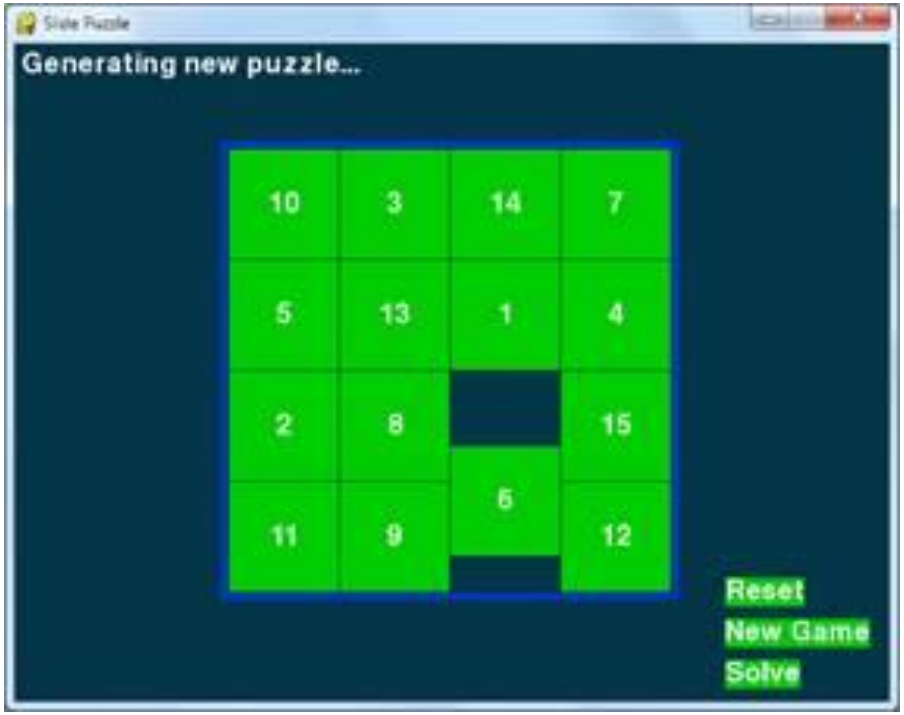
Slide Puzzle

How to Play Slide Puzzle ?



- The board is a 4x4 grid with
  - fifteen tiles - numbered 1 through 15 going left to right and
  - one blank space.
- The tiles start out in random positions, and the player must slide tiles around until the tiles are back in their original order.

Source Code to Slide Puzzle

- This source code can be downloaded from

  http://invpy.com/slidepuzzle.py
- Task 1 – download the code
- Task 2 – activate the game

Second Verse, Same as the First

```
6. import pygame, sys, random
7. from pygame.locals import *
9. # Create the constants (go ahead and experiment with different values)
10. BOARDWIDTH = 4  # number of columns in the board
11. BOARDHEIGHT = 4 # number of rows in the board
12. TILESIZE = 80
13. WINDOWWIDTH = 640
14. WINDOWHEIGHT = 480
15. FPS = 30
16. BLANK = None
18. #     R    G    B
19. BLACK =         (  0,   0,   0)
20. WHITE =         (255, 255, 255)
21. BRIGHTBLUE =    (  0,  50, 255)
22. DARKTURQUOISE = (  3,  54,  73)
23. GREEN =         (  0, 204,   0)
25. BGCOLOR = DARKTURQUOISE
```

- Much of the code is similar to the previous game (Memory Puzzle), especially the constants being set at the start of the code.
- This code at the top of the program just handles all basic importing of modules and creating constants.

. . .

```
26. TILECOLOR = GREEN
27. TEXTCOLOR = WHITE
28. BORDERCOLOR = BRIGHTBLUE
29. BASICFONTSIZE = 20
31. BUTTONCOLOR = WHITE
32. BUTTONTEXTCOLOR = BLACK
33. MESSAGECOLOR = WHITE
35. XMARGIN = int((WINDOWWIDTH - (TILESIZE *BOARDWIDTH +
    (BOARDWIDTH - 1))) / 2)
36. YMARGIN = int((WINDOWHEIGHT - (TILESIZE * BOARDHEIGHT +
    (BOARDHEIGHT - 1))) / 2)
38. UP = 'up'
39. DOWN = 'down'
40. LEFT = 'left'
41. RIGHT = 'right'
```

Setting Up the Buttons

```
43. def main():
44. global FPSCLOCK, DISPLAYSURF, BASICFONT, RESET_SURF, RESET_RECT,
NEW_SURF, NEW_RECT, SOLVE_SURF, SOLVE_RECT
46. pygame.init()
47. FPSCLOCK = pygame.time.Clock()
48. DISPLAYSURF = pygame.display.set_mode((WINDOWWIDTH, WINDOWHEIGHT))
49. pygame.display.set_caption('Slide Puzzle’)
50. BASICFONT = pygame.font.Font('freesansbold.ttf', BASICFONTSIZE)
52. # Store the option buttons and their rectangles in OPTIONS.
53.  RESET_SURF, RESET_RECT = makeText('Reset',    TEXTCOLOR, TILECOLOR,
WINDOWWIDTH - 120, WINDOWHEIGHT - 90)
54.  NEW_SURF,   NEW_RECT   = makeText('New Game', TEXTCOLOR, TILECOLOR,
WINDOWWIDTH - 120, WINDOWHEIGHT - 60)
55.  SOLVE_SURF, SOLVE_RECT = makeText('Solve',    TEXTCOLOR, TILECOLOR,
WINDOWWIDTH - 120, WINDOWHEIGHT – 30)
57.  mainBoard, solutionSeq = generateNewPuzzle(80)
58.   SOLVEDBOARD = getStartingBoard() # a solved board is the same as the board in a start
state.
```

. . .

- The first part of the main() function will handle creating the window, Clock object, and Font object.
- The makeText() returns a pygame.Surface object and pygame.Rect object which can be used to make clickable buttons.
- The Slide Puzzle game will have three buttons:
  - a “Reset” button that will undo any moves the player has made,
  - a “New” button that will create a new slide puzzle, and
  - a “Solve” button that will solve the puzzle for the player.
- We have two board data structures for this program.
  - One board will represent the current game state.
  - The other board will have its tiles in the “solved” state
- The generateNewPuzzle() creates a board data structure
  - It starts off in the ordered, solved state and then performs 80 random slide moves
- **IMPORTANT!** The generateNewBoard() also returns a list of all random moves that were performed on it (which will be stored in a variable named solutionSeq).

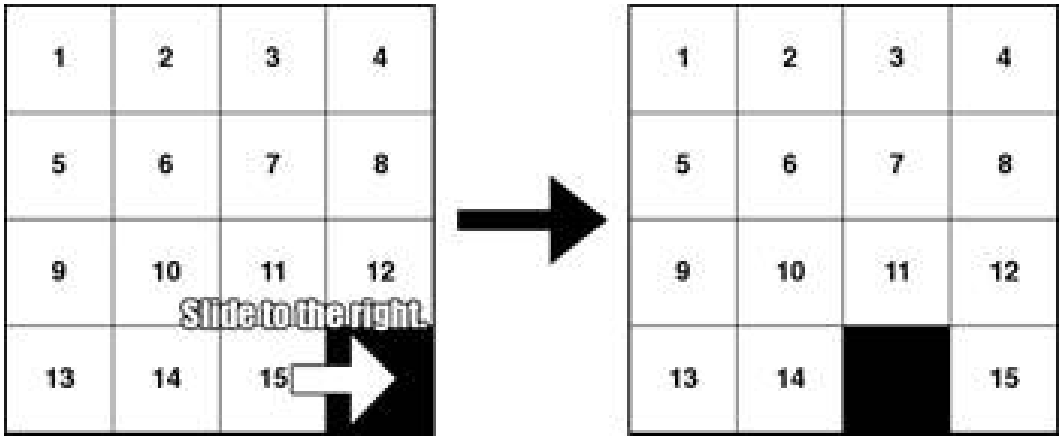
Being Smart By Using Stupid Code

```
59.     allMoves = [] # list of moves made from the solved configuration
```

- Solving a slide puzzle can be really tricky.
  - We would have to figure out an algorithm that can solve the slide puzzle.
  - That would be very difficult and involve a lot of cleverness and effort to put into this program.
- There's an easier way !!!! ☺
  - We could just have the computer memorize all random slides it made when it created the board data structure, and then the board can be solved just by performing the opposite slide.

..

- For example, below we perform a “right” slide on the board on the left side of the page, which leaves the board in the state that is on the right side of the page
- After the right slide, if we do the opposite slide (a left slide) then the board will be back in the original state.
- So to get back to the original state after several slides, we just have to do the opposite slides in reverse order.



The Main Game Loop

```
61.    while True: # main game loop
62.        slideTo = None # the direction, if any, a tile should slide
63.        msg = '' # contains the message to show in the upper left corner.
64.        if mainBoard == SOLVEDBOARD:
65.            msg = 'Solved!’
67.        drawBoard(mainBoard, msg)
```

Clicking on the Buttons

```
69.     checkForQuit()
70.     for event in pygame.event.get(): # event handling loop
71.         if event.type == MOUSEBUTTONUP:
72.             spotx, spoty = getSpotClicked(mainBoard, event.pos[0], event.pos[1])
74.             if (spotx, spoty) == (None, None):
75.                 # check if the user clicked on an option button
76.                 if RESET_RECT.collidepoint(event.pos):
77.                     resetAnimation(mainBoard, allMoves) # clicked on Reset button
78.                     allMoves = []
79.                 elif NEW_RECT.collidepoint(event.pos):
80.                     mainBoard, solutionSeq = generateNewPuzzle(80) # clicked on New Game button
81.                     allMoves = []
82.                 elif SOLVE_RECT.collidepoint(event.pos):
83.                     resetAnimation(mainBoard, solutionSeq + allMoves) # clicked on Solve button
84.                     allMoves = []
```

Reset button is clicked

New button is clicked

Solve button is clicked

Sliding Tiles with the Mouse

```
85.             else:
86.                 # check if the clicked tile was next to the blank spot
88.                 blankx, blanky = getBlankPosition(mainBoard)
89.                 if spotx == blankx + 1 and spoty == blanky:
90.                     slideTo = LEFT
91.                 elif spotx == blankx - 1 and spoty == blanky:
92.                     slideTo = RIGHT
93.                 elif spotx == blankx and spoty == blanky + 1:
94.                     slideTo = UP
95.                 elif spotx == blankx and spoty == blanky - 1:
96.                     slideTo = DOWN
```

Sliding Tiles with the Keyboard

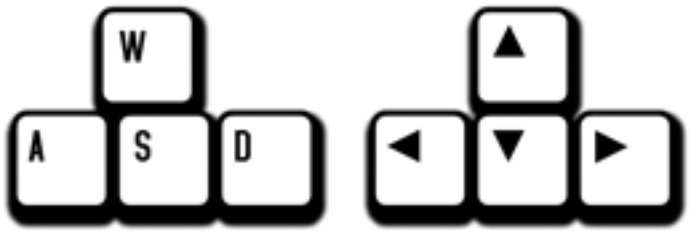
```
98.             elif event.type == KEYUP:
99.                 # check if the user pressed a key to slide a tile
100.                if event.key in (K_LEFT, K_a) and isValidMove(mainBoard, LEFT):
101.                    slideTo = LEFT
102.                elif event.key in (K_RIGHT, K_d) and isValidMove(mainBoard, RIGHT):
103.                    slideTo = RIGHT
104.                elif event.key in (K_UP, K_w) and isValidMove(mainBoard, UP):
105.                    slideTo = UP
106.                elif event.key in (K_DOWN, K_s) and isValidMove(mainBoard, DOWN):
107.                    slideTo = DOWN
```

“Equal To One Of” Trick with the in Operator

- The expression event.key in (K_LEFT, K_a) is just a Python trick to make the code simpler.
- It is a way of saying
  - “evaluate to True if event.key is equal to one of K_LEFT or K_a”.
- The following two expressions will evaluate the exact same way:
  - event.key in (K_LEFT, K_a)
  - event.key == K_LEFT or event.key == K_a
- You can really save on some space by using this trick when you have to check if a value is equal to one of multiple values.

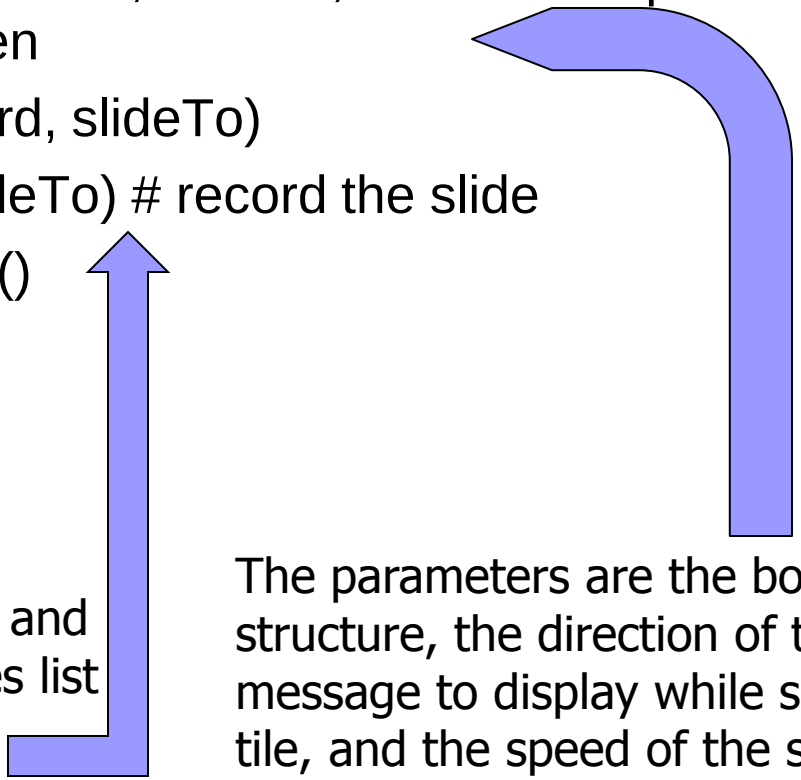
WASD and Arrow Keys

- The W, A, S, and D keys are commonly used in computer games to do the same thing as the arrow keys, except the player can use their left hand instead
  - W is for up,
  - A is for left,
  - S is for down, and
  - D is for right.



Actually Performing the Tile Slide

```
109.        if slideTo:
110.            slideAnimation(mainBoard, slideTo, 'Click tile or press arrow keys
to slide.', 8) # show slide on screen
111.            makeMove(mainBoard, slideTo)
112.            allMoves.append(slideTo) # record the slide
113.        pygame.display.update()
114.        FPSCLOCK.tick(FPS)
```



we update the actual board data structure (makeMove() function) and then add the slide to the allMoves list of all slides made so far.

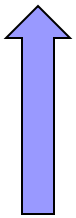
The parameters are the board data structure, the direction of the slide, a message to display while sliding the tile, and the speed of the sliding.

IDLE and Terminating Pygame Programs

```
117. def terminate():
118.     pygame.quit()
119.     sys.exit()
```

Checking for a Specific Event, and Posting Events to Event Queue

```
122. def checkForQuit():
123.     for event in pygame.event.get(QUIT): # get all the QUIT events
124.         terminate() # terminate if any QUIT events are present
125.     for event in pygame.event.get(KEYUP): # get all the KEYUP events
126.         if event.key == K_ESCAPE:
127.             terminate() # terminate if the KEYUP event was for the Esc key
128.         pygame.event.post(event) # put the other KEYUP event objects back
```



The rest of the events will stay in the event queue for the next time pygame.event.get() is called.

Pygame's event queue only stores up to 127 Event objects.
If your program does not call pygame.event.get() frequently enough and the queue fills up, then any new events that happen won't be added to the event queue.

Creating the Board Data Structure

```
131. def getStartingBoard():
132.     # Return a board data structure with tiles in the solved state.
133.     # For example, if BOARDWIDTH and BOARDHEIGHT are both 3, this function
134.     # returns [[1, 4, 7], [2, 5, 8], [3, 6, None]]
135.     counter = 1
136.     board = []
137.     for x in range(BOARDWIDTH):
138.         column = []
139.         for y in range(BOARDHEIGHT):
140.             column.append(counter)
141.             counter += BOARDWIDTH
142.         board.append(column)
143.         counter -= BOARDWIDTH * (BOARDHEIGHT - 1) + BOARDWIDTH - 1
144.
145.     board[BOARDWIDTH-1][BOARDHEIGHT-1] = None
146.     return board
```

Not Tracking the Blank Position

```
149. def getBlankPosition(board):
150.     # Return the x and y of board coordinates of the blank space.
151.     for x in range(BOARDWIDTH)):
152.         for y in range(BOARDHEIGHT):
153.             if board[x][y] == None:
154.                 return (x, y)
```

Making a Move by Updating the Board Data Structure

```
157. def makeMove(board, move):
158.     # This function does not check if the move is valid.
159.     blankx, blanky = getBlankPosition(board)
160.
161.     if move == UP:
162.         board[blankx][blanky], board[blankx][blanky + 1] = board[blankx][blanky + 1], board[blankx][blanky]
163.     elif move == DOWN:
164.         board[blankx][blanky], board[blankx][blanky - 1] = board[blankx][blanky - 1], board[blankx][blanky]
165.     elif move == LEFT:
166.         board[blankx][blanky], board[blankx + 1][blanky] = board[blankx + 1][blanky], board[blankx][blanky]
167.     elif move == RIGHT:
168.         board[blankx][blanky], board[blankx - 1][blanky] = board[blankx - 1][blanky], board[blankx][blanky]
```

When NOT to Use an Assertion

```
171. def isValidMove(board, move):
172.     blankx, blanky = getBlankPosition(board)
173.     return (move == UP and blanky != len(board[0]) - 1) or \
174.             (move == DOWN and blanky != 0) or \
175.             (move == LEFT and blankx != len(board) - 1) or \
176.             (move == RIGHT and blankx != 0)
```

Whether a move is valid or not depends on where the blank space is.

The \ slashes at the end of the first three lines tells the Python interpreter that that is not the end of the line of code (even though it is at the end of the line). This will let us split up a "line of code" across multiple lines to look pretty, rather than just have one very long unreadable line.

Getting a Not-So-Random Move

```
179. def getRandomMove(board, lastMove=None):
180.     # start with a full list of all four moves
181.     validMoves = [UP, DOWN, LEFT, RIGHT]
183.     # remove moves from the list as they are disqualified
184.     if lastMove == UP or not isValidMove(board, DOWN):
185.         validMoves.remove(DOWN)
186.     if lastMove == DOWN or not isValidMove(board, UP):
187.         validMoves.remove(UP)
188.     if lastMove == LEFT or not isValidMove(board, RIGHT):
189.         validMoves.remove(RIGHT)
190.     if lastMove == RIGHT or not isValidMove(board, LEFT):
191.         validMoves.remove(LEFT)
193.     # return a random move from the list of remaining moves
194.     return random.choice(validMoves)
```

Converting Tile Coordinates to Pixel Coordinates

```
197. def getLeftTopOfTile(tileX, tileY):
198.     left = XMARGIN + (tileX * TILESIZE) + (tileX - 1)
199.     top = YMARGIN + (tileY * TILESIZE) + (tileY - 1)
200.     return (left, top)
```

- The getLeftTopOfTile() function converts board coordinates to pixel coordinates.
- For the board XY coordinates that are passed in, the function calculates and returns the pixel XY coordinates of the pixel at the top left of that board space.

Converting from Pixel Coordinates to Board Coordinates

```
203. def getSpotClicked(board, x, y):
204.     # from the x & y pixel coordinates, get the x & y board coordinates
205.     for tileX in range(len(board)):
206.         for tileY in range(len(board[0])):
207.             left, top = getLeftTopOfTile(tileX, tileY)
208.             tileRect = pygame.Rect(left, top, TILESIZE, TILESIZE)
209.             if tileRect.collidepoint(x, y):
210.                 return (tileX, tileY)
211.     return (None, None)
```

Drawing a Tile

```
214. def drawTile(tilex, tiley, number, adjx=0, adjy=0):
215.     # draw a tile at board coordinates tilex and tiley, optionally a few
216.     # pixels over (determined by adjx and adjy)
217.     left, top = getLeftTopOfTile(tilex, tiley)
218.     pygame.draw.rect(DISPLAYSURF, TILECOLOR, (left + adjx, top + adjy,
TILESIZE, TILESIZE))
219.     textSurf = BASICFONT.render(str(number), True, TEXTCOLOR)
220.     textRect = textSurf.get_rect()
221.     textRect.center = left + int(TILESIZE / 2) + adjx, top + int(TILESIZE / 2) +
adjy
222.     DISPLAYSURF.blit(textSurf, textRect)
```

The drawTile() function will draw a single numbered tile on the board.
The tilex and tiley parameters are the board coordinates of the tile.
The number parameter is a string of the tile's number (like '3' or '12').
The adjx and adjy keyword parameters parameters are for making minor adjustments to the position of the tile.

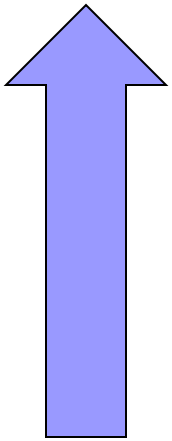
The Making Text Appear on the Screen

```
225. def makeText(text, color, bgcolor, top, left):
226.     # create the Surface and Rect objects for some text.
 227.     textSurf = BASICFONT.render(text, True, color, bgcolor)
228.     textRect = textSurf.get_rect()
229.     textRect.topleft = (top, left)
230.     return (textSurf, textRect)
```

- The makeText() function handles creating the Surface and Rect objects for positioning text on the screen.
- Instead of doing all these calls each time we want to make text on the screen, we can just call makeText() instead.

Drawing the Board

```
233. def drawBoard(board, message):
234.     DISPLAYSURF.fill(BGCOLOR)
235.     if message:
236.         textSurf, textRect = makeText(message, MESSAGECOLOR, BGCOLOR, 5, 5)
237.         DISPLAYSURF.blit(textSurf, textRect)
238.
239.     for tilex in range(len(board)):
 240.         for tiley in range(len(board[0])):
241.             if board[tilex][tiley]:
242.                 drawTile(tilex, tiley, board[tilex][tiley])
```



Remember that if statement conditions consider the blank string to be a False value, so if message is set to '' then the condition is False and lines 236 and 237 are skipped.

Drawing the Border of the Board

```
244.    left, top = getLeftTopOfTile(0, 0)
245.    width = BOARDWIDTH * TILESIZE
246.    height = BOARDHEIGHT * TILESIZE
247.    pygame.draw.rect(DISPLAYSURF, BORDERCOLOR,
(left - 5, top - 5, width + 11, height + 11), 4)
```

- The top left corner of the boarder will be 5 pixels to the left and 5 pixels above the top left corner of the tile at board coordinates (0, 0).
- The width and height of the border are calculated from the number of tiles wide and high the board is (stored in the BOARDWIDTH and BOARDHEIGHT constants) multiplied by the size of the tiles (stored in the TILESIZE constant).

Drawing the Buttons

- 249. DISPLAYSURF.blit(RESET_SURF, RESET_RECT)
- 250. DISPLAYSURF.blit(NEW_SURF, NEW_RECT)
- 251. DISPLAYSURF.blit(SOLVE_SURF, SOLVE_RECT)

- Finally, we draw the buttons off to the slide of the screen.
- The text and position of these buttons never changes, which is why they were stored in constant variables at the beginning of the main() function.

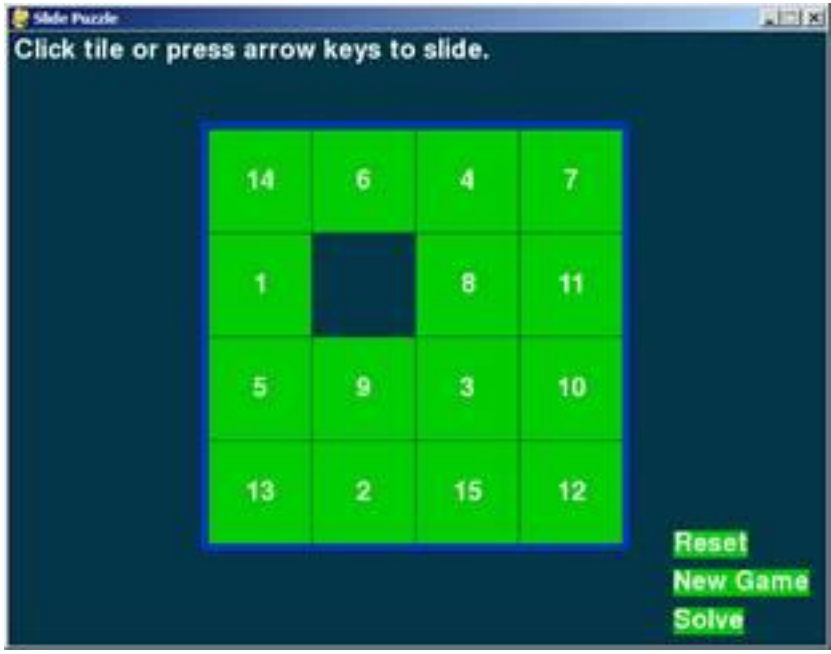
Animating the Tile Slides

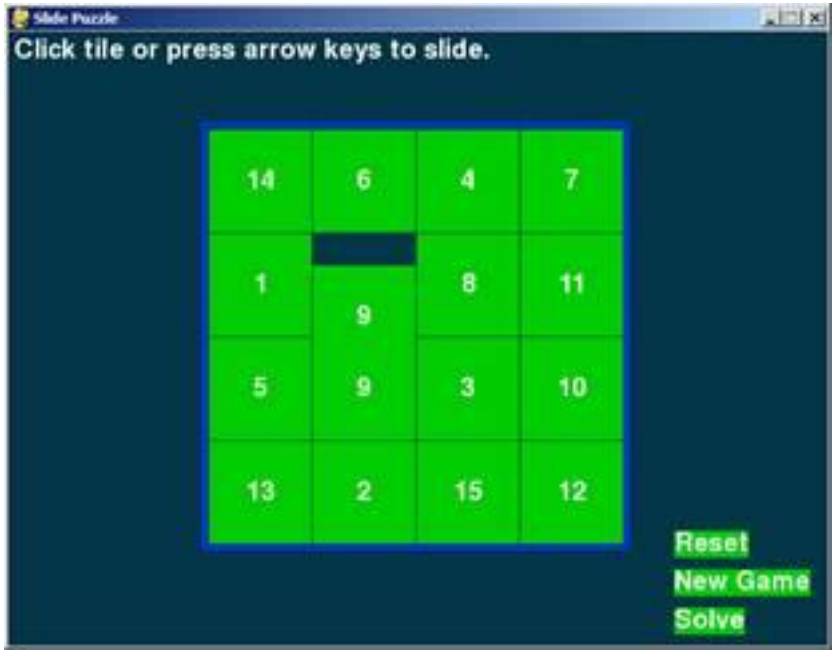
```
254. def slideAnimation(board, direction, message, animationSpeed):
255.     # Note: This function does not check if the move is valid.
256.
257.     blankx, blanky = getBlankPosition(board)
258.     if direction == UP:
259.         movex = blankx
260.         movey = blanky + 1
261.     elif direction == DOWN:
262.         movex = blankx
263.         movey = blanky - 1
264.     elif direction == LEFT:
265.         movex = blankx + 1
266.         movey = blanky
267.     elif direction == RIGHT:
268.         movex = blankx - 1
269.         movey = blanky
```

The copy() Surface Method

```
271.    # prepare the base surface
272.    drawBoard(board, message)
273.    baseSurf = DISPLAYSURF.copy()
 274.    # draw a blank space over the moving tile on the baseSurf Surface.
275.    moveLeft, moveTop = getLeftTopOfTile(movex, movey)
276.    pygame.draw.rect(baseSurf, BGCOLOR, (moveLeft, moveTop,
TILESIZE, TILESIZE))
```

- In that case, here is what the baseSurf Surface would look like:
- And then what it would look like when we draw the “9” tile sliding upwards on top of it:





Notice that while the animation is happening, any events being created by the user are not being handled.

```
278.    for i in range(0, TILESIZE, animationSpeed):
279.        # animate the tile sliding over
280.        checkForQuit()
281.        DISPLAYSURF.blit(baseSurf, (0, 0))
282.        if direction == UP:
283.            drawTile(movex, movey, board[movex][movey], 0, -i)
284.        if direction == DOWN:
285.            drawTile(movex, movey, board[movex][movey], 0, i)
286.        if direction == LEFT:
287.            drawTile(movex, movey, board[movex][movey], -i, 0)
288.        if direction == RIGHT:
289.            drawTile(movex, movey, board[movex][movey], i, 0)
290.
291.        pygame.display.update()
292.        FPSCLOCK.tick(FPS)
```

Creating a New Puzzle

```
295. def generateNewPuzzle(numSlides):
296.     # From a starting configuration, make numSlides number of moves (and
297.     # animate these moves).
298.     sequence = []
299.     board = getStartingBoard()
300.     drawBoard(board, '')
301.     pygame.display.update()
302.     pygame.time.wait(500) # pause 500 milliseconds for effect
```

```
303.    lastMove = None
304.    for i in range(numSlides):
305.        move = getRandomMove(board, lastMove)
306.        slideAnimation(board, move, 'Generating new puzzle...',
int(TILESIZE / 3))
307.        makeMove(board, move)
308.        sequence.append(move)
309.        lastMove = move
310.    return (board, sequence)
```

Animating the Board Reset

```
313. def resetAnimation(board, allMoves):
314.     # make all of the moves in allMoves in reverse.
315.     revAllMoves = allMoves[:] # gets a copy of the list
316.     revAllMoves.reverse()
 317.
318.     for move in revAllMoves:
319.         if move == UP:
 320.             oppositeMove = DOWN
321.         elif move == DOWN:
322.             oppositeMove = UP
323.         elif move == RIGHT:
324.             oppositeMove = LEFT
325.         elif move == LEFT:
326.             oppositeMove = RIGHT
327.         slideAnimation(board, oppositeMove, '', int(TILESIZE TILESIZE / 2))
328.         makeMove(board, oppositeMove)
```

We are familiar with this?

```
331. if __name__ == '__main__':
332.     main()
```

Time vs. Memory Tradeoffs

- There are different ways to write the Slide Puzzle game.
- The most common differences are making tradeoffs between execution time and memory usage.
  - The faster a program can run, the better it is.
- It's also good to use the least amount of memory possible.

Speeding up

- Consider the getBlankPosition() function.
  - This function takes time to run, since it goes through all the possible board coordinates to find where the blank space is.
  - Instead, we could just have a blankspacex and blankspacey variable which would have these XY coordinates so we would not have to look through the entire board each time we want to know where it was.

Decluttering memory

- We keep a board data structure in the solved state in the SOLVEDBOARD variable, so that we can compare the current board to SOLVEDBOARD to see if the player has solved the puzzle.
  - Each time we wanted to do this check, we could just call the getStartingBoard() function and compare the returned value to the current board.
  - Then we would not need the SOLVEDBOARD variable.
- BUT what about the code readability?

Nobody Cares About a Few Bytes

- Also, there is one thing that might seem kind of silly to say in this book because it seem obvious, but many people wonder about it.
- You should know that using short variable names like x or num instead of longer, more descriptive variable names like blankx or numSlides does not save you any memory when your program actually runs.
- Using these longer variable names is better because they'll make your program more readable.
- You might also come up with some clever tricks that do save a few bytes of memory here and there.
- One trick is that when you no longer need a variable, you can reuse that variable name for a different purpose instead of just using two differently named variables.
- Try to avoid the temptation to do this.
- Usually, these tricks reduce code readability and make it harder to debug your programs.
- Modern computers have billions of bytes of memory, and saving a few bytes here and there really isn't worth making the code more confusing for human programmers.

Nobody Cares About a Few Million Nanoseconds

- Similarly, there are times when you can rearrange your code in some way to make it slightly faster by a few nanoseconds.
- These tricks also usually make the code harder to read.
- When you consider that several billion nanoseconds have passed in the time it takes you to read this sentence, saving a few nanoseconds of execution time in your program won’t be noticed by the player.