**Python Project**

**Date**: 16/08/2016

**Roll No. and Name**: Pooja Malvi (13bce051) and Hiral Keshwala (13bce043)

**Course Code and Name**: Open Source Development Lab

**Practical No. 1**

AIM: To create Pacman game using Pygame in Python.

**Methodology followed:**

1. Installation of Pygame module in Pycharm
2. Import necessary files and docs for the game development.
3. Creating layout using Pygame.org libraries.
4. Applying necessary algorithm for movement of the player.
5. Implementing logic and development of game.

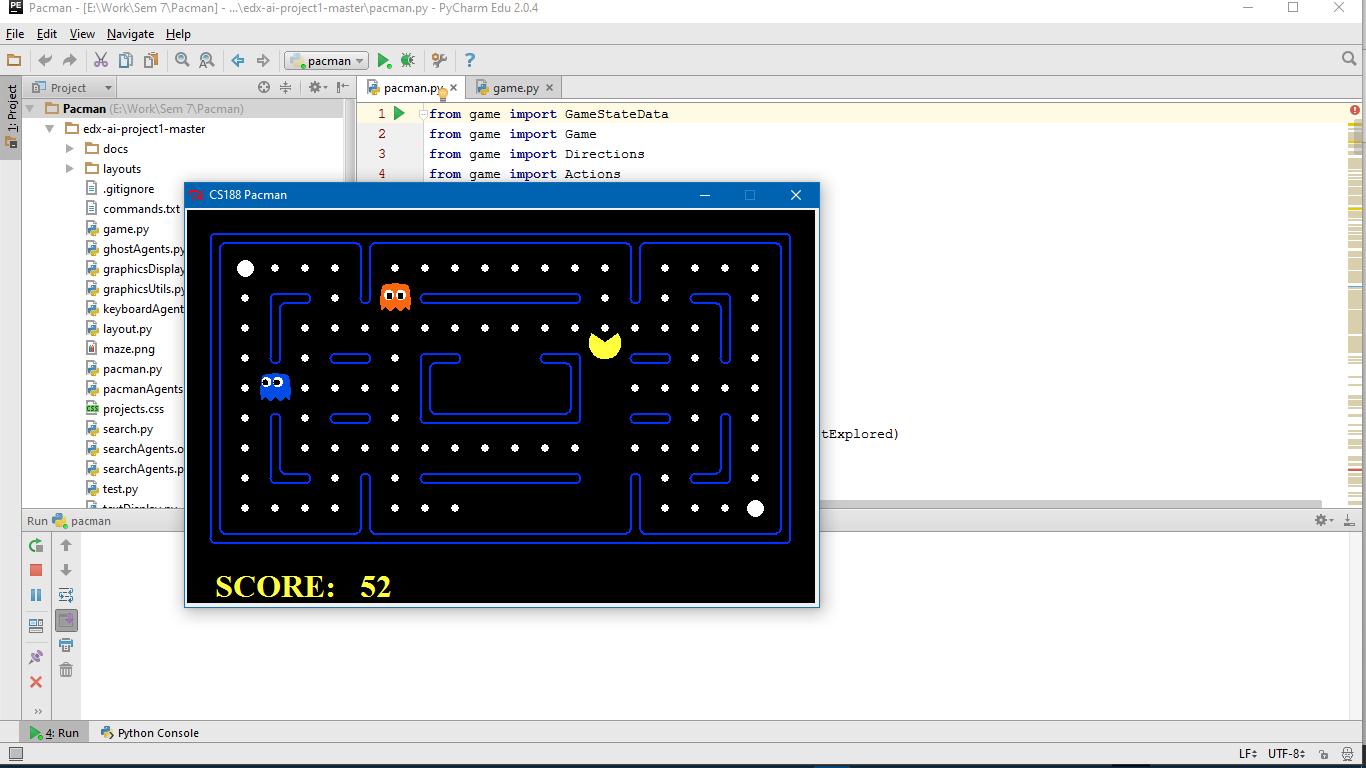
**Code:**

**from** game **import** GameStateData  
**from** game **import** Game  
**from** game **import** Directions  
**from** game **import** Actions  
**from** util **import** nearestPoint  
**from** util **import** manhattanDistance  
**import** util, layout  
**import** sys, types, time, random, os  
  
  
**class** GameState:  
 explored = set()  
 **def** getAndResetExplored():  
 tmp = GameState.explored.copy()  
 GameState.explored = set()  
 **return** tmp  
 getAndResetExplored = staticmethod(getAndResetExplored)  
  
 **def** getLegalActions( self, agentIndex=0 ):  
 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 )  
  
 **def** generateSuccessor( self, agentIndex, action):  
 **if** self.isWin() **or** self.isLose(): **raise** Exception(**'Can\'t generate a successor of a terminal state.'**)  
  
 state = GameState(self)  
  
 **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  
 **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  
 **return** state  
  
 **def** getLegalPacmanActions( self ):  
 **return** self.getLegalActions( 0 )  
  
 **def** generatePacmanSuccessor( self, action ):  
 **return** self.generateSuccessor( 0, action )  
  
 **def** getPacmanState( self ):  
 **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** self.data.score  
  
 **def** getCapsules(self):  
 **return** self.data.capsules  
  
 **def** getNumFood( self ):  
 **return** self.data.food.count()  
  
 **def** getFood(self):  
 **return** self.data.food  
  
 **def** getWalls(self):  
 **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  
  
 **def** \_\_init\_\_( self, prevState = None ):  
 **if** prevState != None: *# Initial state* self.data = GameStateData(prevState.data)  
 **else**:  
 self.data = GameStateData()  
  
 **def** deepCopy( self ):  
 state = GameState( self )  
 state.data = self.data.deepCopy()  
 **return** state  
  
 **def** \_\_eq\_\_( self, other ):  
 **return** self.data == other.data  
  
 **def** \_\_hash\_\_( self ):  
 **return** hash( self.data )  
  
 **def** \_\_str\_\_( self ):  
  
 **return** str(self.data)  
  
 **def** initialize( self, layout, numGhostAgents=1000 ):  
 self.data.initialize(layout, numGhostAgents)  
  
SCARED\_TIME = 40  
COLLISION\_TOLERANCE = 0.7  
TIME\_PENALTY = 1  
  
**class** ClassicGameRules:  
 **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):  
 **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()  
  
 **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  
  
**class** PacmanRules:  
 PACMAN\_SPEED=1  
  
 **def** getLegalActions( state ):  
 **return** Actions.getPossibleActions( state.getPacmanState().configuration, state.data.layout.walls )  
 getLegalActions = staticmethod( getLegalActions )  
  
 **def** applyAction( state, action ):  
 legal = PacmanRules.getLegalActions( state )  
 **if** action **not in** legal:  
 **raise** Exception(**"Illegal action "** + str(action))  
  
 pacmanState = state.data.agentStates[0]  
  
 vector = Actions.directionToVector( action, PacmanRules.PACMAN\_SPEED )  
 pacmanState.configuration = pacmanState.configuration.generateSuccessor( vector )  
  
 next = pacmanState.configuration.getPosition()  
 nearest = nearestPoint( next )  
 **if** manhattanDistance( nearest, next ) <= 0.5 :  
 PacmanRules.consume( nearest, state )  
 applyAction = staticmethod( applyAction )  
  
 **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  
 *#* ***TODO: cache numFood?*** 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  
 **for** index **in** range( 1, len( state.data.agentStates ) ):  
 state.data.agentStates[index].scaredTimer = SCARED\_TIME  
 consume = staticmethod( consume )  
  
**class** GhostRules:  
 GHOST\_SPEED=1.0  
 **def** getLegalActions( state, ghostIndex ):  
 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 )  
  
 **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  
 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 )  
  
**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  
  
**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, cPickle  
 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}  
 cPickle.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  
  
**if** \_\_name\_\_ == **'\_\_main\_\_'**:  
 **"""  
  
 """** args = readCommand( sys.argv[1:] )  
 runGames( \*\*args )  
 **pass**

**Sample Input**

No input required.

**Sample Output**



**Conclusions**

The Pacman game developed using pygame module in python is a One level game, following basic rules of pacman. The development environment chosen is Pycharm IDE.

**Signature of Teacher**