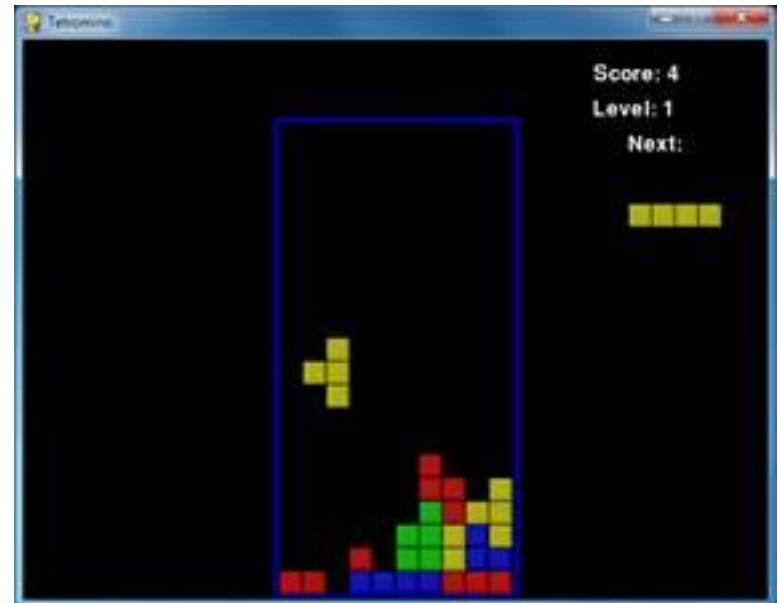


ФАКУЛТЕТ ЗА ИНФОРМАТИЧКИ НАУКИ И КОМПЈУТЕРСКО ИНЖЕНЕРСТВО

Tetromino



How to Play Tetromino

- Tetromino is a Tetris clone.
 - Differently shaped blocks fall from the top of the screen
 - The player must guide them down to form complete rows that have no gaps in them
 - When a complete row is formed, the row disappears and each row above moves down one row.
 - The player tries to keep forming complete lines until the screen fills up and a new falling block cannot fit on the screen.

Some Tetromino Nomenclature

■ Board

- The board is made up of 10 x 20 spaces that the blocks fall and stack up in.

■ Box

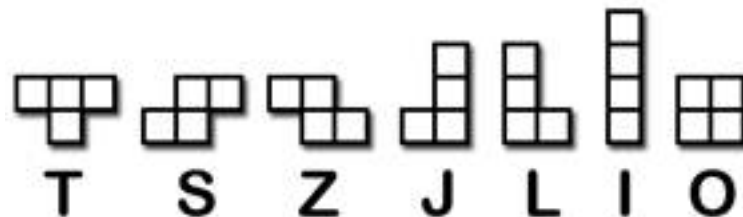
- A box is a single filled-in square space on the board.

■ Piece

- The things that fall from the top of the board that the player can rotate and position.
- Each piece has a shape and is made up of 4 boxes.

■ Shape

- The shapes are the different types of pieces in the game.
- The names of the shapes are T, S, Z, J, L, I, and O.



Some Tetromino Nomenclature

■ Template

- A list of shape data structures that represents all possible rotations of a shape are stored in variables with names like S_SHAPE_TEMPLATE or J_SHAPE_TEMPLATE.

■ Landed

- When a piece has either reached the bottom of the board or is touching a box on the board, we say that the piece has landed.
- At that point, the next piece should start falling.

Source Code to Tetromino

- This source code can be downloaded from
 - <http://invpy.com/tetromino.py>.
- The background music files can be downloaded from here:
 - <http://invpy.com/tetrisc.mid>
 - <http://invpy.com/tetrisb.mid>
- The files should be placed in the same folder as the tetromino.py file
- **Note: You need audio device to run the game!!!**

The Usual Setup Code

2

```
1. # Tetromino (a Tetris clone)
2. # By Al Sweigart al@inventwithpython.com
3. # http://inventwithpython.com/pygame
4. # Creative Commons BY-NC-SA 3.0 US
6. import random, time, pygame, sys
7. from pygame.locals import *
9. FPS = 25
10. WINDOWWIDTH = 640
11. WINDOWHEIGHT = 480
12. BOXSIZE = 20
13. BOARDWIDTH = 10
14. BOARDHEIGHT = 20
15. BLANK = '.'
```

- These are the constants used by our Tetromino game.
- Each box is a square that is 20 pixels wide and high.
- The board itself is 10 boxes wide and 20 boxes tall.
- The BLANK constant will be used as a value to represent blank spaces in the board's data structure.



Setting up Timing Constants for Holding Down Keys

17. `MOVESIDEWAYSFREQ = 0.15`

18. `MOVEDOWNFREQ = 0.1`

- Every time the player pushes the left or right arrow key down, the falling piece should move one box over to the left or right, respectively.
- The player can also hold down the left or right arrow key to keep moving the falling piece.
- The `MOVESIDEWAYSFREQ` constant will set it so that every 0.15 seconds that passes with the left or right arrow key held down, the piece will move another space over.
- The `MOVEDOWNFREQ` constant is the same thing except it tells how frequently the piece drops by one box while the player has the down arrow key held down.

More Setup Code

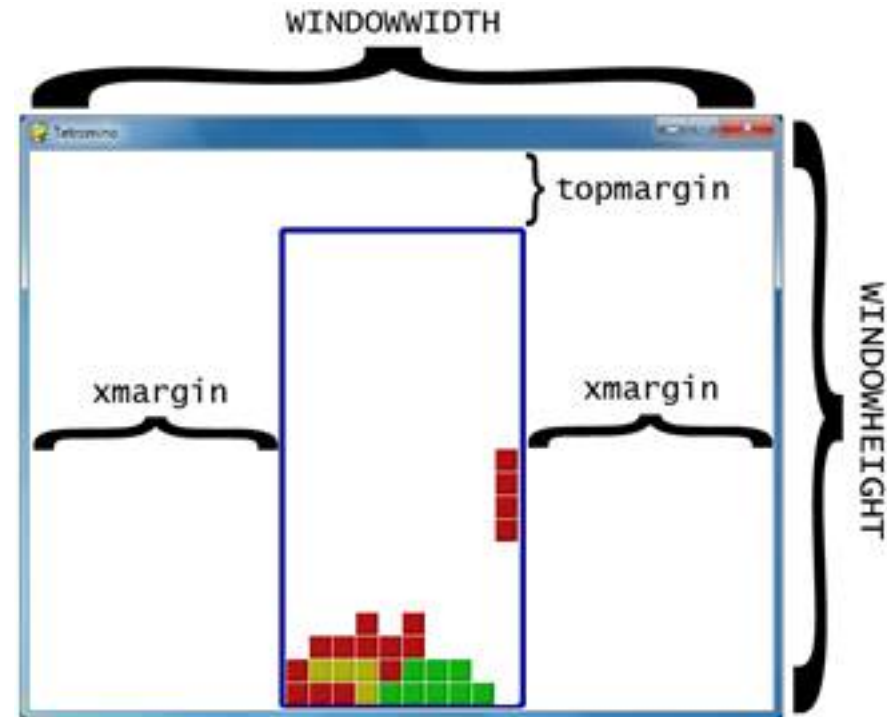
- 20. $\text{XMARGIN} = \text{int}((\text{WINDOWWIDTH} - \text{BOARDWIDTH} * \text{BOXSIZE}) / 2)$
- 21. $\text{TOPMARGIN} = \text{WINDOWHEIGHT} - (\text{BOARDHEIGHT} * \text{BOXSIZE}) - 5$

■ The program needs to calculate how many pixels are to the left and right side of the board.

- WINDOWWIDTH is the width of the entire window measured in pixels.
- The board is BOARDWIDTH boxes wide and each box is BOXSIZE pixels wide.
- If we subtract $(\text{BOARDWIDTH} * \text{BOXSIZE})$ from WINDOWWIDTH, we have the size of the margin to the left and right of the board.
- If we divide this by 2, we have the size of just one margin.
- Since the margins are the same size, we use XMARGIN for either the left-side or right-side margin.

■ The size of the space between the top of the board and the top of the window is calculated in a similar manner.

- The board will be drawn 5 pixels above the bottom of the window, so 5 is subtracted from TOPMARGIN to account for this.



More setup code

```
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)
36. BORDERCOLOR = BLUE
37. BGCOLOR     = BLACK
38. TEXTCOLOR   = WHITE
39. TEXTSHADOWCOLOR = GRAY
```

More setup code

```
40. COLORS = (BLUE, GREEN, RED, YELLOW)
41. LIGHTCOLORS = (LIGHTBLUE, LIGHTGREEN, LIGHTRED, LIGHTYELLOW)
42. assert len(COLORS) == len(LIGHTCOLORS) # each color must have light color
```

- The pieces will come in four colors: blue, green, red, and yellow.
 - When we draw the boxes though, there will be a thin highlight on the box in a lighter color.
 - So we need to create light blue, light green, light red, and light yellow colors as well.
 - Each of these four colors will be stored in tuples named COLORS (for the normal colors) and LIGHTCOLORS (for the lighter colors).

Setting Up the Piece Templates

- The program needs to know how each of the shapes are shaped, including all their possible rotations.
 - So, we will create lists of lists of strings.
 - The inner list of strings will represent a single rotation of a shape, like this:

```
['....',  
 '....',  
 '..OO.',  
 '.OO..',  
 '....']
```



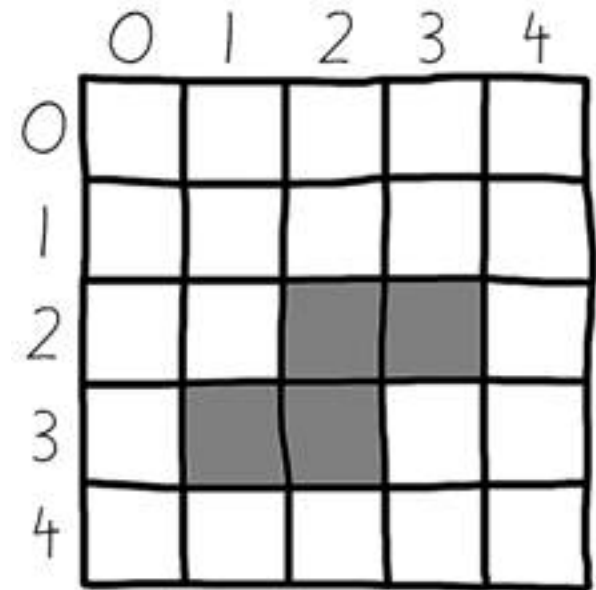
- We will write the rest of our code so that it interprets a list of strings like the one above to represent represent a shape where the periods are empty spaces and the O's are boxes, like as presented on the figure.

Setting Up the Piece Templates

- “template” data structures of the shapes is a list of **these list of strings**, stored in variables such as `S_SHAPE_TEMPLATE`.
 - `len(S_SHAPE_TEMPLATE)` represents how many possible rotations there are for the S shape
 - `S_SHAPE_TEMPLATE[0]` represents the S shape’s first possible rotation.

Setting Up the Piece Templates

- Each possible piece in a tiny 5 x 5 board of empty space, with some of the spaces on the board filled in with boxes.
- The following expressions that use `S_SHAPE_TEMPLATE[0]` are True:
 - `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'`
- If we represented this shape on paper, it would look something like this:



Setting Up the Piece Templates

```
44. TEMPLATEWIDTH = 5
45. TEMPLATEHEIGHT = 5
47. S_SHAPE_TEMPLATE = [['.....',
48.                        '.....',
49.                        '..OO.',
50.                        '.OO..',
51.                        '.....'],
52.                        ['.....',
53.                        '..O..',
54.                        '..OO.',
55.                        '...O.',
56.                        '.....']]
```

Setting Up the Piece Templates

```
58. Z_SHAPE_TEMPLATE = [['.....',  
59.                        '.....',  
60.                        '.OO..',  
61.                        '..OO.',  
62.                        '.....'],  
63. ['.....',  
64.     '..O..',  
65.     '.OO..',  
66.     '.O...',  
67.     '.....']]
```

Setting Up the Piece Templates

```
69. I_SHAPE_TEMPLATE = [['..O..',
70.                     '..O..',
71.                     '..O..',
72.                     '..O..',
73.                     '.....'],
74.                     ['.....',
75.                     '.....',
76.                     'OOOOO.',
77.                     '.....',
78.                     '.....']]
```




Setting Up the Piece Templates

```
80. O_SHAPE_TEMPLATE = [['.....',  
81.                        '.....',  
82.                        '.OO..',  
83.                        '.OO..',  
84.                        '.....']]
```

Setting Up the Piece Templates

```
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.                     '.....']]
```

Setting Up the Piece Templates

```

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.                        '.....']]

```

Setting Up the Piece Templates

```

128. T_SHAPE_TEMPLATE = [['.....',
129.                        '..O..',
130.                        '...O..',
131.                        '.....',
132.                        '.....'],
133.                       ['.....',
134.                        '..O..',
135.                        '..OO..',
136.                        '..O..',
137.                        '.....'],
138.                       ['.....',
139.                        '...O..',
140.                        '...O..',
141.                        '...O..',
142.                        '.....'],
143.                       ['.....',
144.                        '..O..',
145.                        '..OO..',
146.                        '..O..',
147.                        '.....']]

```



!!! Python things:

Splitting a “Line of Code” Across Multiple Lines

- You can see that this list is spread across many lines in the file editor.
- This is perfectly valid Python, because the Python interpreter realizes that until it sees the] closing square bracket, the list isn't finished.
- The indentation doesn't matter because Python knows you won't have different indentation for a new block in the middle of a list.
- This code below works just fine:

```
spam = ['hello', 3.14, 'world', 42, 10, 'fuzz']  
eggs = ['hello', 3.14,  
'world' , 42,  
10, 'fuzz']
```
- Normally, splitting a line of code across multiple lines in the file editor would require putting a \ character at the end of the line.

Setting Up the Piece Templates

3

```
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}
```

- The SHAPES variable will be a dictionary that stores all templates.
- Because each template has all possible rotations of a single shape, this means that the SHAPES variable contains all possible rotations of every possible shape.
- This will be the data structure that contains all shape data in our game.



The main() Function

```
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')
167.     showTextScreen('Tetromino')
```

- The main() function handles creating some more global constants and showing the start screen that appears when the program is run.

The main() Function

4

```
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')
```

- The code for the actual game is all in runGame().
- The main() function simply randomly decides what background music to start playing, and calls runGame() to begin the game.
- When the player loses, runGame() will return to main(), which then stops the background music and displays the game over screen.
- When the player presses a key, the showTextScreen() function that displays the game over screen will return.

The Start of a New Game

```

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)
191.     fallingPiece = getNewPiece()
192.     nextPiece = getNewPiece()

```

the fallingPiece will be set to the currently falling piece that can be rotated by the player.

the nextPiece variable will be set to the piece that shows up in the "Next" part of the screen so that player knows what piece is coming up after setting the falling piece.

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
201.         if not isValidPosition(board, fallingPiece):
202.             return # can't fit a new piece on the board, so game over
204.     checkForQuit()
```

- The fallingPiece variable is set to None after the falling piece has landed.
- The piece in nextPiece should be copied to the fallingPiece variable, and a random new piece should be put into the nextPiece variable.
- The lastFallTime variable is reset to the current time so that the piece will fall in however many seconds is in fallFreq.
- The pieces that getNewPiece() are positioned a little bit above the board, usually with part of the piece already on the board.
- But if this is an invalid position because the board is already filled up there, then we know that the board is full and the player should lose the game.
- When this happens, the runGame() function returns.

The Event Handling Loop

```
205.         for event in pygame.event.get(): # event handling loop
206.             if event.type == KEYUP:
```

- The event handling loop takes care of when the player rotates the falling piece, moves the falling piece, or pauses the game.

Pausing the Game

```

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()
    
```

- If the player has pressed the P key, then the game should pause.
- We hide the board from the player (otherwise the player could cheat by pausing the game and taking time to decide where to move the piece).
- The code blanks out the display Surface with a call to `DISPLAYSURF.fill(BGCOLOR)` and stops the music.
- The `showTextScreen()` function is called to display the “Paused” text and wait for the player to press a key to continue.
- Once the player has pressed a key, `showTextScreen()` will return.
- Since a large amount of time could have passed since the player paused the game, the `lastFallTime`, `lastMoveDownTime`, and `lastMoveSidewaysTime` variables should all be reset to the current time

Using Movement Variables to Handle User Input

```

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
    
```

- Letting up on one of the arrow keys (or the WASD keys) will set the movingLeft, movingRight, or movingDown variables back to False, indicating that the player no longer wants to move the piece in those directions.
- Note that the up arrow and W keys are used for rotating the piece, not moving the piece up.
 - This is why there is no movingUp variable.

Checking if a Slide or Rotation is Valid

```

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()

```

When the left arrow key is pressed down and moving to the left is a valid move for the falling piece, we change the position to one space to the left by subtracting the value of fallingPiece['x'] by 1.

If the movingLeft variable is set to True, the program can know that the left arrow key (or A key) has been pressed and not yet let go.

The lastMoveSidewaysTime variable will be updated to the current time. And if 0.15 seconds (the number stored in MOVESIDEWAYSFREQ) has passed since the time stored in lastMoveSidewaysTime, then it is time for the program to move the falling piece to the left again.

isValidPosition() function

- has optional parameters called adjX and adjY.
- checks the position of the data provided by the piece object that is passed for the second parameter.
- However, sometimes we don't want to check where the piece is currently located, but rather a few spaces over from that position.
- If we pass -1 for the adjX (a short name for “adjusted X”), then it doesn't check the validity of the position in the piece's data structure, but rather if the position of where the piece would be if it was one space to the left.
- Passing 1 for adjX would check one space to the right.
- There is also an adjY optional parameter.
- Passing -1 for adjY checks one space above where the piece is currently positioned, and passing a value like 3 for adjY would check three spaces down from where the piece is.

Checking if a Slide or Rotation is Valid

```
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()
```


Checking if a Slide or Rotation is Valid

```

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']])

```

- The up arrow key (or W key) will rotate the falling piece to its next rotation.
- All code has to do is increment the 'rotation' key's value in the fallingPiece dictionary by 1.
- However, if incrementing the 'rotation' key's value makes it larger than the total number of rotations, then “modding” by the total number of possible rotations for that shape (which is what `len(SHAPES[fallingPiece['shape']])` is) then it will “roll over” to 0.
- If the new rotated position is not valid because it overlaps some boxes already on the board, then we want to switch it back to the original rotation by subtracting 1 from `fallingPiece['rotation']`.
- We can also mod it by `len(SHAPES[fallingPiece['shape']])` so that if the new value is -1, the modding will change it back to the last rotation in the list.

Checking if a Slide or Rotation is Valid

```

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()
    
```

- If the down arrow or S key is pressed down, then the player wants the piece to fall faster than normal.
 - The movingDown variable is set to True and lastMoveDownTime is reset to the current time.
 - These variables will be checked later so that the piece keeps falling at the faster rate as long as the down arrow or S key is held down.

Finding the Bottom

```

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

```

← When the player presses the space key the falling piece will immediately drop down as far as it can go on the board and land.

The program first needs to find out how many spaces the piece can move until it lands.

←

Lines 256 to 258 will set all moving variables to False. This is done because this code will move the piece to the absolute bottom and begin falling the next piece, and we don't want to surprise the player by having those pieces immediately start moving just because they were holding down an arrow key when they hit the space key.



Moving by Holding Down the Key

```
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()
272.      if movingDown and time.time() - lastMoveDownTime > MOVEDOWNFREQ and
isValidPosition(board, fallingPiece, adjY=1):
273.          fallingPiece['y'] += 1
274.          lastMoveDownTime = time.time()
```

Letting the Piece “Naturally” Fall

```

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()
    
```

Drawing Everything on the Screen

```
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)
298.     pygame.display.update()
299.     FPSCLOCK.tick(FPS)
```

- Now that the game loop has handled all events and updated the game state, the game loop just needs to draw the game state to the screen.
- Most of the drawing is handled by other functions, so the game loop code just needs to call those functions.
- Then the call to `pygame.display.update()` makes the display Surface appear on the actual computer screen, and the `tick()` method call adds a slight pause so the game doesn't run too fast.

makeTextObjs(), A Shortcut Function for Making Text

```
302. def makeTextObjs(text, font, color):  
303.     surf = font.render(text, True, color)  
304.     return surf, surf.get_rect()
```

- The makeTextObjs() function just provides us with a shortcut.
 - Given the text, Font object, and a Color object, it calls render() for us and returns the Surface and Rect object for this text.
 - This just saves us from typing out the code to create the Surface and Rect object each time we need them.

The Same Old terminate() Function

```
307. def terminate():  
308.     pygame.quit()  
309.     sys.exit()
```

- The terminate() function works the same as in the previous game programs.



Waiting for a Key Press Event with the `checkForKeyPress()` Function


```
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
```

A Generic Text Screen Function

```

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)
342.     while checkForKeyPress() == None:
343.         pygame.display.update()
344.         FPSLOCK.tick()

```


 This small loop will constantly call `pygame.display.update()` and `FPSLOCK.tick()` until `checkForKeyPress()` returns a value other than `None`.
 This happens when the user presses a key.



The checkForQuit() Function

```
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
```

The calculateLevelAndFallFreq() Function

```
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
```

- Every time the player completes a line, their score will increase by one point.
- Every ten points, the game goes up a level and the pieces start falling down faster.
- Both the level and the falling frequency can be calculated from the score that is passed to this function.
- You can think of the $\text{level} * 0.02$ part of the equation as “for every level, the piece will fall 0.02 seconds faster than the previous level.”
- !!! You can see that at level 14, the falling frequency will be less than 0.
- !!! If the falling frequency is negative, then the condition on line 277 will always be True and the piece will fall on every iteration of the game loop.
- !!! From level 14 and beyond, the piece cannot fall any faster.

Generating Pieces with the getNewPiece() Function

```
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
```

Adding Pieces to the Board Data Structure

```
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']
```

Creating a New Board Data Structure

```
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
```

The isOnBoard() Function

```
390. def isOnBoard(x, y):
391.     return x >= 0 and x < BOARDWIDTH and y < BOARDHEIGHT
```

- The isOnBoard() is a simple function which checks that the XY coordinates that are passed represent valid values that exist on the board.
- As long as both the XY coordinates are not less 0 or greater than or equal to the BOARDWIDTH and BOARDHEIGHT constants, then the function returns True.

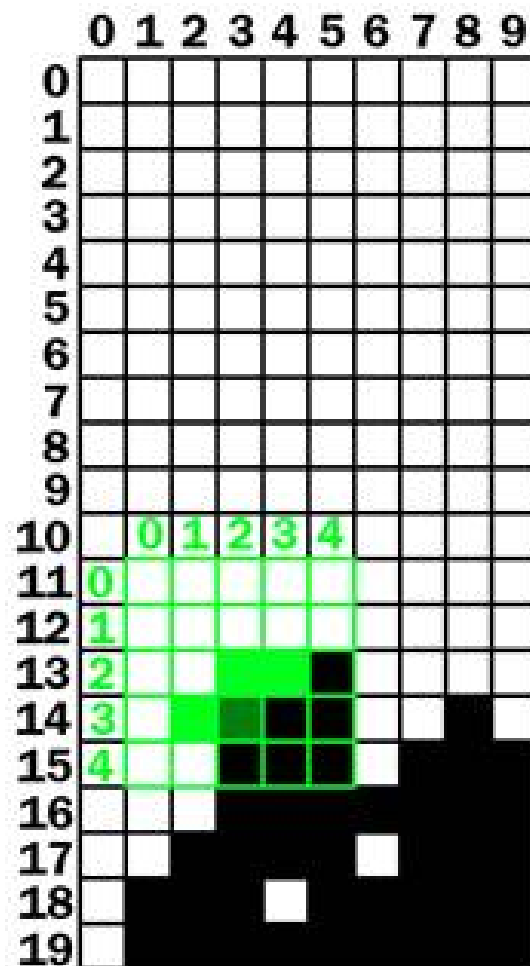
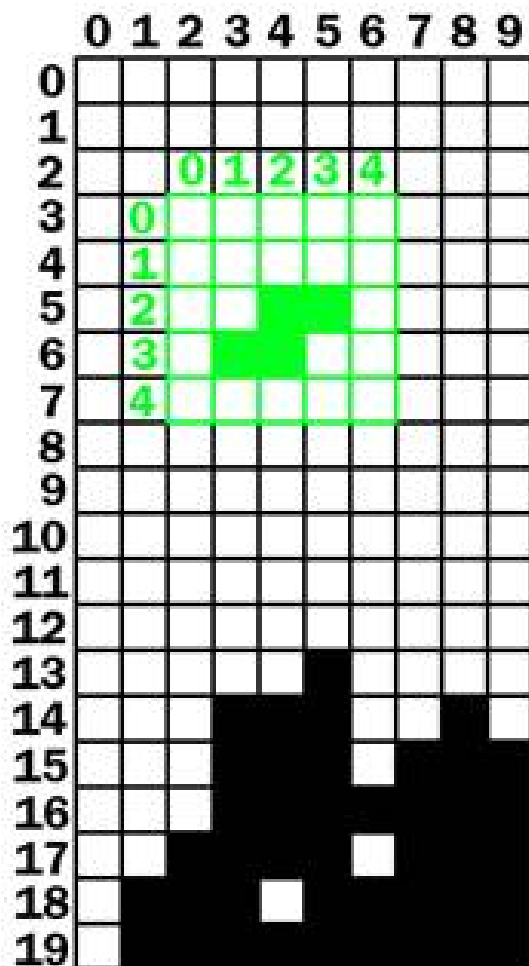
The isValidPosition() Function

```

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
    
```

- The isValidPosition() function is given a board data structure and a piece data structure, and returns True if all boxes in the piece are both on the board and not overlapping any boxes on the board.



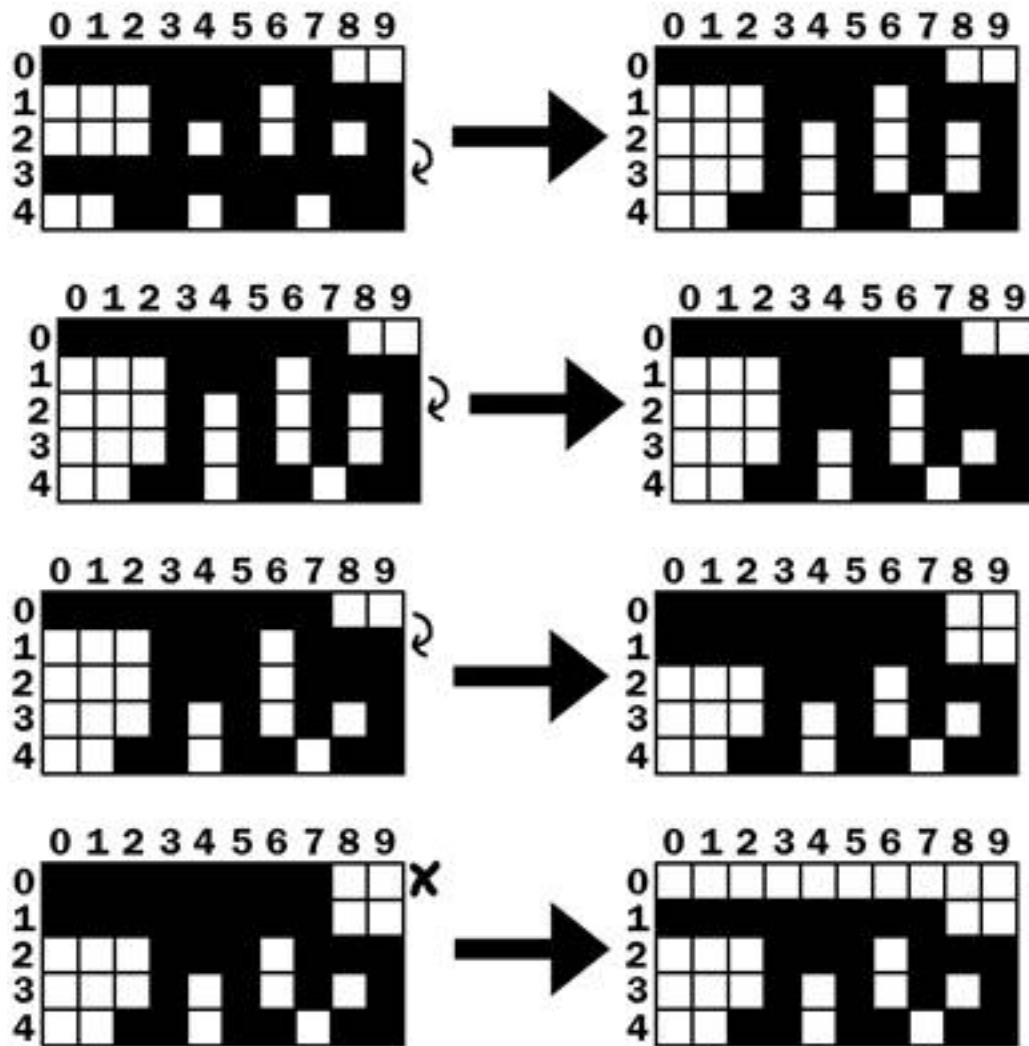


Checking for Complete Lines

```
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
```

Removing Complete Lines

```
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
```



Convert from Board Coordinates to Pixel Coordinates

- 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))`

Drawing a Box on the Board or Elsewhere on the Screen

```

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))

```



Drawing Everything to the Screen

```
456. def drawBoard(board):
457.     # draw the border around the board
458.     pygame.draw.rect(DISPLAYSURF, BORDERCOLOR, XMARGIN - 3, TOPMARGIN - 7,
459. (BOARDWIDTH * BOXSIZE) + 8, (BOARDHEIGHT * BOXSIZE) + 8), 5)
460.
461.     # fill the background of the board
462.     pygame.draw.rect(DISPLAYSURF, BGCOLOR, (XMARGIN, TOPMARGIN, BOXSIZE *
463. BOARDWIDTH, BOXSIZE * BOARDHEIGHT))
464.     # draw the individual boxes on the board
465.     for x in range(BOARDWIDTH):
466.         for y in range(BOARDHEIGHT):
467.             drawBox(x, y, board[x][y])
```


Drawing the Score and Level Text

```
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)
```

Drawing a Piece on the Board or Elsewhere on the Screen

```
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
486.         # stored in the piece data structure
487.         pixelx, pixely = convertToPixelCoords(piece['x'], piece['y'])
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))
```



Drawing the “Next” Piece

```
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()
```

Summary

- The Tetromino game (a clone of “Tetris”) is pretty easy to explain to someone in English:
 - “Blocks fall from the top of a board, and the player moves and rotates them so that they form complete lines. The complete lines disappear (giving the player points) and the lines above them move down. The game keeps going until the blocks fill up the entire board and the player loses.”
- The original Tetris game was designed and programmed one person, Alex Pajitnov, in the Soviet Union in 1984.
- The game is simple, fun, and addictive.
- It is one of the most popular video games ever made, and has sold 100 million copies with many people creating their own clones and variations of it.

Summary

- For additional programming practice
 - you can download buggy versions of Tetromino from <http://invpy.com/buggy/tetromino> and try to figure out how to fix the bugs.
 - “Pentomino” is a version of this game with pieces made up of five boxes.
 - There is also “Tetromino for Idiots”, where all of the pieces are made up of just one box.
 - These variations can be downloaded from:
<http://invpy.com/pentomino.py>
<http://invpy.com/tetrominoforidiots.py>