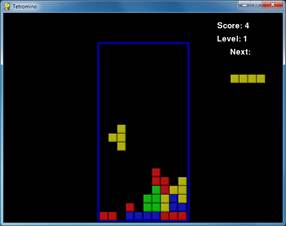
TETROMINO



**Cómo jugar tetromino**

Tetromino es un clon de Tetris. Bloques de formas diferentes (cada uno compuesto por cuatro cajas) caen desde la parte superior de la pantalla, y el jugador debe guiarlos hacia abajo para formar filas completas que no tengan espacios. Cuando se forma una fila completa, la fila desaparece y cada fila de arriba se mueve hacia abajo una fila. El jugador intenta seguir formando líneas completas hasta que la pantalla se llena y un nuevo bloque que cae no cabe en la pantalla.

**Alguna nomenclatura de tetromino**

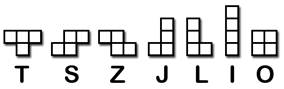
En este capítulo, he creado un conjunto de términos para las diferentes cosas en el programa del juego.

         **Tablero** : el tablero está formado por 10 x 20 espacios en los que los bloques caen y se apilan.

         **Caja** : una caja es un único espacio cuadrado rellenado en el tablero.

         **Pieza** : las cosas que caen desde la parte superior del tablero que el jugador puede rotar y colocar. Cada pieza tiene una forma y se compone de 4 cajas.

         **Forma** : las formas son los diferentes tipos de piezas del juego. Los nombres de las formas son T, S, Z, J, L, I y O.



         **Plantilla** : una lista de estructuras de datos de formas que representa todas las rotaciones posibles de una forma. Estos se almacenan en variables con nombres como S\_SHAPE\_TEMPLATE o J\_SHAPE\_TEMPLATE .

         **Aterrizado** : cuando una pieza ha alcanzado el fondo del tablero o está tocando una caja en el tablero, decimos que la pieza ha aterrizado. En ese punto, la siguiente pieza debería comenzar a caer.

**Código fuente a Tetromino**

Este código fuente se puede descargar desde <http://invpy.com/tetromino.py> . Si recibe algún mensaje de error, mire el número de línea que se menciona en el mensaje de error y verifique su código para detectar errores tipográficos. También puede copiar y pegar su código en el formulario web en <http://invpy.com/diff/tetromino> para ver si [existen](http://invpy.com/diff/tetromino) diferencias entre su código y el código del libro.

También necesitará los archivos de música de fondo en la misma carpeta que el archivo *tetromino.py* . Usted puede descargarlos desde aquí:

         [Http://invpy.com/tetrisb.mid](http://invpy.com/tetrisb.mid)

         [Http://invpy.com/tetrisc.mid](http://invpy.com/tetrisc.mid)

  1. # Tetromino (a Tetris clone)

  2.

  3.

  4. # Creative Commons BY-NC-SA 3.0 US

  5.

  6. import random, time, pygame, sys

  7. from pygame.locals import \*

  8.

  9. FPS = 25

 10. WINDOWWIDTH = 640

 11. WINDOWHEIGHT = 480

 12. BOXSIZE = 20

 13. BOARDWIDTH = 10

 14. BOARDHEIGHT = 20

 15. BLANK = '.'

 16.

 17. MOVESIDEWAYSFREQ = 0.15

 18. MOVEDOWNFREQ = 0.1

 19.

 20. XMARGIN = int((WINDOWWIDTH - BOARDWIDTH \* BOXSIZE) / 2)

 21. TOPMARGIN = WINDOWHEIGHT - (BOARDHEIGHT \* BOXSIZE) - 5

 22.

 23. #               R    G    B

 24. WHITE       = (255, 255, 255)

 25. GRAY        = (185, 185, 185)

 26. BLACK       = (  0,   0,   0)

 27. RED         = (155,   0,   0)

 28. LIGHTRED    = (175,  20,  20)

 29. GREEN       = (  0, 155,   0)

 30. LIGHTGREEN  = ( 20, 175,  20)

 31. BLUE        = (  0,   0, 155)

 32. LIGHTBLUE   = ( 20,  20, 175)

 33. YELLOW      = (155, 155,   0)

 34. LIGHTYELLOW = (175, 175,  20)

 35.

 36. BORDERCOLOR = BLUE

 37. BGCOLOR = BLACK

 38. TEXTCOLOR = WHITE

 39. TEXTSHADOWCOLOR = GRAY

 40. COLORS      = (     BLUE,      GREEN,      RED,      YELLOW)

 41. LIGHTCOLORS = (LIGHTBLUE, LIGHTGREEN, LIGHTRED, LIGHTYELLOW)

 42. assert len(COLORS) == len(LIGHTCOLORS) # each color must have light color

 43.

 44. TEMPLATEWIDTH = 5

 45. TEMPLATEHEIGHT = 5

 46.

 47. S\_SHAPE\_TEMPLATE = [['.....',

 48.                      '.....',

 49.                      '..OO.',

 50.                      '.OO..',

 51.                      '.....'],

 52.                     ['.....',

 53.                      '..O..',

 54.                      '..OO.',

 55.                      '...O.',

 56.                      '.....']]

 57.

 58. Z\_SHAPE\_TEMPLATE = [['.....',

 59.                      '.....',

 60.                      '.OO..',

 61.                      '..OO.',

 62.                      '.....'],

 63.                     ['.....',

 64.                      '..O..',

 65.                      '.OO..',

 66.                      '.O...',

 67.                      '.....']]

 68.

 69. I\_SHAPE\_TEMPLATE = [['..O..',

 70.                      '..O..',

 71.                      '..O..',

 72.                      '..O..',

 73.                      '.....'],

 74.                     ['.....',

 75.                      '.....',

 76.                      'OOOO.',

 77.                      '.....',

 78.                      '.....']]

 79.

 80. O\_SHAPE\_TEMPLATE = [['.....',

 81.                      '.....',

 82.                      '.OO..',

 83.                      '.OO..',

 84.                      '.....']]

 85.

 86. J\_SHAPE\_TEMPLATE = [['.....',

 87.                      '.O...',

 88.                      '.OOO.',

 89.                      '.....',

 90.                      '.....'],

 91.                     ['.....',

 92.                      '..OO.',

 93.                      '..O..',

 94.                      '..O..',

 95.                      '.....'],

 96.                     ['.....',

 97.                      '.....',

 98.                      '.OOO.',

 99.                      '...O.',

100.                      '.....'],

101.                     ['.....',

102.                      '..O..',

103.                      '..O..',

104.                      '.OO..',

105.                      '.....']]

106.

107. L\_SHAPE\_TEMPLATE = [['.....',

108.                      '...O.',

109.                      '.OOO.',

110.                      '.....',

111.                      '.....'],

112.                     ['.....',

113.                      '..O..',

114.                      '..O..',

115.                      '..OO.',

116.                      '.....'],

117.                     ['.....',

118.                      '.....',

119.                      '.OOO.',

120.                      '.O...',

121.                      '.....'],

122.                     ['.....',

123.                      '.OO..',

124.                      '..O..',

125.                      '..O..',

126.                      '.....']]

127.

128. T\_SHAPE\_TEMPLATE = [['.....',

129.                      '..O..',

130.                      '.OOO.',

131.                      '.....',

132.                      '.....'],

133.                     ['.....',

134.                      '..O..',

135.                      '..OO.',

136.                      '..O..',

137.                      '.....'],

138.                     ['.....',

139.                      '.....',

140.                      '.OOO.',

141.                      '..O..',

142.                      '.....'],

143.                     ['.....',

144.                      '..O..',

145.                      '.OO..',

146.                      '..O..',

147.                      '.....']]

148.

149. SHAPES = {'S': S\_SHAPE\_TEMPLATE,

150.           'Z': Z\_SHAPE\_TEMPLATE,

151.           'J': J\_SHAPE\_TEMPLATE,

152.           'L': L\_SHAPE\_TEMPLATE,

153.           'I': I\_SHAPE\_TEMPLATE,

154.           'O': O\_SHAPE\_TEMPLATE,

155.           'T': T\_SHAPE\_TEMPLATE}

156.

157.

158. def main():

159.     global FPSCLOCK, DISPLAYSURF, BASICFONT, BIGFONT

160.     pygame.init()

161.     FPSCLOCK = pygame.time.Clock()

162.     DISPLAYSURF = pygame.display.set\_mode((WINDOWWIDTH, WINDOWHEIGHT))

163.     BASICFONT = pygame.font.Font('freesansbold.ttf', 18)

164.     BIGFONT = pygame.font.Font('freesansbold.ttf', 100)

165.     pygame.display.set\_caption('Tetromino')

166.

167.     showTextScreen('Tetromino')

168.     while True: # game loop

169.         if random.randint(0, 1) == 0:

170.             pygame.mixer.music.load('tetrisb.mid')

171.         else:

172.             pygame.mixer.music.load('tetrisc.mid')

173.         pygame.mixer.music.play(-1, 0.0)

174.         runGame()

175.         pygame.mixer.music.stop()

176.         showTextScreen('Game Over')

177.

178.

179. def runGame():

180.     # setup variables for the start of the game

181.     board = getBlankBoard()

182.     lastMoveDownTime = time.time()

183.     lastMoveSidewaysTime = time.time()

184.     lastFallTime = time.time()

185.     movingDown = False # note: there is no movingUp variable

186.     movingLeft = False

187.     movingRight = False

188.     score = 0

189.     level, fallFreq = calculateLevelAndFallFreq(score)

190.

191.     fallingPiece = getNewPiece()

192.     nextPiece = getNewPiece()

193.

194.     while True: # main game loop

195.         if fallingPiece == None:

196.             # No falling piece in play, so start a new piece at the top

197.             fallingPiece = nextPiece

198.             nextPiece = getNewPiece()

199.             lastFallTime = time.time() # reset lastFallTime

200.

201.             if not isValidPosition(board, fallingPiece):

202.                 return # can't fit a new piece on the board, so game over

203.

204.         checkForQuit()

205.         for event in pygame.event.get(): # event handling loop

206.             if event.type == KEYUP:

207.                 if (event.key == K\_p):

208.                     # Pausing the game

209.                     DISPLAYSURF.fill(BGCOLOR)

210.                     pygame.mixer.music.stop()

211.                     showTextScreen('Paused') # pause until a key press

212.                     pygame.mixer.music.play(-1, 0.0)

213.                     lastFallTime = time.time()

214.                     lastMoveDownTime = time.time()

215.                     lastMoveSidewaysTime = time.time()

216.                 elif (event.key == K\_LEFT or event.key == K\_a):

217.                     movingLeft = False

218.                 elif (event.key == K\_RIGHT or event.key == K\_d):

219.                     movingRight = False

220.                 elif (event.key == K\_DOWN or event.key == K\_s):

221.                     movingDown = False

222.

223.             elif event.type == KEYDOWN:

224.                 # moving the block sideways

225.                 if (event.key == K\_LEFT or event.key == K\_a) and isValidPosition(board, fallingPiece, adjX=-1):

226.                     fallingPiece['x'] -= 1

227.                     movingLeft = True

228.                     movingRight = False

229.                     lastMoveSidewaysTime = time.time()

230.

231.                 elif (event.key == K\_RIGHT or event.key == K\_d) and isValidPosition(board, fallingPiece, adjX=1):

232.                     fallingPiece['x'] += 1

233.                     movingRight = True

234.                     movingLeft = False

235.                     lastMoveSidewaysTime = time.time()

236.

237.                 # rotating the block (if there is room to rotate)

238.                 elif (event.key == K\_UP or event.key == K\_w):

239.                     fallingPiece['rotation'] = (fallingPiece['rotation'] + 1) % len(SHAPES[fallingPiece['shape']])

240.                     if not isValidPosition(board, fallingPiece):

241.                         fallingPiece['rotation'] = (fallingPiece['rotation'] - 1) % len(SHAPES[fallingPiece['shape']])

242.                 elif (event.key == K\_q): # rotate the other direction

243.                     fallingPiece['rotation'] = (fallingPiece['rotation'] - 1) % len(SHAPES[fallingPiece['shape']])

244.                     if not isValidPosition(board, fallingPiece):

245.                         fallingPiece['rotation'] = (fallingPiece['rotation'] + 1) % len(SHAPES[fallingPiece['shape']])

246.

247.                 # making the block fall faster with the down key

248.                 elif (event.key == K\_DOWN or event.key == K\_s):

249.                     movingDown = True

250.                     if isValidPosition(board, fallingPiece, adjY=1):

251.                         fallingPiece['y'] += 1

252.                     lastMoveDownTime = time.time()

253.

254.                 # move the current block all the way down

255.                 elif event.key == K\_SPACE:

256.                     movingDown = False

257.                     movingLeft = False

258.                     movingRight = False

259.                     for i in range(1, BOARDHEIGHT):

260.                         if not isValidPosition(board, fallingPiece, adjY=i):

261.                             break

262.                     fallingPiece['y'] += i - 1

263.

264.         # handle moving the block because of user input

265.         if (movingLeft or movingRight) and time.time() - lastMoveSidewaysTime > MOVESIDEWAYSFREQ:

266.             if movingLeft and isValidPosition(board, fallingPiece, adjX=-1):

267.                 fallingPiece['x'] -= 1

268.             elif movingRight and isValidPosition(board, fallingPiece, adjX=1):

269.                 fallingPiece['x'] += 1

270.             lastMoveSidewaysTime = time.time()

271.

272.         if movingDown and time.time() - lastMoveDownTime > MOVEDOWNFREQ and isValidPosition(board, fallingPiece, adjY=1):

273.             fallingPiece['y'] += 1

274.             lastMoveDownTime = time.time()

275.

276.         # let the piece fall if it is time to fall

277.         if time.time() - lastFallTime > fallFreq:

278.             # see if the piece has landed

279.             if not isValidPosition(board, fallingPiece, adjY=1):

280.                 # falling piece has landed, set it on the board

281.                 addToBoard(board, fallingPiece)

282.                 score += removeCompleteLines(board)

283.                 level, fallFreq = calculateLevelAndFallFreq(score)

284.                 fallingPiece = None

285.             else:

286.                 # piece did not land, just move the block down

287.                 fallingPiece['y'] += 1

288.                 lastFallTime = time.time()

289.

290.         # drawing everything on the screen

291.         DISPLAYSURF.fill(BGCOLOR)

292.         drawBoard(board)

293.         drawStatus(score, level)

294.         drawNextPiece(nextPiece)

295.         if fallingPiece != None:

296.             drawPiece(fallingPiece)

297.

298.         pygame.display.update()

299.         FPSCLOCK.tick(FPS)

300.

301.

302. def makeTextObjs(text, font, color):

303.     surf = font.render(text, True, color)

304.     return surf, surf.get\_rect()

305.

306.

307. def terminate():

308.     pygame.quit()

309.     sys.exit()

310.

311.

312. def checkForKeyPress():

313.     # Go through event queue looking for a KEYUP event.

314.     # Grab KEYDOWN events to remove them from the event queue.

315.     checkForQuit()

316.

317.     for event in pygame.event.get([KEYDOWN, KEYUP]):

318.         if event.type == KEYDOWN:

319.             continue

320.         return event.key

321.     return None

322.

323.

324. def showTextScreen(text):

325.     # This function displays large text in the

326.     # center of the screen until a key is pressed.

327.     # Draw the text drop shadow

328.     titleSurf, titleRect = makeTextObjs(text, BIGFONT, TEXTSHADOWCOLOR)

329.     titleRect.center = (int(WINDOWWIDTH / 2), int(WINDOWHEIGHT / 2))

330.     DISPLAYSURF.blit(titleSurf, titleRect)

331.

332.     # Draw the text

333.     titleSurf, titleRect = makeTextObjs(text, BIGFONT, TEXTCOLOR)

334.     titleRect.center = (int(WINDOWWIDTH / 2) - 3, int(WINDOWHEIGHT / 2) - 3)

335.     DISPLAYSURF.blit(titleSurf, titleRect)

336.

337.     # Draw the additional "Press a key to play." text.

338.     pressKeySurf, pressKeyRect = makeTextObjs('Press a key to play.', BASICFONT, TEXTCOLOR)

339.     pressKeyRect.center = (int(WINDOWWIDTH / 2), int(WINDOWHEIGHT / 2) + 100)

340.     DISPLAYSURF.blit(pressKeySurf, pressKeyRect)

341.

342.     while checkForKeyPress() == None:

343.         pygame.display.update()

344.         FPSCLOCK.tick()

345.

346.

347. def checkForQuit():

348.     for event in pygame.event.get(QUIT): # get all the QUIT events

349.         terminate() # terminate if any QUIT events are present

350.     for event in pygame.event.get(KEYUP): # get all the KEYUP events

351.         if event.key == K\_ESCAPE:

352.             terminate() # terminate if the KEYUP event was for the Esc key

353.         pygame.event.post(event) # put the other KEYUP event objects back

354.

355.

356. def calculateLevelAndFallFreq(score):

357.     # Based on the score, return the level the player is on and

358.     # how many seconds pass until a falling piece falls one space.

359.     level = int(score / 10) + 1

360.     fallFreq = 0.27 - (level \* 0.02)

361.     return level, fallFreq

362.

363. def getNewPiece():

364.     # return a random new piece in a random rotation and color

365.     shape = random.choice(list(SHAPES.keys()))

366.     newPiece = {'shape': shape,

367.                 'rotation': random.randint(0, len(SHAPES[shape]) - 1),

368.                 'x': int(BOARDWIDTH / 2) - int(TEMPLATEWIDTH / 2),

369.                 'y': -2, # start it above the board (i.e. less than 0)

370.                 'color': random.randint(0, len(COLORS)-1)}

371.     return newPiece

372.

373.

374. def addToBoard(board, piece):

375.     # fill in the board based on piece's location, shape, and rotation

376.     for x in range(TEMPLATEWIDTH):

377.         for y in range(TEMPLATEHEIGHT):

378.             if SHAPES[piece['shape']][piece['rotation']][y][x] != BLANK:

379.                 board[x + piece['x']][y + piece['y']] = piece['color']

380.

381.

382. def getBlankBoard():

383.     # create and return a new blank board data structure

384.     board = []

385.     for i in range(BOARDWIDTH):

386.         board.append([BLANK] \* BOARDHEIGHT)

387.     return board

388.

389.

390. def isOnBoard(x, y):

391.     return x >= 0 and x < BOARDWIDTH and y < BOARDHEIGHT

392.

393.

394. def isValidPosition(board, piece, adjX=0, adjY=0):

395.     # Return True if the piece is within the board and not colliding

396.     for x in range(TEMPLATEWIDTH):

397.         for y in range(TEMPLATEHEIGHT):

398.             isAboveBoard = y + piece['y'] + adjY < 0

399.             if isAboveBoard or SHAPES[piece['shape']][piece['rotation']][y][x] == BLANK:

400.                 continue

401.             if not isOnBoard(x + piece['x'] + adjX, y + piece['y'] + adjY):

402.                 return False

403.             if board[x + piece['x'] + adjX][y + piece['y'] + adjY] != BLANK:

404.                 return False

405.     return True

406.

407. def isCompleteLine(board, y):

408.     # Return True if the line filled with boxes with no gaps.

409.     for x in range(BOARDWIDTH):

410.         if board[x][y] == BLANK:

411.             return False

412.     return True

413.

414.

415. def removeCompleteLines(board):

416.     # Remove any completed lines on the board, move everything above them down, and return the number of complete lines.

417.     numLinesRemoved = 0

418.     y = BOARDHEIGHT - 1 # start y at the bottom of the board

419.     while y >= 0:

420.         if isCompleteLine(board, y):

421.             # Remove the line and pull boxes down by one line.

422.             for pullDownY in range(y, 0, -1):

423.                 for x in range(BOARDWIDTH):

424.                     board[x][pullDownY] = board[x][pullDownY-1]

425.             # Set very top line to blank.

426.             for x in range(BOARDWIDTH):

427.                 board[x][0] = BLANK

428.             numLinesRemoved += 1

429.             # Note on the next iteration of the loop, y is the same.

430.             # This is so that if the line that was pulled down is also

431.             # complete, it will be removed.

432.         else:

433.             y -= 1 # move on to check next row up

434.     return numLinesRemoved

435.

436.

437. def convertToPixelCoords(boxx, boxy):

438.     # Convert the given xy coordinates of the board to xy

439.     # coordinates of the location on the screen.

440.     return (XMARGIN + (boxx \* BOXSIZE)), (TOPMARGIN + (boxy \* BOXSIZE))

441.

442.

443. def drawBox(boxx, boxy, color, pixelx=None, pixely=None):

444.     # draw a single box (each tetromino piece has four boxes)

445.     # at xy coordinates on the board. Or, if pixelx & pixely

446.     # are specified, draw to the pixel coordinates stored in

447.     # pixelx & pixely (this is used for the "Next" piece).

448.     if color == BLANK:

449.         return

450.     if pixelx == None and pixely == None:

451.         pixelx, pixely = convertToPixelCoords(boxx, boxy)

452.     pygame.draw.rect(DISPLAYSURF, COLORS[color], (pixelx + 1, pixely + 1, BOXSIZE - 1, BOXSIZE - 1))

453.     pygame.draw.rect(DISPLAYSURF, LIGHTCOLORS[color], (pixelx + 1, pixely + 1, BOXSIZE - 4, BOXSIZE - 4))

454.

455.

456. def drawBoard(board):

457.     # draw the border around the board

458.     pygame.draw.rect(DISPLAYSURF, BORDERCOLOR, (XMARGIN - 3, TOPMARGIN - 7, (BOARDWIDTH \* BOXSIZE) + 8, (BOARDHEIGHT \* BOXSIZE) + 8), 5)

459.

460.     # fill the background of the board

461.     pygame.draw.rect(DISPLAYSURF, BGCOLOR, (XMARGIN, TOPMARGIN, BOXSIZE \* BOARDWIDTH, BOXSIZE \* BOARDHEIGHT))

462.     # draw the individual boxes on the board

463.     for x in range(BOARDWIDTH):

464.         for y in range(BOARDHEIGHT):

465.             drawBox(x, y, board[x][y])

466.

467.

468. def drawStatus(score, level):

469.     # draw the score text

470.     scoreSurf = BASICFONT.render('Score: %s' % score, True, TEXTCOLOR)

471.     scoreRect = scoreSurf.get\_rect()

472.     scoreRect.topleft = (WINDOWWIDTH - 150, 20)

473.     DISPLAYSURF.blit(scoreSurf, scoreRect)

474.

475.     # draw the level text

476.     levelSurf = BASICFONT.render('Level: %s' % level, True, TEXTCOLOR)

477.     levelRect = levelSurf.get\_rect()

478.     levelRect.topleft = (WINDOWWIDTH - 150, 50)

479.     DISPLAYSURF.blit(levelSurf, levelRect)

480.

481.

482. def drawPiece(piece, pixelx=None, pixely=None):

483.     shapeToDraw = SHAPES[piece['shape']][piece['rotation']]

484.     if pixelx == None and pixely == None:

485.         # if pixelx & pixely hasn't been specified, use the location stored in the piece data structure

486.         pixelx, pixely = convertToPixelCoords(piece['x'], piece['y'])

487.

488.     # draw each of the blocks that make up the piece

489.     for x in range(TEMPLATEWIDTH):

490.         for y in range(TEMPLATEHEIGHT):

491.             if shapeToDraw[y][x] != BLANK:

492.                 drawBox(None, None, piece['color'], pixelx + (x \* BOXSIZE), pixely + (y \* BOXSIZE))

493.

494.

495. def drawNextPiece(piece):

496.     # draw the "next" text

497.     nextSurf = BASICFONT.render('Next:', True, TEXTCOLOR)

498.     nextRect = nextSurf.get\_rect()

499.     nextRect.topleft = (WINDOWWIDTH - 120, 80)

500.     DISPLAYSURF.blit(nextSurf, nextRect)

501.     # draw the "next" piece

502.     drawPiece(piece, pixelx=WINDOWWIDTH-120, pixely=100)

503.

504.

505. if \_\_name\_\_ == '\_\_main\_\_':

506.     main()

**El código de configuración habitual**

1. # Tetromino (a Tetris clone)

  2.

  3.

  4.

  5.

  6. import random, time, pygame, sys

  7. from pygame.locals import \*

  8.

  9. FPS = 25

 10. WINDOWWIDTH = 640

 11. WINDOWHEIGHT = 480

 12. BOXSIZE = 20

 13. BOARDWIDTH = 10

 14. BOARDHEIGHT = 20

 15. BLANK = '.'

Estas son las constantes utilizadas por nuestro juego Tetromino. Cada caja es un cuadrado de 20 píxeles de ancho y alto. El tablero en sí es de 10 cajas de ancho y 20 cajas de altura. La constante EN BLANCOse utilizará como un valor para representar espacios en blanco en la estructura de datos de la pizarra.

**Configuración de constantes de tiempo para mantener pulsadas las teclas**

 17. MOVESIDEWAYSFREQ = 0.15

 18. MOVEDOWNFREQ = 0.1

Cada vez que el jugador empuja la tecla de flecha hacia la izquierda o hacia la derecha, la pieza que cae debe mover un cuadro hacia la izquierda o hacia la derecha, respectivamente. Sin embargo, el jugador también puede mantener presionada la tecla de flecha izquierda o derecha para seguir moviendo la pieza que cae. La constante MOVESIDEWAYSFREQ lo establecerá de modo que cada 0.15 segundos que pase con la tecla de flecha izquierda o derecha presionada, la pieza se moverá otro espacio.

La constante MOVEDOWNFREQ es la misma cosa, excepto que indica con qué frecuencia la pieza cae en una casilla mientras el jugador mantiene presionada la tecla de flecha hacia abajo.

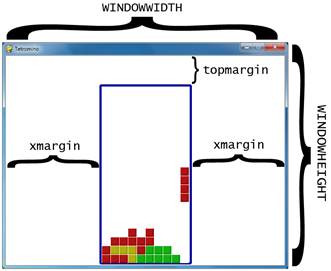
**Más código de configuración**

 20. XMARGIN = int((WINDOWWIDTH - BOARDWIDTH \* BOXSIZE) / 2)

 21. TOPMARGIN = WINDOWHEIGHT - (BOARDHEIGHT \* BOXSIZE) - 5

El programa necesita calcular cuántos píxeles hay en el lado izquierdo y derecho de la placa para usar más adelante en el programa. WINDOWWIDTH es el número total de píxeles de ancho que tiene toda la ventana. El tablero tiene una anchura de cajas de ancho de tabla y cada una de ellas tiene una anchura de píxeles de BOXSIZE . Si restamos BOXSIZE píxeles de esto para cada una de las casillas de ancho en el tablero (que es BOARDWIDTH \* BOXSIZE ), tendremos el tamaño del margen a la izquierda y derecha del tablero. Si dividimos esto por 2 , entonces tendremos el tamaño de solo un margen. Como los márgenes son del mismo tamaño, podemos usar XMARGIN para el margen izquierdo o derecho.

Podemos calcular el tamaño del espacio entre la parte superior del tablero y la parte superior de la ventana de una manera similar. La pizarra se dibujará 5 píxeles por encima de la parte inferior de la ventana, por lo que se resta 5 de topmargin para tener en cuenta esto.



 23. #               R    G    B

 24. WHITE       = (255, 255, 255)

 25. GRAY        = (185, 185, 185)

 26. BLACK       = (  0,   0,   0)

 27. RED         = (155,   0,   0)

 28. LIGHTRED    = (175,  20,  20)

 29. GREEN       = (  0, 155,   0)

 30. LIGHTGREEN  = ( 20, 175,  20)

 31. BLUE        = (  0,   0, 155)

 32. LIGHTBLUE   = ( 20,  20, 175)

 33. YELLOW      = (155, 155,   0)

 34. LIGHTYELLOW = (175, 175,  20)

 35.

 36. BORDERCOLOR = BLUE

 37. BGCOLOR = BLACK

 38. TEXTCOLOR = WHITE

 39. TEXTSHADOWCOLOR = GRAY

 40. COLORS      = (     BLUE,      GREEN,      RED,      YELLOW)

 41. LIGHTCOLORS = (LIGHTBLUE, LIGHTGREEN, LIGHTRED, LIGHTYELLOW)

 42. assert len(COLORS) == len(LIGHTCOLORS) # each color must have light color

Las piezas vendrán en cuatro colores: azul, verde, rojo y amarillo. Sin embargo, cuando dibujemos los cuadros, habrá un resaltado delgado en el cuadro en un color más claro. Así que esto significa que también debemos crear colores azul claro, verde claro, rojo claro y amarillo claro.

Cada uno de estos cuatro colores se almacenará en tuplas llamadas COLORS (para los colores normales) y LIGHTCOLORS (para los colores más claros).

**Configuración de las plantillas de pieza**

 44. TEMPLATEWIDTH = 5

 45. TEMPLATEHEIGHT = 5

 46.

 47. S\_SHAPE\_TEMPLATE = [['.....',

 48.                      '.....',

 49.                      '..OO.',

 50.                      '.OO..',

 51.                      '.....'],

 52.                     ['.....',

 53.                      '..O..',

 54.                      '..OO.',

 55.                      '...O.',

 56.                      '.....']]

 57.

 58. Z\_SHAPE\_TEMPLATE = [['.....',

 59.                      '.....',

 60.                      '.OO..',

 61.                      '..OO.',

 62.                      '.....'],

 63.                     ['.....',

 64.                      '..O..',

 65.                      '.OO..',

 66.                      '.O...',

 67.                      '.....']]

 68.

 69. I\_SHAPE\_TEMPLATE = [['..O..',

 70.                      '..O..',

 71.                      '..O..',

 72.                      '..O..',

 73.                      '.....'],

 74.                     ['.....',

 75.                      '.....',

 76.                      'OOOO.',

 77.                      '.....',

 78.                      '.....']]

 79.

 80. O\_SHAPE\_TEMPLATE = [['.....',

 81.                      '.....',

 82.                      '.OO..',

 83.                      '.OO..',

 84.                      '.....']]

 85.

 86. J\_SHAPE\_TEMPLATE = [['.....',

 87.                      '.O...',

 88.                      '.OOO.',

 89.                      '.....',

 90.                      '.....'],

 91.                     ['.....',

 92.                      '..OO.',

 93.                      '..O..',

 94.                      '..O..',

 95.                      '.....'],

 96.                     ['.....',

 97.                      '.....',

 98.                      '.OOO.',

 99.                      '...O.',

100.                      '.....'],

101.                     ['.....',

102.                      '..O..',

103.                      '..O..',

104.                      '.OO..',

105.                      '.....']]

106.

107. L\_SHAPE\_TEMPLATE = [['.....',

108.                      '...O.',

109.                      '.OOO.',

110.                      '.....',

111.                      '.....'],

112.                     ['.....',

113.                      '..O..',

114.                      '..O..',

115.                      '..OO.',

116.                      '.....'],

117.                     ['.....',

118.                      '.....',

119.                      '.OOO.',

120.                      '.O...',

121.                      '.....'],

122.                     ['.....',

123.                      '.OO..',

124.                      '..O..',

125.                      '..O..',

126.                      '.....']]

127.

128. T\_SHAPE\_TEMPLATE = [['.....',

129.                      '..O..',

130.                      '.OOO.',

131.                      '.....',

132.                      '.....'],

133.                     ['.....',

134.                      '..O..',

135.                      '..OO.',

136.                      '..O..',

137.                      '.....'],

138.                     ['.....',

139.                      '.....',

140.                      '.OOO.',

141.                      '..O..',

142.                      '.....'],

143.                     ['.....',

144.                      '..O..',

145.                      '.OO..',

146.                      '..O..',

147.                      '.....']]

Nuestro programa de juego necesita saber cómo se forma cada una de las formas, incluso para todas las rotaciones posibles. Para hacer esto, crearemos listas de listas de cadenas. La lista interna de cadenas representará una sola rotación de una forma, como esta:

['.....',

 '.....',

 '.OO.',

 '.OO ..',

 '.....']

Escribiremos el resto de nuestro código para que interprete una lista de cadenas como la de arriba para representar una forma donde los puntos son espacios vacíos y las O son cuadros, como este:



**División de una "línea de código" en varias líneas**

Puede ver que esta lista se extiende a lo largo de muchas líneas en el editor de archivos. Esto es perfectamente válido para Python, porque el intérprete de Python se da cuenta de que, hasta que ve el ]corchete de cierre, la lista no está terminada. La sangría no importa porque Python sabe que no tendrá una sangría diferente para un nuevo bloque en medio de una lista. Este código de abajo funciona bien:

spam = ['hello', 3.14, 'world', 42, 10, 'fuzz']

huevos = ['hola', 3.14,

   'mundo'

         42

       10, 'fuzz']

Aunque, por supuesto, el código para la lista de huevos sería mucho más legible si alineamos todos los elementos de la lista o los colocamos en una sola línea como spam .

Normalmente, dividir una línea de código en varias líneas en el editor de archivos requeriría poner un carácter \ al final de la línea. El \ le dice a Python, "Este código continúa en la siguiente línea". (Esta barra se utilizó por primera vez en el juego Sliding Puzzle en la función isValidMove () ).

Haremos las estructuras de datos de las formas de la "plantilla" mediante la creación de una lista de estas cadenas de cadenas y las almacenaremos en variables como S\_SHAPE\_TEMPLATE . De esta manera, len (S\_SHAPE\_TEMPLATE) representará cuántas rotaciones posibles hay para la forma de S, y S\_SHAPE\_TEMPLATE [0] representará la primera rotación posible de la forma de S. Las líneas 47 a 147 crearán estructuras de datos de "plantilla" para cada una de las formas.

Imagine que cada pieza posible en un pequeño tablero de 5 x 5 de espacio vacío, con algunos de los espacios en el tablero rellenados con cajas. Las siguientes expresiones que usan S\_SHAPE\_TEMPLATE [0] son verdaderas :

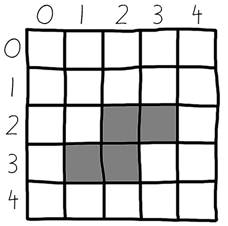
S\_SHAPE\_TEMPLATE [0] [2] [2] == 'O'

S\_SHAPE\_TEMPLATE [0] [2] [3] == 'O'

S\_SHAPE\_TEMPLATE [0] [3] [1] == 'O'

S\_SHAPE\_TEMPLATE [0] [3] [2] == 'O'

Si representáramos esta forma en papel, se vería algo como esto:



Así es como podemos representar cosas como piezas de Tetromino como valores de Python, como cadenas y listas. Las constantes TEMPLATEWIDTH y TEMPLATEHEIGHT simplemente establecen el tamaño de cada fila y columna para la rotación de cada forma. (Las plantillas siempre serán 5x5.)

149. SHAPES = {'S': S\_SHAPE\_TEMPLATE,

150.           'Z': Z\_SHAPE\_TEMPLATE,

151.           'J': J\_SHAPE\_TEMPLATE,

152.           'L': L\_SHAPE\_TEMPLATE,

153.           'I': I\_SHAPE\_TEMPLATE,

154.           'O': O\_SHAPE\_TEMPLATE,

155.           'T': T\_SHAPE\_TEMPLATE}

La variable SHAPES será un diccionario que almacena todas las plantillas diferentes. Debido a que cada plantilla tiene todas las rotaciones posibles de una sola forma, esto significa que la variable SHAPEScontiene todas las rotaciones posibles de cada forma posible. Esta será la estructura de datos que contiene todos los datos de formas en nuestro juego.

**El principal) ( Función**

158. def main():

159.     global FPSCLOCK, DISPLAYSURF, BASICFONT, BIGFONT

160.     pygame.init()

161.     FPSCLOCK = pygame.time.Clock()

162.     DISPLAYSURF = pygame.display.set\_mode((WINDOWWIDTH, WINDOWHEIGHT))

163.     BASICFONT = pygame.font.Font('freesansbold.ttf', 18)

164.     BIGFONT = pygame.font.Font('freesansbold.ttf', 100)

165.     pygame.display.set\_caption('Tetromino')

166.

167.     showTextScreen('Tetromino')

La función main () maneja la creación de algunas constantes más globales y muestra la pantalla de inicio que aparece cuando se ejecuta el programa.

168.     while True: # game loop

169.         if random.randint(0, 1) == 0:

170.             pygame.mixer.music.load('tetrisb.mid')

171.         else:

172.             pygame.mixer.music.load('tetrisc.mid')

173.         pygame.mixer.music.play(-1, 0.0)

174.         runGame()

175.         pygame.mixer.music.stop()

176.         showTextScreen('Game Over')

El código para el juego real está todo en runGame () . La función main () aquí simplemente decide aleatoriamente qué música de fondo comenzar a reproducir (ya sea el archivo de música MIDI *tetrisb.mid* o *tetrisc.mid* ), luego llama a runGame () para comenzar el juego. Cuando el jugador pierde, runGame () regresará a main () , que luego detiene la música de fondo y muestra el juego en la pantalla.

Cuando el jugador presiona una tecla, la función showTextScreen () que muestra el juego sobre la pantalla regresará. El bucle del juego volverá al principio en la línea 169 y comenzará otro juego.

**El comienzo de un nuevo juego**

179. def runGame():

180.     # setup variables for the start of the game

181.     board = getBlankBoard()

182.     lastMoveDownTime = time.time()

183.     lastMoveSidewaysTime = time.time()

184.     lastFallTime = time.time()

185.     movingDown = False # note: there is no movingUp variable

186.     movingLeft = False

187.     movingRight = False

188.     score = 0

189.     level, fallFreq = calculateLevelAndFallFreq(score)

190.

191.     fallingPiece = getNewPiece()

192.     nextPiece = getNewPiece()

Antes de que el juego comience y las piezas comiencen a caer, debemos inicializar algunas variables a sus valores de inicio del juego. En la línea 191, la variable fallingPiece se establecerá en la pieza que está cayendo y que el jugador puede rotar. En la línea 192, la variable nextPiece se establecerá en la pieza que se muestra en la parte "Siguiente" de la pantalla para que el jugador sepa qué pieza está subiendo después de configurar la pieza que cae.

**The Game Loop**

194.     while True: # main game loop

195.         if fallingPiece == None:

196.             # No falling piece in play, so start a new piece at the top

197.             fallingPiece = nextPiece

198.             nextPiece = getNewPiece()

199.             lastFallTime = time.time() # reset lastFallTime

200.

201.             if not isValidPosition(board, fallingPiece):

202.                 return # can't fit a new piece on the board, so game over

203.

204.         checkForQuit()

El bucle principal del juego que comienza en la línea 194 maneja todo el código de la parte principal del juego cuando las piezas caen al final. La variable fallingPiece se establece en Ninguno después de que la pieza descendente haya aterrizado. Esto significa que la pieza en nextPiece debe copiarse a la variable fallingPiece , y una nueva pieza aleatoria debe colocarse en la variable nextPiece . Se puede generar una nueva pieza desde la función getNewPiece () . La variable lastFallTime también se restablece a la hora actual para que la pieza caiga, sin embargo, muchos segundos están en fallFreq .

Las piezas que obtienenNewPiece () se colocan un poco por encima del tablero, generalmente con parte de la pieza ya en el tablero. Pero si esta posición no es válida porque el tablero ya está lleno allí (en cuyo caso la llamada isValidPosition () en la línea 201 devolverá Falso ), entonces sabemos que el tablero está lleno y el jugador debería perder el juego. Cuando esto sucede, la función runGame () regresa.

**El evento Handling Loop**

205.         for event in pygame.event.get(): # event handling loop

206.             if event.type == KEYUP:

El bucle de manejo de eventos se encarga de cuando el jugador gira la pieza que cae, mueve la pieza que cae o hace una pausa en el juego.

**Pausando el juego**

207.                 if (event.key == K\_p):

208.                     # Pausing the game

209.                     DISPLAYSURF.fill(BGCOLOR)

210.                     pygame.mixer.music.stop()

211.                     showTextScreen('Paused') # pause until a key press

212.                     pygame.mixer.music.play(-1, 0.0)

213.                     lastFallTime = time.time()

214.                     lastMoveDownTime = time.time()

215.                     lastMoveSidewaysTime = time.time()

Si el jugador ha presionado la tecla P, entonces el juego debe pausar. Necesitamos ocultar el tablero al jugador (de lo contrario, el jugador podría hacer trampa haciendo una pausa en el juego y tomando tiempo para decidir dónde mover la pieza).

El código borra la superficie de la pantalla con una llamada a DISPLAYSURF.fill (BGCOLOR) y detiene la música. Se llama a la función showTextScreen () para mostrar el texto "Pausado" y esperar a que el jugador presione una tecla para continuar.

Una vez que el jugador haya presionado una tecla, regresará showTextScreen () . Línea 212 se reiniciará la música de fondo. Además, dado que podría haber pasado una gran cantidad de tiempo desde que el jugador detuvo el juego, las variables lastFallTime , lastMoveDownTime y lastMoveSidewaysTime deberían restablecerse a la hora actual (que se realiza en las líneas 213 a 215).

**Uso de variables de movimiento para manejar la entrada del usuario**

216.                 elif (event.key == K\_LEFT or event.key == K\_a):

217.                     movingLeft = False

218.                 elif (event.key == K\_RIGHT or event.key == K\_d):

219.                     movingRight = False

220.                 elif (event.key == K\_DOWN or event.key == K\_s):

221.                     movingDown = False

Dejar de presionar una de las teclas de flecha (o las teclas WASD) establecerá las variables movingLeft , movingRight o movingDown de nuevo en False , lo que indica que el jugador ya no quiere mover la pieza en esas direcciones. El código posterior manejará qué hacer en función de los valores booleanos dentro de estas variables "en movimiento". Tenga en cuenta que la flecha hacia arriba y las teclas W se utilizan para girar la pieza, no para moverla hacia arriba. Es por esto que no hay una variable movingUp .

**Comprobando si una diapositiva o rotación es válida**

223.             elif event.type == KEYDOWN:

224.                 # moving the block sideways

225.                 if (event.key == K\_LEFT or event.key == K\_a) and isValidPosition(board, fallingPiece, adjX=-1):

226.                     fallingPiece['x'] -= 1

227.                     movingLeft = True

228.                     movingRight = False

229.                     lastMoveSidewaysTime = time.time()

Cuando la tecla de flecha izquierda se presiona hacia abajo (y moverse hacia la izquierda es un movimiento válido para la pieza que cae, según lo determinado por la llamada a isValidPosition () ), entonces debemos cambiar la posición a un espacio a la izquierda restando el valor de fallingPiece ['x'] por 1 . La función isValidPosition () tiene parámetros opcionales llamados adjX y adjY . Normalmente, la función isValidPosition () verifica la posición de los datos proporcionados por el objeto pieza que se pasa para el segundo parámetro. Sin embargo, a veces no queremos comprobar dónde se encuentra actualmente la pieza, sino más bien unos pocos espacios desde esa posición.

Si pasamos -1 para el adjX (un nombre corto para "X ajustado"), entonces no verifica la validez de la posición en la estructura de datos de la pieza, sino si la posición de donde estaría la pieza si fuera Un espacio a la izquierda. Al pasar 1 para adjX se marcaría un espacio a la derecha. También hay un parámetro opcional adjY . Al pasar -1 para adjY se verifica un espacio arriba de la posición actual de la pieza, y al pasar un valor como 3 para adjY se verifican tres espacios hacia abajo desde donde se encuentra la pieza.

La variable movingLeft se establece en Verdadero , y solo para asegurarse de que la pieza que cae no se mueva tanto a la izquierda *como a la* derecha, la variable movingRight se establece en Falso en la línea 228. La variable lastMoveSidewaysTime se actualizará a la hora actual en la línea 229.

Estas variables se configuran para que el jugador solo pueda mantener presionada la tecla de flecha para seguir moviendo la pieza. Si la variable movingLeft está configurada en Verdadero , el programa puede saber que la tecla de flecha izquierda (o la tecla A) se presionó y aún no se ha soltado. Y si 0.15 segundos (el número almacenado en MOVESIDEWAYSFREQ ) ha pasado desde el tiempo almacenado en lastMoveSidewaysTime , entonces es hora de que el programa mueva la pieza descendente nuevamente hacia la izquierda.

El lastMoveSidewaysTime funciona igual que la variable lastClickTime en el capítulo Simular.

231.                 elif (event.key == K\_RIGHT or event.key == K\_d) and isValidPosition(board, fallingPiece, adjX=1):

232.                     fallingPiece['x'] += 1

233.                     movingRight = True

234.                     movingLeft = False

235.                     lastMoveSidewaysTime = time.time()

El código en las líneas 231 a 235 es casi idéntico a las líneas 225 a 229, excepto que maneja el movimiento de la pieza descendente hacia la derecha cuando se presiona la tecla de flecha derecha (o la tecla D).

237.                 # rotating the block (if there is room to rotate)

238.                 elif (event.key == K\_UP or event.key == K\_w):

239.                     fallingPiece['rotation'] = (fallingPiece['rotation'] + 1) % len(SHAPES[fallingPiece['shape']])

La tecla de flecha hacia arriba (o la tecla W) girará la pieza descendente a su próxima rotación. Todo lo que tiene que hacer el código es incrementar el valor de la clave de 'rotación' en el diccionario fallingPiece en 1 . Sin embargo, si el aumento del valor de la clave de 'rotación' lo hace más grande que el número total de rotaciones, entonces "modding" por el número total de rotaciones posibles para esa forma (que es lo que len (SHAPES [fallingPiece ['shape']] )) is) entonces "rodará" a 0 .

Este es un ejemplo de este modding con la forma J, que tiene 4 rotaciones posibles:

>>> 0% 4

0

>>> 1% 4

1

>>> 2% 4

2

>>> 3% 4

3

>>> 5% 4

1

>>> 6% 4

2

>>> 7% 4

3

>>> 8% 4

0

>>>

240.                     if not isValidPosition(board, fallingPiece):

241.                         fallingPiece['rotation'] = (fallingPiece['rotation'] - 1) % len(SHAPES[fallingPiece['shape']])

Si la nueva posición girada no es válida porque se superpone a algunas cajas que ya están en el tablero, queremos volver a la rotación original restando 1 de fallingPiece ['rotación'] . También podemos modificarlo mediante len (SHAPES [fallingPiece ['shape']]) de modo que si el nuevo valor es -1 , el modding volverá a la última rotación de la lista. Aquí hay un ejemplo de modding un número negativo:

>>> -1% 4

3

242.                 elif (event.key == K\_q): # rotate the other direction

243.                     fallingPiece['rotation'] = (fallingPiece['rotation'] - 1) % len(SHAPES[fallingPiece['shape']])

244.                     if not isValidPosition(board, fallingPiece):

245.                         fallingPiece['rotation'] = (fallingPiece['rotation'] + 1) % len(SHAPES[fallingPiece['shape']])

Las líneas 242 a 245 hacen lo mismo 238 a 241, excepto que manejan el caso donde el jugador ha presionado la tecla Q que gira la pieza en la dirección opuesta. En este caso, *restamos*1 de fallingPiece ['rotación'] (que se realiza en la línea 243) en lugar de agregar 1 .

247.                 # making the block fall faster with the down key

248.                 elif (event.key == K\_DOWN or event.key == K\_s):

249.                     movingDown = True

250.                     if isValidPosition(board, fallingPiece, adjY=1):

251.                         fallingPiece['y'] += 1

252.                     lastMoveDownTime = time.time()

Si se presiona la flecha hacia abajo o la tecla S, entonces el jugador desea que la pieza caiga más rápido de lo normal. La línea 251 mueve la pieza hacia abajo un espacio en el tablero (pero solo si es un espacio válido). La variable movingDown se establece en True y lastMoveDownTime se restablece a la hora actual. Estas variables se revisarán más adelante para que la pieza siga cayendo a un ritmo más rápido siempre que la flecha hacia abajo o la tecla S se mantengan presionadas.

**Encontrar el fondo**

254.                 # move the current block all the way down

255.                 elif event.key == K\_SPACE:

256.                     movingDown = False

257.                     movingLeft = False

258.                     movingRight = False

259.                     for i in range(1, BOARDHEIGHT):

260.                         if not isValidPosition(board, fallingPiece, adjY=i):

261.                             break

262.                     fallingPiece['y'] += i - 1

Cuando el jugador presiona la tecla de espacio, la pieza que cae caerá inmediatamente lo más que pueda en el tablero y aterrizará. El programa primero debe averiguar cuántos espacios puede mover la pieza hasta que aterrice.

Las líneas 256 a 258 establecerán todas las variables en movimiento en Falso (lo que hace que el código en partes posteriores de la programación piense que el usuario ha dejado de presionar cualquier tecla de flecha que se presionó). Esto se hace porque este código moverá la pieza al fondo absoluto y comenzará a caer la siguiente pieza, y no queremos sorprender al jugador al hacer que esas piezas comiencen a moverse inmediatamente solo porque estaban presionando una tecla de flecha cuando pulsaron la tecla de espacio.

Para encontrar lo más lejos que pueda caer la pieza, primero debemos llamar a isValidPosition () y pasar el entero 1 para el parámetro adjY . Si isValidPosition () devuelve False , sabemos que la pieza no puede caer más y ya está en la parte inferior. Si isValidPosition () devuelve True , entonces sabemos que puede caer 1 espacio hacia abajo.

En ese caso, deberíamos llamar a isValidPosition () con adjY establecido en 2 . Si devuelve True nuevamente, llamaremos a isValidPosition () con adjY establecido en 3 , y así sucesivamente. Esto es lo que maneja el bucle for en la línea 259: llamar a isValidPosition () con valores enteros crecientes para pasar por adjY hasta que la llamada a la función devuelva False . En ese momento, sabemos que el valor en i es un espacio más allá de la parte inferior. Esta es la razón por la que la línea 262 aumenta la caída de la pieza ['y'] en i - 1 en lugar de i .

(También tenga en cuenta que el segundo parámetro a range () en la línea 259 para la declaración se establece en BOARDHEIGHT porque esta es la cantidad máxima que la pieza podría caer antes de que llegue al fondo del tablero).

**Moviéndose manteniendo pulsada la tecla**

264.         # handle moving the block because of user input

265.         if (movingLeft or movingRight) and time.time() - lastMoveSidewaysTime > MOVESIDEWAYSFREQ:

266.             if movingLeft and isValidPosition(board, fallingPiece, adjX=-1):

267.                 fallingPiece['x'] -= 1

268.             elif movingRight and isValidPosition(board, fallingPiece, adjX=1):

269.                 fallingPiece['x'] += 1

270.             lastMoveSidewaysTime = time.time()

¿Recuerda que en la línea 227 la variable movingLeft se estableció en Verdadero si el jugador presionó la tecla de flecha izquierda? (Lo mismo para la línea 233 en donde movingRight se estableció en Verdadero si el jugador presionó la tecla de la flecha hacia la derecha). Las variables en movimiento se configuraron en Falso si el usuario también dejó estas teclas (vea las líneas 217 y 219).

Lo que también sucedió cuando el jugador presionó la tecla de flecha hacia la izquierda o hacia la derecha fue que la variable lastMoveSidewaysTime se configuró a la hora actual (que era el valor de retorno de time.time () ). Si el jugador continuaba manteniendo presionada la tecla de flecha sin dejar de hacerlo, entonces la variable movingLeft o movingRight todavía se establecería en Verdadero .

Si el usuario mantiene presionada la tecla durante más de 0.15 segundos (el valor almacenado en MOVESIDEWAYSFREQ es el flotante 0.15 ), la expresión time.time () - lastMoveSidewaysTime> MOVESIDEWAYSFREQ se evaluará como Verdadero . La condición de la línea 265 es Verdadera si el usuario ha mantenido presionada la tecla de flecha y pasaron 0.15 segundos, y en ese caso deberíamos mover la pieza descendente hacia la izquierda o hacia la derecha aunque el usuario no haya presionado la tecla de flecha nuevamente.

Esto es muy útil porque sería molesto para el jugador presionar repetidamente las teclas de flecha para que la pieza que cae se mueva sobre múltiples espacios en el tablero. En su lugar, solo pueden mantener presionada una tecla de flecha y la pieza seguirá moviéndose hasta que dejen de tocar la tecla. Cuando eso suceda, el código en las líneas 216 a 221 establecerá la variable en movimiento en Falso y la condición en la línea 265 será Falso . Eso es lo que evita que la pieza que cae se deslice sobre más.

Para demostrar por qué el time.time () - lastMoveSidewaysTime> MOVESIDEWAYSFREQ devuelve True después de que haya transcurrido el número de segundos en MOVESIDEWAYSFREQ, ejecute este breve programa:

tiempo de importación

Hora de esperar = 4

comenzar = tiempo.tiempo ()

mientras que True

    ahora = tiempo.tiempo ()

    mensaje = '% s,% s,% s'% (comenzar, ahora, (ahora - comenzar))

    si ahora - comenzar>

        imprimir (mensaje + '¡PASADO EL TIEMPO DE ESPERA!')

    más:

        imprimir (mensaje + 'Aún no ...')

    time.sleep (0.2)

Este programa tiene un bucle infinito, así que para finalizarlo, presione Ctrl-C. La salida de este programa se verá así:

1322106392.2, 1322106392.2, 0.0 Not yet...

1322106392.2, 1322106392.42, 0.219000101089 Not yet...

1322106392.2, 1322106392.65, 0.449000120163 Not yet...

1322106392.2, 1322106392.88, 0.680999994278 Not yet...

1322106392.2, 1322106393.11, 0.910000085831 Not yet...

1322106392.2, 1322106393.34, 1.1400001049 Not yet...

1322106392.2, 1322106393.57, 1.3710000515 Not yet...

1322106392.2, 1322106393.83, 1.6360001564 Not yet...

1322106392.2, 1322106394.05, 1.85199999809 Not yet...

1322106392.2, 1322106394.28, 2.08000016212 Not yet...

1322106392.2, 1322106394.51, 2.30900001526 Not yet...

1322106392.2, 1322106394.74, 2.54100012779 Not yet...

1322106392.2, 1322106394.97, 2.76999998093 Not yet...

1322106392.2, 1322106395.2, 2.99800014496 Not yet...

1322106392.2, 1322106395.42, 3.22699999809 Not yet...

1322106392.2, 1322106395.65, 3.45600008965 Not yet...

1322106392.2, 1322106395.89, 3.69200015068 Not yet...

1322106392.2, 1322106396.12, 3.92100000381 Not yet...

1322106392.2, 1322106396.35, 4.14899992943 PASSED WAIT TIME!

1322106392.2, 1322106396.58, 4.3789999485 PASSED WAIT TIME!

1322106392.2, 1322106396.81, 4.60700011253 PASSED WAIT TIME!

1322106392.2, 1322106397.04, 4.83700013161 PASSED WAIT TIME!

1322106392.2, 1322106397.26, 5.06500005722 PASSED WAIT TIME!

Traceback (most recent call last):

  File "C:\timetest.py", line 13, in <module>

    time.sleep(0.2)

KeyboardInterrupt

El primer número en cada línea de salida es el valor de retorno de time.time () cuando el programa se inició por primera vez (y este valor nunca cambia). El segundo número es el último valor de retorno de time.time () (este valor se actualiza constantemente en cada iteración del bucle). Y el tercer número es la hora actual menos la hora de inicio. Este tercer número es el número de segundos que han transcurrido desde que se ejecutó la línea de código begin = time.time () .

Si este número es mayor que 4, el código comenzará a imprimir "¡PASADO EL TIEMPO DE ESPERA!" En lugar de "Todavía no ...". Así es como nuestro programa de juego puede saber si ha transcurrido una cierta cantidad de tiempo desde que se ejecutó una línea de código.

En nuestro programa Tetromino, la expresión time.time () - lastMoveSidewaysTime se evaluará a la cantidad de segundos transcurridos desde la última vez que se configuró lastMoveSidewaysTime a la hora actual. Si este valor es mayor que el valor en MOVESIDEWAYSFREQ , sabemos que es hora de que el código mueva la pieza que cae sobre un espacio más.

¡No olvides actualizar lastMoveSidewaysTime a la hora actual otra vez! Esto es lo que hacemos en la línea 270.

272.         if movingDown and time.time() - lastMoveDownTime > MOVEDOWNFREQ and isValidPosition(board, fallingPiece, adjY=1):

273.             fallingPiece['y'] += 1

274.             lastMoveDownTime = time.time()

Las líneas 272 a 274 hacen casi lo mismo que las líneas 265 a 270, excepto para mover la pieza que cae hacia abajo. Esto tiene una variable de movimiento separada ( movingDown ) y una variable de "última vez" ( lastMoveDownTime ), así como una variable diferente de "frecuencia de movimiento" ( MOVEDOWNFREQ ).

**Dejando caer la pieza “naturalmente”**

276.         # let the piece fall if it is time to fall

277.         if time.time() - lastFallTime > fallFreq:

278.             # see if the piece has landed

279.             if not isValidPosition(board, fallingPiece, adjY=1):

280.                 # falling piece has landed, set it on the board

281.                 addToBoard(board, fallingPiece)

282.                 score += removeCompleteLines(board)

283.                 level, fallFreq = calculateLevelAndFallFreq(score)

284.                 fallingPiece = None

285.             else:

286.                 # piece did not land, just move the block down

287.                 fallingPiece['y'] += 1

288.                 lastFallTime = time.time()

La variable lastFallTime hace un seguimiento de la velocidad a la que la pieza se mueve naturalmente hacia abajo (es decir, la caída) . Si ha transcurrido suficiente tiempo desde la última vez que la pieza que cayó cayó un espacio, las líneas 279 a 288 manejarán soltar la pieza un espacio.

Si la condición en la línea 279 es Verdadera , entonces la pieza ha aterrizado. La llamada a addToBoard () hará que la pieza forme parte de la estructura de datos de la placa (de modo que las futuras piezas puedan aterrizar en ella), y la llamada removeCompleteLines () manejará el borrado de cualquier línea completa en la placa y tirando de las casillas hacia abajo. La función removeCompleteLines () también devuelve un valor entero de cuántas líneas se eliminaron, por lo que agregamos este número a la puntuación.

Debido a que la puntuación puede haber cambiado, llamamos a la función CalculateLevelAndFallFreq () para actualizar el nivel actual y la frecuencia con la que caen las piezas. Y, por último, establecemos la variable fallingPiece en Ninguno para indicar que la siguiente pieza debería convertirse en la nueva pieza descendente, y se debe generar una nueva pieza aleatoria para la nueva pieza siguiente. (Eso se hace en las líneas 195 a 199 al comienzo del ciclo de juego).

Si la pieza no ha aterrizado, simplemente fijamos su posición Y hacia abajo un espacio (en la línea 287) y restablecemos lastFallTime a la hora actual (en la línea 288).

**Dibujando todo en la pantalla**

290.         # drawing everything on the screen

291.         DISPLAYSURF.fill(BGCOLOR)

292.         drawBoard(board)

293.         drawStatus(score, level)

294.         drawNextPiece(nextPiece)

295.         if fallingPiece != None:

296.             drawPiece(fallingPiece)

297.

298.         pygame.display.update()

299.         FPSCLOCK.tick(FPS)

Ahora que el bucle del juego ha manejado todos los eventos y actualizado el estado del juego, el bucle del juego solo necesita dibujar el estado del juego en la pantalla. La mayoría de los dibujos son manejados por otras funciones, por lo que el código de bucle del juego solo necesita llamar a esas funciones. Luego, la llamada a pygame.display.update () hace que la pantalla de Surface aparezca en la pantalla real de la computadora, y la llamada al método tick () agrega una ligera pausa para que el juego no se ejecute demasiado rápido.

**makeTextObjs () , una función de acceso directo para hacer texto**

302. def makeTextObjs(text, font, color):

303.     surf = font.render(text, True, color)

304.     return surf, surf.get\_rect()

La función makeTextObjs () solo nos proporciona un acceso directo. Dado el texto, el objeto Fuente y el objeto Color, llama a render () para nosotros y devuelve el objeto Superficie y Rect para este texto. Esto solo nos evita escribir el código para crear el objeto Surface y Rect cada vez que lo necesitemos.

**El Mismo de terminar () Función**

307. def terminate():

308.     pygame.quit()

309.     sys.exit()

La función terminate () funciona igual que en los programas de juegos anteriores.

**A la espera de un evento para la prensa tecla con el checkForKeyPress () Función**

312. def checkForKeyPress():

313.     # Go through event queue looking for a KEYUP event.

314.     # Grab KEYDOWN events to remove them from the event queue.

315.     checkForQuit()

316.

317.     for event in pygame.event.get([KEYDOWN, KEYUP]):

318.         if event.type == KEYDOWN:

319.             continue

320.         return event.key

321.     return None

La función checkForKeyPress () funciona casi igual que en el juego Wormy. Primero, llama a checkForQuit () para manejar cualquier evento QUIT (o eventos KEYUP específicamente para la tecla Esc) y finaliza el programa si hay alguno. Luego saca todos los eventos KEYUP y KEYDOWN de la cola de eventos. Ignora los eventos de KEYDOWN ( KEYDOWN se especificó en pygame.event.get () solo para eliminar esos eventos de la cola de eventos).

Si no hubo eventos KEYUP en la cola de eventos, la función devuelve Ninguno .

**showTextScreen () , una función de pantalla de texto genérico**

324. def showTextScreen(text):

325.     # This function displays large text in the

326.     # center of the screen until a key is pressed.

327.     # Draw the text drop shadow

328.     titleSurf, titleRect = makeTextObjs(text, BIGFONT, TEXTSHADOWCOLOR)

329.     titleRect.center = (int(WINDOWWIDTH / 2), int(WINDOWHEIGHT / 2))

330.     DISPLAYSURF.blit(titleSurf, titleRect)

331.

332.     # Draw the text

333.     titleSurf, titleRect = makeTextObjs(text, BIGFONT, TEXTCOLOR)

334.     titleRect.center = (int(WINDOWWIDTH / 2) - 3, int(WINDOWHEIGHT / 2) - 3)

335.     DISPLAYSURF.blit(titleSurf, titleRect)

336.

337.     # Draw the additional "Press a key to play." text.

338.     pressKeySurf, pressKeyRect = makeTextObjs('Press a key to play.', BASICFONT, TEXTCOLOR)

339.     pressKeyRect.center = (int(WINDOWWIDTH / 2), int(WINDOWHEIGHT / 2) + 100)

340.     DISPLAYSURF.blit(pressKeySurf, pressKeyRect)

En lugar de funciones separadas para la pantalla de inicio y el juego sobre las pantallas, crearemos una función genérica llamada showTextScreen () . La función showTextScreen () dibujará cualquier texto que pasemos para el parámetro de texto. Además, se mostrará además el texto "Presionar una tecla para jugar".

Observe que las líneas 328 a 330 dibujan el texto en un color de sombra más oscuro primero, y luego las líneas 333 a 335 dibujan el mismo texto nuevamente, excepto el desplazamiento de 3 píxeles a la izquierda y 3 píxeles hacia arriba. Esto crea un efecto de "sombra paralela" que hace que el texto se vea un poco más bonito. Puede comparar la diferencia comentando las líneas 328 a 330 para ver el texto sin una sombra paralela.

Se usará showTextScreen () para la pantalla de inicio, el juego sobre la pantalla y también para una pantalla de pausa. (La pantalla de pausa se explica más adelante en este capítulo).

342.     while checkForKeyPress() == None:

343.         pygame.display.update()

344.         FPSCLOCK.tick()

Queremos que el texto permanezca en la pantalla hasta que el usuario presione una tecla. Este pequeño bucle llamará constantemente a pygame.display.update () y FPSCLOCK.tick () hasta que checkForKeyPress () devuelva un valor distinto de Ninguno . Esto sucede cuando el usuario presiona una tecla.

**El checkForQuit () Función**

347. def checkForQuit():

348.     for event in pygame.event.get(QUIT): # get all the QUIT events

349.         terminate() # terminate if any QUIT events are present

350.     for event in pygame.event.get(KEYUP): # get all the KEYUP events

351.         if event.key == K\_ESCAPE:

352.             terminate() # terminate if the KEYUP event was for the Esc key

353.         pygame.event.post(event) # put the other KEYUP event objects back

Se puede llamar a la función checkForQuit () para manejar cualquier evento que haga que el programa finalice. Esto sucede si hay eventos QUIT en la cola de eventos (esto es manejado por las líneas 348 y 349), o si hay un evento KEYUP de la tecla Esc. El jugador debe poder presionar la tecla Esc en cualquier momento para salir del programa.

Debido a que la llamada pygame.event.get () en la línea 350 saca todos los eventos KEYUP (incluidos los eventos para las teclas que no sean la tecla Esc), si el evento no es para la tecla Esc, queremos volver a colocarlo en la cola de eventos llamando a la función pygame.event.post () .

**El calculateLevelAndFallFreq () Función**

356. def calculateLevelAndFallFreq(score):

357.     # Based on the score, return the level the player is on and

358.     # how many seconds pass until a falling piece falls one space.

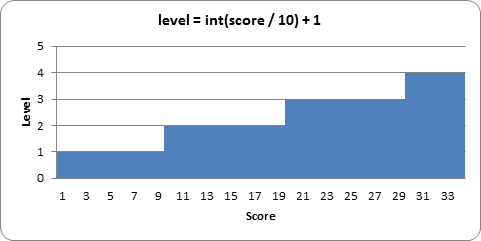
359.     level = int(score / 10) + 1

360.     fallFreq = 0.27 - (level \* 0.02)

361.     return level, fallFreq

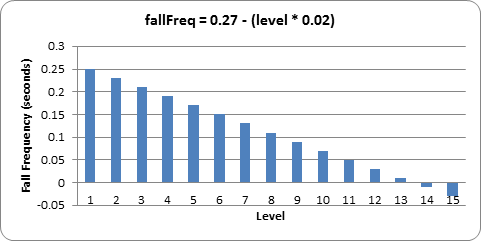
Cada vez que el jugador completa una línea, su puntuación aumentará en un punto. Cada diez puntos, el juego sube un nivel y las piezas comienzan a caer más rápido. Tanto el nivel como la frecuencia de caída se pueden calcular a partir de la puntuación que se pasa a esta función.

Para calcular el nivel, usamos la función int () para redondear el puntaje dividido por 10 . Entonces, si la puntuación de cualquier número entre 0 y 9 , la llamada int () lo redondeará a 0 . La parte + 1 del código está ahí porque queremos que el primer nivel sea el nivel 1, no el nivel 0. Cuando el puntaje llegue a 10 , int (10/10) se evaluará a 1, y el + 1 hará el nivel 2. Aquí hay una gráfica que muestra los valores de nivel para las puntuaciones 1 a 34:



Para calcular la frecuencia de caída, comenzamos con un tiempo base de 0.27 (lo que significa que la pieza caerá naturalmente una vez cada 0.27 segundos). Luego, multiplicamos el nivel por 0.02 y lo restamos del tiempo base de 0.27 . Entonces, en el nivel 1, restamos 0.02 \* 1 (es decir, 0.02 ) de 0.27 para obtener 0.25 . En el nivel 2, restamos 0.02 \* 2 (es decir, 0.04 ) para obtener 0.23 . Puede pensar en el nivel \* 0.02 parte de la ecuación como "para cada nivel, la pieza caerá 0.02 segundos más rápido que el nivel anterior".

También podemos hacer un gráfico que muestre qué tan rápido caerán las piezas en cada nivel del juego:

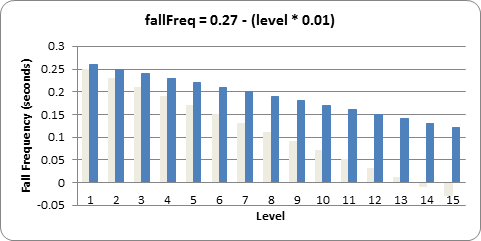


Puedes ver que en el nivel 14, la frecuencia de caída será menor que 0 . Esto no causará ningún error en nuestro código, porque la línea 277 solo comprueba que el tiempo transcurrido desde la última caída de la pieza que cae es un espacio mayor que la frecuencia de caída calculada. Entonces, si la frecuencia de caída es negativa, entonces la condición en la línea 277 siempre será Verdadera y la pieza caerá en cada iteración del bucle del juego. Desde el nivel 14 y más allá, la pieza no puede caer más rápido.

Si el FPS se establece en 25 , esto significa que al alcanzar el nivel 14, la pieza que cae caerá 25 espacios por segundo. ¡Teniendo en cuenta que el tablero solo tiene 20 espacios de altura, eso significa que el jugador tendrá menos de un segundo para colocar cada pieza!

Si quieres que las piezas comiencen (si puedes ver lo que quiero decir) que caen más rápido a un ritmo más lento, puedes cambiar la ecuación que usa la calculaLevelAndFallFreq () . Por ejemplo, digamos que la línea 360 fue esta:

360.     fallFreq = 0.27 - (level \* 0.01)En el caso anterior, las piezas solo caerían 0.01 segundos más rápido en cada nivel en lugar de 0.02 segundos más rápido. El gráfico se vería así (la línea original también está en el gráfico en gris claro):



Como puede ver, con esta nueva ecuación, el nivel 14 solo sería tan difícil como el nivel original 7. Puede cambiar el juego para que sea tan difícil o fácil como lo desee cambiando las ecuaciones en CalculateLevelAndFallFreq () .

**La generación de piezas con el getNewPiece () Función**

363. def getNewPiece():

364.     # return a random new piece in a random rotation and color

365.     shape = random.choice(list(SHAPES.keys()))

366.     newPiece = {'shape': shape,

367.                 'rotation': random.randint(0, len(SHAPES[shape]) - 1),

368.                 'x': int(BOARDWIDTH / 2) - int(TEMPLATEWIDTH / 2),

369.                 'y': -2, # start it above the board (i.e. less than 0)

370.                 'color': random.randint(0, len(COLORS)-1)}

371.     return newPiece

La función getNewPiece () genera una pieza aleatoria que se coloca en la parte superior del tablero. Primero, para elegir aleatoriamente la forma de la pieza, creamos una lista de todas las formas posibles mediante la lista de llamadas (SHAPES.keys ()) en la línea 365. El método de diccionario de keys () devuelve un valor del tipo de datos "dict\_keys" , que debe convertirse a un valor de lista con la función list () antes de pasar a random.choice () . Esto se debe a que la función random.choice () solo acepta valores de lista para su parámetro. La función random.choice () luego retorna aleatoriamente el valor de un elemento de la lista.

Las estructuras de datos de pieza son simplemente un valor de diccionario con las claves 'forma' , 'rotación' , 'x' , 'y' y 'color' .

El valor de la tecla 'rotación' es un número entero aleatorio entre 0 y uno menos que las rotaciones posibles para esa forma. El número de rotaciones para una forma se puede encontrar en la expresión len (SHAPES [forma]) .

Tenga en cuenta que no almacenamos la lista de valores de cadena (como los que se almacenan en las constantes como S\_SHAPE\_TEMPLATE ) en la estructura de datos de cada pieza para representar los cuadros de cada pieza. En su lugar, simplemente almacenamos un índice para la forma y la rotación que se refieren a la constante PIEZAS .

El valor de la tecla 'x' siempre se establece en el centro del tablero (también se tiene en cuenta el ancho de las piezas en sí, que se encuentra en nuestra constante TEMPLATEWIDTH ). El valor de la tecla 'y' siempre se establece en -2 para colocarlo ligeramente por encima del tablero. (La fila superior del tablero es la fila 0).

Dado que la constante de COLORES es una tupla de los diferentes colores, la selección de un número aleatorio de 0 a la longitud de COLORES (restando uno) nos dará un valor de índice aleatorio para el color de la pieza.

Una vez que se configuran todos los valores en el diccionario newPiece , la función getNewPiece () devuelve newPiece .

**Agregando piezas a la estructura de datos del tablero**

374. def addToBoard(board, piece):

375.     # fill in the board based on piece's location, shape, and rotation

376.     for x in range(TEMPLATEWIDTH):

377.         for y in range(TEMPLATEHEIGHT):

378.             if SHAPES[piece['shape']][piece['rotation']][y][x] != BLANK:

379.                 board[x + piece['x']][y + piece['y']] = piece['color']

La estructura de datos del tablero es una representación de datos para el espacio rectangular donde se rastrean las piezas que han aterrizado previamente. La pieza que cae actualmente no está marcada en la estructura de datos del tablero. Lo que hace la función addToBoard () es tomar una estructura de datos de pieza y agrega sus cuadros a la estructura de datos de la placa. Esto sucede después de que una pieza ha aterrizado.

Los anidados para los bucles en las líneas 376 y 377 recorren cada espacio en la estructura de datos de la pieza, y si encuentra una caja en el espacio (línea 378), la agrega al tablero (línea 379).

**Creando una nueva estructura de datos de tablero**

382. def getBlankBoard():

383.     # create and return a new blank board data structure

384.     board = []

385.     for i in range(BOARDWIDTH):

386.         board.append([BLANK] \* BOARDHEIGHT)

387.     return board

La estructura de datos utilizada para el tablero es bastante simple: es una lista de listas de valores. Si el valor es el mismo que el de BLANK , entonces es un espacio vacío. Si el valor es un entero, entonces representa un cuadro que es el color que el entero indexa en la lista de constantes de COLORES . Es decir, 0 es azul, 1 es verde, 2 es rojo y 3 es amarillo.

Para crear una placa en blanco, la replicación de listas se utiliza para crear las listas de valores en BLANCO que representan una columna. Esto se hace en la línea 386. Una de estas listas se crea para cada una de las columnas en el tablero (esto es lo que hace el bucle for en la línea 385).

**El isOnBoard () y isValidPosition () Funciones**

390. def isOnBoard(x, y):

391.     return x >= 0 and x < BOARDWIDTH and y < BOARDHEIGHT

El isOnBoard () es una función simple que comprueba que las coordenadas XY que se pasan representan valores válidos que existen en el tablero. Siempre que ambas coordenadas XY no sean menos 0ni mayores o iguales que las constantes BOARDWIDTH y BOARDHEIGHT , entonces la función devuelve True .

394. def isValidPosition(board, piece, adjX=0, adjY=0):

395.     # Return True if the piece is within the board and not colliding

396.     for x in range(TEMPLATEWIDTH):

397.         for y in range(TEMPLATEHEIGHT):

398.             isAboveBoard = y + piece['y'] + adjY < 0

399.             if isAboveBoard or SHAPES[piece['shape']][piece['rotation']][y][x] == BLANK:

400.                 continue

A la función isValidPosition () se le asigna una estructura de datos de tablero y una estructura de datos de pieza, y devuelve True si todas las casillas de la pieza están en la pizarra y no se superponen a ninguna de las casillas de la pizarra. Esto se hace tomando las coordenadas XY de la pieza (que en realidad es la coordenada del cuadro superior derecho de las casillas 5x5 para la pieza) y agregando la coordenada dentro de la estructura de datos de la pieza. Aquí hay un par de fotos para ayudar a ilustrar esto:

|  |  |
| --- | --- |
| http://inventwithpython.com/pygame/chapter7_files/image009.jpg | http://inventwithpython.com/pygame/chapter7_files/image010.jpg |
| El tablero con una pieza que cae en una posición válida. | El tablero con la pieza que cae en una posición inválida. |

En el tablero izquierdo, las coordenadas XY de la pieza descendente (es decir, la esquina superior izquierda de la pieza descendente) están (2, 3) en el tablero. Pero las cajas dentro del sistema de coordenadas de la pieza que cae tienen sus propias coordenadas. Para encontrar las coordenadas de "tabla" de estas piezas, solo tenemos que agregar las coordenadas de "tabla" del cuadro superior izquierdo de la pieza descendente y las coordenadas de "pieza" de las cajas.

En el tablero izquierdo, las casillas de la pieza que cae están en las siguientes coordenadas de la "pieza":

            (2, 2) (3, 2) (1, 3) (2, 3)

Cuando agregamos la coordenada (2, 3) (las coordenadas de la pieza en el tablero) a estas coordenadas, se ve así:

            (2 + 2, 2 + 3) (3 + 2, 2 + 3) (1 + 2, 3 + 3) (2 + 2, 3 + 3)

Después de agregar las coordenadas (2, 3), las casillas se encuentran en las siguientes coordenadas de la "tabla":

            (4, 5) (5, 5) (3, 6) (4, 6)

Y ahora que podemos averiguar dónde están las cajas de la pieza que cae como coordenadas del tablero, podemos ver si se superponen con las cajas que ya están en el tablero. Los anidados para bucles en las líneas 396 y 397 pasan por cada una de las coordenadas posibles en la pieza descendente.

Queremos verificar si una caja de la pieza que cae está fuera de la tabla o si se superpone a una caja en la tabla. (Aunque una excepción es si el cuadro está sobre el tablero, que es donde podría estar cuando la pieza que cae simplemente comienza a caer). La línea 398 crea una variable llamada isAboveBoard que se establece en Verdadero si el cuadro en la parte que cae en las coordenadas señalado para ser x e y está por encima del tablero. De lo contrario, se establece en Falso .

La declaración if en la línea 399 verifica si el espacio en la pieza está sobre el tablero o si está en blanco. Si cualquiera de estos es verdadero , entonces el código ejecuta una instrucción de continuación y pasa a la siguiente iteración. (Tenga en cuenta que el final de la línea 399 tiene [y] [x] en lugar de [x] [y] . Esto se debe a que las coordenadas en la estructura de datos de PIEZAS están invertidas. Consulte la sección anterior, “Configuración de las plantillas de piezas”) .

401.             if not isOnBoard(x + piece['x'] + adjX, y + piece['y'] + adjY):

402.                 return False

403.             if board[x + piece['x'] + adjX][y + piece['y'] + adjY] != BLANK:

404.                 return False

405.     return True

La declaración if en la línea 401 comprueba que la caja de la pieza no se encuentra en el tablero. La declaración if en la línea 403 verifica que el espacio de la pizarra en el que se encuentra la caja de la pieza no esté en blanco. Si cualquiera de estas condiciones es verdadera , entonces la función isValidPosition () devolverá falsa . Observe que estas declaraciones if también ajustan las coordenadas para los parámetros adjX y adjY que se pasaron a la función.

Si el código pasa por el anidado de bucle y no ha encontrado una razón para volver Falso , entonces la posición de la pieza debe ser válido y por lo que la función devuelve Verdadero en la línea 405.

**Comprobación y eliminación de líneas completas**

407. def isCompleteLine(board, y):

408.     # Return True if the line filled with boxes with no gaps.

409.     for x in range(BOARDWIDTH):

410.         if board[x][y] == BLANK:

411.             return False

412.     return True

El isCompleteLine hace una simple comprobación en la fila especificada por el y parámetro. Una fila en el tablero se considera "completa" cuando cada espacio se llena con un cuadro. El bucle for en la línea 409 atraviesa cada espacio de la fila. Si un espacio está en blanco (lo que se debe a que tiene el mismo valor que la constante BLANK ), la función devuelve False .

415. def removeCompleteLines(board):

416.     # Remove any completed lines on the board, move everything above them down, and return the number of complete lines.

417.     numLinesRemoved = 0

418.     y = BOARDHEIGHT - 1 # start y at the bottom of the board

419.     while y >= 0:

La función removeCompleteLines () encontrará líneas completas en la estructura de datos de la placa pasada, eliminará las líneas y luego desplazará todas las casillas de la placa sobre esa línea hacia abajo una fila. La función devolverá el número de líneas que se eliminaron (lo que se realiza un seguimiento mediante la variable numLinesRemoved ) para que se pueda agregar a la puntuación.

La forma en que funciona esta función es ejecutándose en un bucle que comienza en la línea 419 con la variable y comenzando en la fila más baja (que es BOARDHEIGHT - 1 ). Siempre que la fila especificada por y no esté completa, y se reducirá a la siguiente fila más alta. El bucle finalmente se detiene una vez que y llega a -1 .

420.         if isCompleteLine(board, y):

421.             # Remove the line and pull boxes down by one line.

422.             for pullDownY in range(y, 0, -1):

423.                 for x in range(BOARDWIDTH):

424.                     board[x][pullDownY] = board[x][pullDownY-1]

425.             # Set very top line to blank.

426.             for x in range(BOARDWIDTH):

427.                 board[x][0] = BLANK

428.             numLinesRemoved += 1

429.             # Note on the next iteration of the loop, y is the same.

430.             # This is so that if the line that was pulled down is also

431.             # complete, it will be removed.

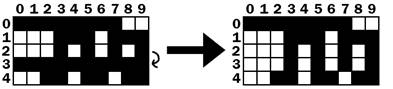
432.         else:

433.             y -= 1 # move on to check next row up

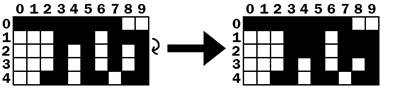
434.     return numLinesRemoved

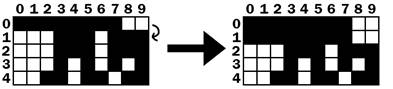
El isCompleteLine () función devolverá Verdadero si la línea que Y se refiere es completa. En ese caso, el programa necesita copiar los valores de cada fila sobre la línea eliminada a la siguiente línea más baja. Esto es lo que hace el bucle for en la línea 422 (razón por la cual su llamada a la función range () comienza en y , en lugar de 0. También tenga en cuenta que usa la forma de tres argumentos de range () , de modo que la lista retorna comienza en y , termina en 0 , y después de cada iteración "aumenta" en -1 .)

Veamos el siguiente ejemplo. Para ahorrar espacio, solo se muestran las cinco filas superiores del tablero. La fila 3 es una línea completa, lo que significa que todas las filas que se encuentran sobre ella (filas 2, 1 y 0) deben "bajarse". Primero, la fila 2 se copia a la fila 3. La tabla a la derecha muestra cómo se verá la tabla después de que se haga esto:

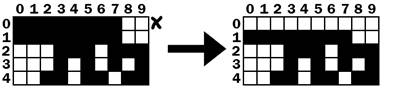


Este "tirón hacia abajo" es realmente simplemente copiando los valores de la fila superior a la fila debajo de ella en la línea 424. Después de que la fila 2 se copia en la fila 3, la fila 1 se copia en la fila 2 seguida de la fila 0 se copia en la fila 1:





La fila 0 (la fila en la parte superior) no tiene una fila encima para copiar los valores. Pero la fila 0 no necesita una fila copiada, solo necesita que todos los espacios estén en BLANCO . Esto es lo que hacen las líneas 426 y 427. Después de eso, el tablero habrá cambiado del tablero que se muestra abajo a la izquierda al tablero que se muestra abajo a la derecha:



Después se retira la línea completa, la ejecución llega al final de la mientras bucle que se inició en la línea 419, por lo que la ejecución salta de nuevo al principio del bucle. Tenga en cuenta que en ningún momento cuando se eliminó la línea y las filas se bajaron, la variable y cambió en absoluto. Entonces, en la siguiente iteración, la variable y apunta a la misma fila que antes.

Esto es necesario porque si hubiera dos líneas completas, entonces la segunda línea completa habría sido tirada hacia abajo y también tendría que ser eliminada. El código eliminará esta línea completa y luego pasará a la siguiente iteración. Es solamente cuando no hay una línea completa que el y variable se decrementa en la línea 433. Una vez que el y la variable ha disminuido hasta llegar a 0 , la ejecución va a salir del tiempo de bucle.

**Convertir de coordenadas del tablero a coordenadas de píxeles**

437. def convertToPixelCoords(boxx, boxy):

438.     # Convert the given xy coordinates of the board to xy

439.     # coordinates of the location on the screen.

440.     return (XMARGIN + (boxx \* BOXSIZE)), (TOPMARGIN + (boxy \* BOXSIZE))

Esta función auxiliar convierte las coordenadas de la caja de la placa en coordenadas de píxeles. Esta función funciona de la misma manera que las otras funciones de "convertir coordenadas" usadas en los programas de juegos anteriores.

**Dibujar una caja en la pizarra o en otra parte en la pantalla**

443. def drawBox(boxx, boxy, color, pixelx=None, pixely=None):

444.     # draw a single box (each tetromino piece has four boxes)

445.     # at xy coordinates on the board. Or, if pixelx & pixely

446.     # are specified, draw to the pixel coordinates stored in

447.     # pixelx & pixely (this is used for the "Next" piece).

448.     if color == BLANK:

449.         return

450.     if pixelx == None and pixely == None:

451.         pixelx, pixely = convertToPixelCoords(boxx, boxy)

452.     pygame.draw.rect(DISPLAYSURF, COLORS[color], (pixelx + 1, pixely + 1, BOXSIZE - 1, BOXSIZE - 1))

453.     pygame.draw.rect(DISPLAYSURF, LIGHTCOLORS[color], (pixelx + 1, pixely + 1, BOXSIZE - 4, BOXSIZE - 4))

La función drawBox () dibuja un solo cuadro en la pantalla. La función puede recibir boxx y cuadradas parámetros para el tablero de coordenadas donde la caja debe ser dibujada. Sin embargo, si el pixelX y pixely se especifican parámetros, entonces estas coordenadas en píxeles anularán la boxx y cuadradas parámetros. Los parámetros pixelx y pixely se utilizan para dibujar las cajas de la pieza "Siguiente", que no está en el tablero.

Si los parámetros pixelx y pixely no se configuran, entonces se establecerán en Ninguno de forma predeterminada cuando la función comience por primera vez. Luego, la instrucción if en la línea 450 sobrescribirá los valores de Ninguno con los valores de retorno de convertToPixelCoords () . Esta llamada se pone las coordenadas de píxel de la tabla de coordenadas especificadas por boxx y cuadrada .

El código no llenará de color el espacio de toda la caja. Para tener un contorno negro entre las cajas de una pieza, los parámetros izquierdo y superior en la llamada pygame.draw.rect () tienen + 1 agregado y se agrega un - 1 a los parámetros de ancho y alto . Para dibujar el cuadro resaltado, primero se dibuja el cuadro con el color más oscuro en la línea 452. Luego, se dibuja un cuadro más pequeño en la parte superior del cuadro más oscuro en la línea 453.

**Dibujando todo a la pantalla**

456. def drawBoard(board):

457.     # draw the border around the board

458.     pygame.draw.rect(DISPLAYSURF, BORDERCOLOR, (XMARGIN - 3, TOPMARGIN - 7, (BOARDWIDTH \* BOXSIZE) + 8, (BOARDHEIGHT \* BOXSIZE) + 8), 5)

459.

460.     # fill the background of the board

461.     pygame.draw.rect(DISPLAYSURF, BGCOLOR, (XMARGIN, TOPMARGIN, BOXSIZE \* BOARDWIDTH, BOXSIZE \* BOARDHEIGHT))

462.     # draw the individual boxes on the board

463.     for x in range(BOARDWIDTH):

464.         for y in range(BOARDHEIGHT):

465.             drawBox(x, y, board[x][y])

La función drawBoard () es responsable de llamar a las funciones de dibujo para el borde de la pizarra y todas las casillas de la pizarra. Primero se dibuja el borde de la pizarra en DISPLAYSURF , seguido del color de fondo de la pizarra. Luego se realiza una llamada a drawBox () para cada espacio en el tablero. La función drawBox () es lo suficientemente inteligente como para dejar de lado el cuadro si la placa [x] [y] está configurada en BLANK .

**Dibujar la puntuación y el texto de nivel**

468. def drawStatus(score, level):

469.     # draw the score text

470.     scoreSurf = BASICFONT.render('Score: %s' % score, True, TEXTCOLOR)

471.     scoreRect = scoreSurf.get\_rect()

472.     scoreRect.topleft = (WINDOWWIDTH - 150, 20)

473.     DISPLAYSURF.blit(scoreSurf, scoreRect)

474.

475.     # draw the level text

476.     levelSurf = BASICFONT.render('Level: %s' % level, True, TEXTCOLOR)

477.     levelRect = levelSurf.get\_rect()

478.     levelRect.topleft = (WINDOWWIDTH - 150, 50)

479.     DISPLAYSURF.blit(levelSurf, levelRect)

La función drawStatus () es responsable de representar el texto de la información "Puntuación:" y "Nivel:" que aparece en la esquina superior derecha de la esquina de la pantalla.

**Dibujar una pieza en la pizarra o en otro lugar en la pantalla**

482. def drawPiece(piece, pixelx=None, pixely=None):

483.     shapeToDraw = SHAPES[piece['shape']][piece['rotation']]

484.     if pixelx == None and pixely == None:

485.         # if pixelx & pixely hasn't been specified, use the location stored in the piece data structure

486.         pixelx, pixely = convertToPixelCoords(piece['x'], piece['y'])

487.

488.     # draw each of the blocks that make up the piece

489.     for x in range(TEMPLATEWIDTH):

490.         for y in range(TEMPLATEHEIGHT):

491.             if shapeToDraw[y][x] != BLANK:

492.                 drawBox(None, None, piece['color'], pixelx + (x \* BOXSIZE), pixely + (y \* BOXSIZE))

La función drawPiece () dibujará los cuadros de una pieza de acuerdo con la estructura de datos de la pieza que se le pasa. Esta función se utilizará para dibujar la pieza descendente y la pieza "Siguiente". Dado que la estructura de datos de la pieza contendrá toda la información de forma, posición, rotación y color, no se debe pasar nada más que la estructura de datos de la pieza a la función.

Sin embargo, la pieza "Siguiente" no está dibujada en el tablero. En este caso, ignoramos la información de posición almacenada dentro de la estructura de datos de la pieza y en su lugar permitimos que la persona que llama de la función drawPiece () pase los argumentos para los parámetros pixelx y pixely opcionales para especificar dónde exactamente se debe dibujar la pieza en la ventana.

Si no se pasan argumentos pixelx y pixely , entonces las líneas 484 y 486 sobrescribirán esas variables con los valores de retorno de la llamada convertToPixelCoords () .

Los anidados para bucles en las líneas 489 y 490 llamarán a drawBox () para cada caja de la pieza que se debe dibujar.

**Dibujando la siguiente pieza**

495. def drawNextPiece(piece):

496.     # draw the "next" text

497.     nextSurf = BASICFONT.render('Next:', True, TEXTCOLOR)

498.     nextRect = nextSurf.get\_rect()

499.     nextRect.topleft = (WINDOWWIDTH - 120, 80)

500.     DISPLAYSURF.blit(nextSurf, nextRect)

501.     # draw the "next" piece

502.     drawPiece(piece, pixelx=WINDOWWIDTH-120, pixely=100)

503.

504.

505. if \_\_name\_\_ == '\_\_main\_\_':

506.     main()

El drawNextPiece () dibuja la pieza en “Siguiente” en la esquina superior derecha de la pantalla. Esto se hace llamando a la drawPiece () la función y pasando los argumentos para drawPiece () 's pixelX y pixely parámetros.

Esa es la última función. Las líneas 505 y 506 se ejecutan después de que se hayan ejecutado todas las definiciones de funciones, y luego se llama a la función main () para comenzar la parte principal del programa.

**Resumen**

El juego Tetromino (que es un clon del juego más popular, "Tetris") es bastante fácil de explicar a alguien en inglés: "Los bloques caen desde la parte superior de un tablero, y el jugador los mueve y los gira para que se completen. líneas. Las líneas completas desaparecen (dando puntos al jugador) y las líneas sobre ellas se mueven hacia abajo. El juego continúa hasta que los bloques llenan todo el tablero y el jugador pierde ".

Explicarlo en un lenguaje sencillo es una cosa, pero cuando tenemos que decirle a una computadora exactamente qué hacer hay muchos detalles que debemos completar. El juego original de Tetris fue diseñado y programado para una persona, Alex Pajitnov, en la Unión Soviética en 1984. El juego es simple, divertido y adictivo. Es uno de los videojuegos más populares que se haya hecho, y ha vendido 100 millones de copias con muchas personas creando sus propios clones y variaciones.

Y todo fue creado por una persona que sabía cómo programar.

Con la idea correcta y algunos conocimientos de programación, puedes crear juegos increíblemente divertidos. ¡Y con algo de práctica, podrás convertir tus ideas de juegos en programas reales que podrían llegar a ser tan populares como Tetris!

Para una práctica de programación adicional, puede descargar versiones con errores de Tetromino desde <http://invpy.com/buggy/tetromino> e intentar descubrir cómo corregir los errores.

También hay variaciones del juego de Tetromino en el sitio web del libro. "Pentomino" es una versión de este juego con piezas compuestas de cinco cajas. También hay "Tetromino para idiotas", donde todas las piezas están formadas por una sola caja.

