modules for Pacman and ghost agents, then initializes the game using ClassicGameRules. It sets up agents for Pacman (a GreedyAgent) and the ghosts (RandomGhost agents).

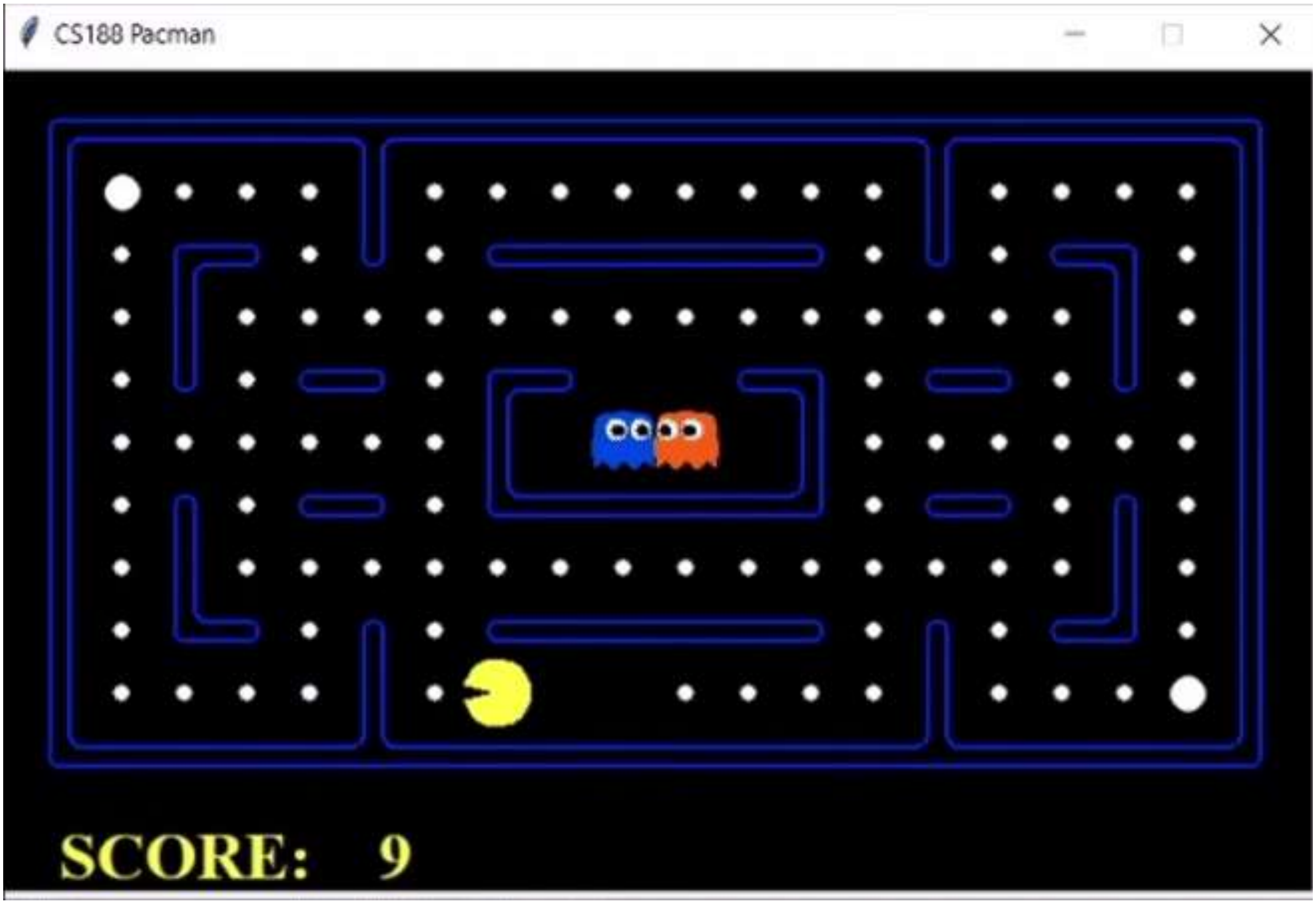
```
def runGames(layout, pacman, ghosts, display, numGames, record, numTraining=0, catchExceptions=False, timeout=30):
    import __main__
    __main__.__dict__['_display'] = display

    rules = ClassicGameRules(timeout)
    games = []

    for i in range(numGames):
        beQuiet = i < numTraining
        if beQuiet:
                # Suppress output and graphics
            import textDisplay
            gameDisplay = textDisplay.NullGraphics()
            rules.quiet = True
        else:
            gameDisplay = display
            rules.quiet = False
        game = rules.newGame(layout, pacman, ghosts,
                             gameDisplay, beQuiet, catchExceptions)
        game.run()
        if not beQuiet:
            games.append(game)

        if record:
            import time
            import pickle
            fname = ('recorded-game-%d' % (i + 1)) + \
                '-'.join([str(t) for t in time.localtime()[1:6]])
            f = file(fname, 'w')
            components = {'layout': layout, 'actions': game.moveHistory}
            pickle.dump(components, f)
            f.close()

    if (numGames-numTraining) >= 0:
        scores = [game.state.getScore() for game in games]
        wins = [game.state.isWin() for game in games]
        winRate = wins.count(True) / float(len(wins))
        print('Average Score:', sum(scores) / float(len(scores)))
        print('Scores:       ', ', '.join([str(score) for score in scores]))
        print('Win Rate:      %d/%d (%.2f)' %
              (wins.count(True), len(wins), winRate))
        print('Record:       ', ', '.join(
            [['Loss', 'Win'][int(w)] for w in wins]))

    return games
```

- runGames function handles the running of multiple games in a Pacman environment. It sets up the game display, initializes game rules, and runs each game sequentially. During the games, it records moves if specified, calculates statistics such as scores and win rates, and prints the results. Finally, it returns a list of game instances.

```
if __name__ == '__main__':
    """
    The main function called when pacman.py is run
    from the command line:

    > python pacman.py

    See the usage string for more details.

    > python pacman.py --help
    """
    args = readCommand(sys.argv[1:])  # Get game components based on input
    runGames(**args)
```

- Finally in the main part of the code we take the input from the cmd and run the game

14) GUI & Results:-



Pacman Agent	Win Rate (for 100 games)
Reflex Agent	90%
Left Turn Agent	0%
Greedy Agent	5%