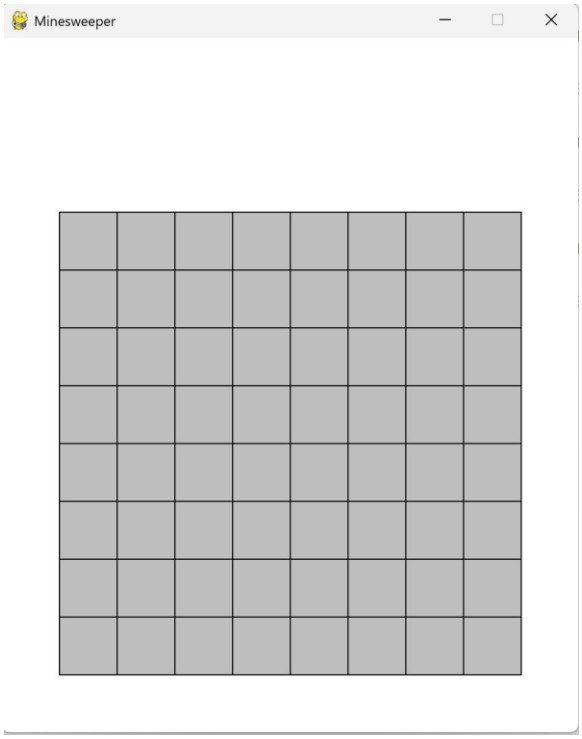
pyGame Lecture – Game Example

Minesweeper

I. Game Summary

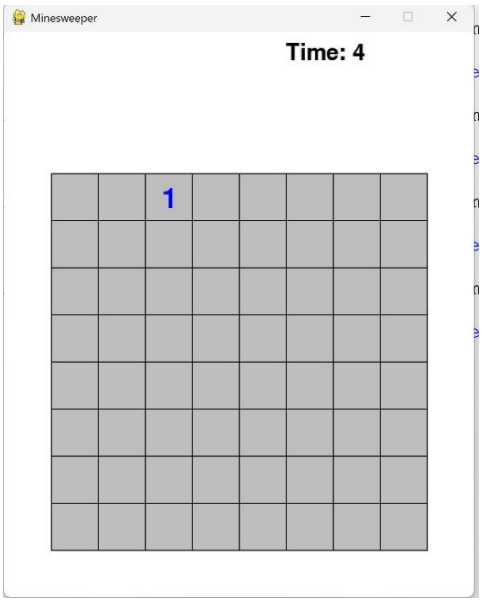
Minesweeper was an included game with the old Windows operating system, starting in the early 1990s. The game is played on a rectangular grid of squares, whose contents are initially hidden from the user. Some of the squares contain bombs while other squares are safe. Here is a screen capture of our version of the game when it starts:



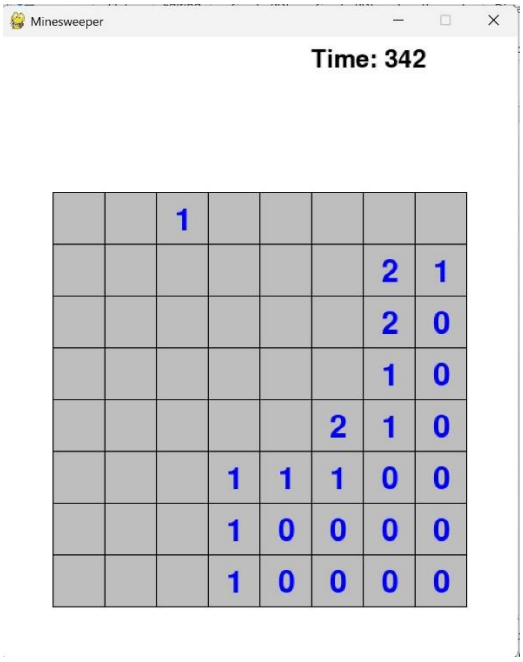
To play the game, the user clicks on a square. If the square is a bomb, the user loses. Otherwise, when the square is uncovered, a number is shown. That number represents the number of bombs adjacent to the square. The player can use this information to determine which squares to click and which ones to avoid. The player wins if they click (uncover) all of the safe squares without clicking on a bomb.

In the real game, a player has the option to mark the squares he/she thinks are bombs to help them visually without clicking on them. (In the real game, a flag is placed on top of the square when such a marking is made. This is made with a right mouse click while the regular uncovering of squares is done with a left mouse click.) In our version of the game, we can't mark squares we think are bombs.

On the next page is an image after the player has clicked on a single non-bomb square:



You'll notice that we also display the number of seconds after the game started as the user plays. Finally, if the player clicks on a square with 0 adjacent bombs, it's obvious that all the surrounding squares can be clicked. In the original version of the game, the game automatically cleared these squares for you as well. Of course, some of those might reveal 0 adjacent bombs and that would trigger the process again. This is called the "recursive clear" and is implemented in our version of the game. The screen capture below shows the result of the second click, which was made on the square in row 7, column 7 (using 1-based counting):



The 0s "flooded" to adjacent squares and continued until they hit a "border" of non-zero values where the recursive clears stopped. (If you start with the 1 on the bottom row, and make a path going up, then right, then up, then right, then up and one last right turn, you'll see that border.)

II. Organization of This Implementation

Our implementation has 3 files:

1. minemain
2. minesweeper
3. minefunc

The first file handles the main game screen, which stays fairly consistent, just showing changes from the various clicks.

The second stores a single class, the mineclass, which handles all of the game logic. A mineclass object stores the status of the board (its size, the contents of each square, which squares are visible, etc.) Finally the minefunc class just has a few functions that allow us to visually display the graphics that are stored in the mineclass object. In the object, the data is just stored as a list of lists either storing the '*' character for a bomb of a numeric character for the number of adjacent bombs. The functions in minefunc take that list of lists data and draw the appropriate graphics on the screen.

III. minemain.py

In the real game, when the user first clicks, she is guaranteed not to click on a bomb. The way this is handled is that the random placements of the bomb are determined AFTER the user chooses the first square. In addition, the game timer doesn't start until the user makes this first click.

In our main file, we handle this portion of the logic, initially setting our mineclass object to None (special reference that means nothing), and only creating our object once the user has pressed on a valid square. We also keep track of the status of the game. Here are out initial settings for variables in this file:

```
myMine = None
status = minefunc.NOT_STARTED
endT = -1
curT = -1
startT = -1
```

When the user clicks on a square, here is how we handle the left mouse click:

```
            if event.type == MOUSEBUTTONDOWN:
                if pygame.mouse.get_pressed()[0]:
                    sq = minefunc.getXY(event.pos)

                    if myMine == None and sq != None:
                        myMine = mineclass(sq[0], sq[1])
                        myMine.move(sq[0], sq[1])
                        status = minefunc.CONT
                        startT = time.time()

                    # Valid move in the game.
                    elif myMine != None and sq != None:
                        status = myMine.move(sq[0], sq[1])
```

As soon as we get a button click, we take the position and pass it to a function, getXY, in the file minefunc. This returns either None, if the user clicks outside one of the squares or it returns the position (0-indexed row and column on the board) of the click.

The first if statement detects if the game hasn't yet started and the click is a valid one. In this case, the mineclass object is created (the constructor takes in where the click is so that it can ensure that it doesn't create a bomb at that location). We mark the status of the game as continuing, execute the move and start the timer. If the game is already going, then the only action to execute is executing the move, which is the elif clause.

Finally, as the game loop runs, we will always draw the empty screen, followed by drawing all visible items on top of the empty screen. Then we do all of the game processing. Here is that code:

```
# Now draw the board.
minefunc.drawEmptyBoard(screen)
if myMine != None:
    myMine.drawVisible(screen, minefunc.TOP_X, minefunc.TOP_Y, minefunc.STEP)

# Calculate to display current game time.
if startT > 0 and status == minefunc.CONT:
    curT = int(time.time() - startT)

# Draw only if game is running.
if startT > 0:
    minefunc.draw("Time: "+str(curT), font, 'black', screen, SCREEN_W-200, 10)

# Player lost.
if status == minefunc.LOST:
    minefunc.draw("You lose!", font, 'red', screen, 10, 10)
    if endT == -1:
        endT = time.time()

# Here's for winning.
elif status == minefunc.WIN:
    minefunc.draw("You win in "+str(curT)+" seconds!",font,'red',screen,10, 10)
    if endT == -1:
        endT = time.time()

# So it doesn't disappear right away.
if endT > 0 and time.time()-endT > 5:
    pygame.quit()
    sys.exit()

# Update and wait.
pygame.display.update()
clock.tick(30)
```

The first segment draws the empty board and the numbers on top of it. The following segment updates the current time. Only if the game is running is the time drawn onto the screen. Then the last three segments are for handling winning, losing and a five second delay after the game is over so the user can see the result. For both winning and losing, the final time of the game is noted and presented on the screen.

IV. minefunc.py

This file just stores the many constants used in the game, has the draw function that draws text on the screen and has the mechanics to draw an empty board and determine the row column location of a click on the board. First, let's look at all of the constants in the file:

```
# Where the board will start.
TOP_X = 50
TOP_Y = 150
STEP = 50
OFF_X = 18
OFF_Y = 13

# Size of board.
ROWS = 8
COLS = 8
NUMB = 10

# Codes for game states.
NOT_STARTED = -1
NO_MOVE = 0
LOST = 1
CONT = 2
WIN = 3

# Directions to adjacent squares.
DX = [-1,-1,-1,0,0,1,1,1]
DY = [-1,0,1,-1,1,-1,0,1]
```

The first set define the x and y coordinates of the top of the actual board. The step indicates how many pixels each square will be and the offsets are based on the margins from the actual (0, 0) on the display screen.

The size of the board and number of bombs is the classical setting from the original game.

This is followed by the codes for the various statuses allowed in the game. We start in the mode NOT_STARTED, and then move to the CONT mode, where we usually stay. NO_MOVE is if the user clicks somewhere, but it's not a new valid square to uncover.

Finally, the DX, DY arrays are critical. They store the relative direction of movement from a square to all its adjacent squares. We can think of index i as storing the movement in X and Y for direction i. For example, direction 0 is up and left, direction 1 is just up, etc.

Here is the drawEmptyBoard function:

```
# Draws an empty board.
def drawEmptyBoard(surface):

    # Total size of grid.
    myw = STEP*COLS
    myh = STEP*ROWS

    # Gray in the background.
    pygame.draw.rect(surface, 'gray', (TOP_X, TOP_Y, myw, myh))

    # Horizontal lines.
    for i in range(ROWS+1):
        pygame.draw.line(surface, 'black', (TOP_X, TOP_Y+STEP*i), (TOP_X+myw,
TOP_Y+STEP*i))

    # Vertical lines.
    for i in range(COLS+1):
        pygame.draw.line(surface,    'black',    (TOP_X+STEP*i    ,TOP_Y)    ,
(TOP_X+STEP*i, TOP_Y+myh))
```

Notice that we just draw one big gray rectangle for the whole board, but then we draw black lines to give the appearance of separate squares. Since the board is 8 by 8, we draw 9 horizontal and vertical lines.

Finally, the getXY just uses some subtraction and integer division to recover the row and column of a mouse click. It returns None if the click isn't on a valid square:

```
def getXY(pos):

    # Calculate mathematically.
    myC = (pos[0] - TOP_X)//STEP
    myR = (pos[1] - TOP_Y)//STEP

    # Out of bounds test.
    if myR < 0 or myR >= ROWS or myC < 0 or myC >= COLS:
        return None

    # This is good.
    return (myR, myC)
```

Subtracting by TOP_X and TOP_Y offsets the math to 0-based top left coordinates. Then we just use integer division by the pixel size of each square to get the 0 based row and column. Finally we screen out invalid row, column values before returning the tuple.

V. minesweeper.py

This is where the complicated game logic occurs. Our mineclass object has three instance variables:

1. board (8 by 8 list of lists storing the characters hidden underneath the board)
2. show (8 by 8 list of lists storing True or False indicating if the square should be shown or not)
3. need (an integer representing the number of new squares that need to be cleared for the win)

Let's take a look at the constructor, which initializes all of these items. The board starts as empty, but then we'll fill it with randomly placed bombs in different squares than the user chose and then fill the rest of the squares with numbers indicating the number of adjacent bombs. show will get set to all False, and need will get set to the number of total squares minus the number of bombs.

Here is the first part of the constructor which puts the bombs in randomly generated places but not (x, y), where the user first clicked:

```
    def __init__(self, x, y):

        self.board = []
        for i in range(minefunc.ROWS):
            tmp = []
            for j in range(minefunc.COLS):
                tmp.append(' ')
            self.board.append(tmp)

        i = 0
        while i < minefunc.NUMB:

            # Generate a random location.
            loc = random.randint(0, minefunc.ROWS*minefunc.COLS-1)
            myx = loc//minefunc.COLS
            myy = loc%minefunc.COLS

            # Can't place bomb on the first clicked square.
            if myx == x and myy == y:
                continue

            # Forces unique squars.
            if self.board[myx][myy] == '*':
                continue

            # Very helpful for testing.
            if minefunc.DEBUG:
                print(myx, myy, "making bomb")

            # Place a bomb here.
            self.board[myx][myy] = '*'
            i += 1
```

The first segment of code is fairly self-explanatory. We use a while loop to control the number of bombs we place, since some placements may fail. We generate a random location on the grid (we use a single number and integer division and mod). The two cases we don't place the bomb are if the user clicked on that square first or if the square already has a bomb. Otherwise, we place the '*' character

in the randomly selected square and add 1 to the bombs placed (i). Also, there's a debug flag which just prints out to the regular screen where the bombs are =)

Now, let's look at the rest of the constructor:

```
        # Here we generate the hidden board with all numbers and bombs.
        for i in range(minefunc.ROWS):
            for j in range(minefunc.COLS):

                # These squares are done.
                if self.board[i][j] == '*':
                    continue

                # Put the number here, computed by a function
                self.board[i][j] = chr(ord('0') + self.numAdj(i, j))

        # This will keep track of what we show to the player.
        self.show = []
        for i in range(minefunc.ROWS):
            tmp = []
            for j in range(minefunc.COLS):
                tmp.append(False)
            self.show.append(tmp)

        # Number of squares we must clear.
        self.need = minefunc.ROWS*minefunc.COLS - minefunc.NUMB
```

The double for loop finds all non-bomb squares and fills them with the appropriate character representing the number of adjacent bombs. The numAdj method does this calculation.

Finally we initialize show to be all False and the number of squares we need to clear to the total minus the number of bombs.

Next, let's look at how we calculate the number of adjacent bombs. This is where the DX/DY array comes in handy. When we are at a position (x, y), we want to look at the following positions:

(x-1, y-1), (x-1, y), (x-1, y+1), (x, y-1), (x, y+1), (x+1, y-1), (x+1, y), and (x+1, y+1).

Notice that these are all of the form:

x + DX[i]
y + DY[i]

as i ranges from 0 to 7, inclusive. The only issue is that we want to avoid a list out of bounds error. So, before we go to the square, we just check to see if the adjacent square is in bounds. If it's not, we skip it in our count.

The code for the method is on the next page:

```
# Returns the number of bombs adjacent to (x, y) on board myb.
    def numAdj(self, x, y):

        res = 0

        # Go in all directions.
        for i in range(len(minefunc.DX)):

            # Adjacent square i.
            nX = x + minefunc.DX[i]
            nY = y + minefunc.DY[i]

            # Out of bounds.
            if nX < 0 or nX >= minefunc.ROWS or nY < 0 or nY >= minefunc.COLS:
                continue

            # It's a bomb, count it.
            if self.board[nX][nY] == '*':
                res += 1

        # Total number of adjacent bombs.
        return res
```

Here we just store the coordinates of each adjacent square in (nX, nY), which is just nextX, nextY, for short. We use the if statement to screen out the out of bounds case, and then we add 1 to our result of the corresponding adjacent square is a bomb.

Finally, we come to the critical move method, which executes a move. This method is recursive, which means that sometimes it calls itself, whenever the user presses a square that has 0 adjacent bombs. We'll take the strategy of having lots of base cases to stop the method early in many cases and only recursing at the end.

Our base cases are:

1. You click on an illegal location.
2. You click on a square already shown.
3. You click on a bomb.

Here are those cases handled at the beginning of the method:

```
    def move(self, x, y):

        if x < 0 or x >= minefunc.ROWS or y < 0 or y >= minefunc.COLS:
            return minefunc.NO_MOVE

        if self.show[x][y]:
            return minefunc.NO_MOVE

        if self.board[x][y] == '*':
            self.showAll()
            return minefunc.LOST
```

In each case, we just return the appropriate game code.

If these cases aren't relevant, then the move is valid and we execute it:

```
self.show[x][y] = True
self.need -= 1
```

Now, here is where the recursive clear comes in. If the value that is uncovered is 0 then we just call this move method on all adjacent squares. Again, we make use the the DX/DY arrays to write this code succinctly:

```
# Recursive clear check.
if self.board[x][y] == '0':

    # Here it is!
    for i in range(len(minefunc.DX)):
        self.move(x + minefunc.DX[i], y + minefunc.DY[i])
```

It's quite impressive how short this is! We just check if the character's 0, and then we just recursively call clear in all directions. Notice that we can call the function on out of bounds squares, because this is one of the base cases we catch!

Finally, after this is done, we must just return the status of the game, which is based on the number of squares we need to clear:

```
# Here is how we determine our current game state.
if self.need > 0:
    return minefunc.CONT
else:
    return minefunc.WIN
```