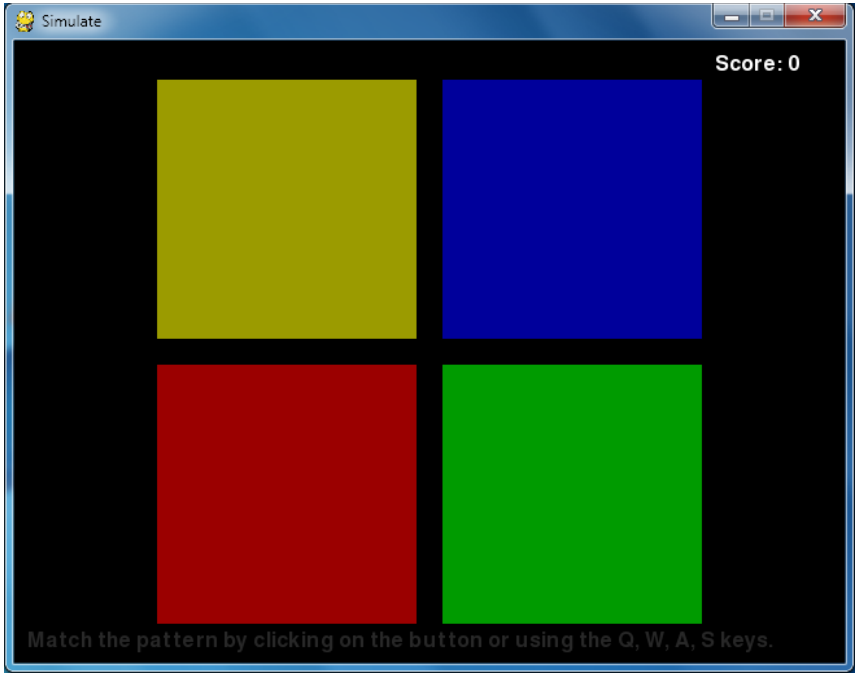
Simulate

How to Play Simulate?



- There are four colored buttons on the screen.
- The buttons light up in a certain random pattern. Then the player must repeat this pattern by pressing the buttons in the correct order.
- Each time the player successfully simulates the pattern, the pattern gets longer.
- The player tries to match the pattern for as long as possible.

Source Code to Simulate

- This source code can be downloaded from

  http://invpy.com/simulate.py

- You can download the four sound files that this program uses from:
  - http://invpy.com/beep1.ogg
  - http://invpy.com/beep2.ogg
  - http://invpy.com/beep3.ogg
  - http://invpy.com/beep4.ogg
- Task 1 – download the code
- Task 2 – activate the game

The Usual Starting Stuff (1)

```
1. # Simulate (a Simon clone)
2. # By Al Sweigart al@inventwithpython.com
3. # http://inventwithpython.com/pygame
4. # Creative Commons BY-NC-SA 3.0 US
5.
6. import random, sys, time, pygame
7. from pygame.locals import *
8.
9. FPS = 30
10. WINDOWWIDTH = 640
11. WINDOWHEIGHT = 480
12. FLASHSPEED = 500 # in milliseconds
13. FLASHDELAY = 200 # in milliseconds
14. BUTTONSIZE = 200
15. BUTTONGAPSIZE = 20
16. TIMEOUT = 4 # seconds before game over if no button is pushed.
```

- Set up the usual constants for things that we might want to modify later:
  - the size of the four buttons,
  - the shades of color used for the buttons (the bright colors are used when the buttons light up) and
  - the amount of time the player has to push the next button in the sequence before the game times out.

The Usual Starting Stuff (2)

```
17.
18. #                R    G    B
19. WHITE         = (255, 255, 255)
20. BLACK         = (  0,   0,   0)
21. BRIGHTRED     = (255,   0,   0)
22. RED           = (155,   0,   0)
23. BRIGHTGREEN   = (  0, 255,   0)
24. GREEN         = (  0, 155,   0)
25. BRIGHTBLUE    = (  0,   0, 255)
26. BLUE          = (  0,   0, 155)
27. BRIGHTYELLOW  = (255, 255,   0)
28. YELLOW        = (155, 155,   0)
29. DARKGRAY      = ( 40,  40,  40)
30. bgColor = BLACK
31.
32. XMARGIN = int((WINDOWWIDTH - (2 * BUTTONSIZE) - BUTTONGAPSIZE) / 2)
33. YMARGIN = int((WINDOWHEIGHT - (2 * BUTTONSIZE) - BUTTONGAPSIZE) / 2)
```

Setting Up the Buttons

```
35. # Rect objects for each of the four buttons
36. YELLOWRECT = pygame.Rect(XMARGIN, YMARGIN, BUTTONSIZE, BUTTONSIZE)
37. BLUERECT   = pygame.Rect(XMARGIN + BUTTONSIZE + BUTTONGAPSIZE, YMARGIN,
BUTTONSIZE, BUTTONSIZE)
38. REDRECT    = pygame.Rect(XMARGIN, YMARGIN + BUTTONSIZE + BUTTONGAPSIZE,
BUTTONSIZE, BUTTONSIZE)
39. GREENRECT  = pygame.Rect(XMARGIN + BUTTONSIZE + BUTTONGAPSIZE, YMARGIN +
BUTTONSIZE + BUTTONGAPSIZE, BUTTONSIZE, BUTTONSIZE)
```

- Four rectangular areas and code to handle when the player clicks inside of those areas.
- The program will need Rect objects for the areas of the four buttons so it can call the collidepoint() method on them.
- Lines 36 to 39 set up these Rect objects with the appropriate coordinates and sizes.

The main() Function (1)

```
41. def main():
42.     global FPSCLOCK, DISPLAYSURF, BASICFONT, BEEP1, BEEP2, BEEP3, BEEP4
43.
44.     pygame.init()
45.     FPSCLOCK = pygame.time.Clock()
46.     DISPLAYSURF = pygame.display.set_mode((WINDOWWIDTH, WINDOWHEIGHT))
47.     pygame.display.set_caption('Simulate')
48.
49.     BASICFONT = pygame.font.Font('freesansbold.ttf', 16)
50.
51.     infoSurf = BASICFONT.render('Match the pattern by clicking on the
button or using the Q, W, A, S keys.', 1, DARKGRAY)
52.     infoRect = infoSurf.get_rect()
53.     infoRect.topleft = (10, WINDOWHEIGHT - 25)
54.     # load the sound files
55.     BEEP1 = pygame.mixer.Sound('beep1.ogg')
56.     BEEP2 = pygame.mixer.Sound('beep2.ogg')
57.     BEEP3 = pygame.mixer.Sound('beep3.ogg')
58.     BEEP4 = pygame.mixer.Sound('beep4.ogg')
```

The main() Function (2)

- The usual Pygame setup functions are called to:
  - initialize the library,
  - create a Clock object,
  - create a window,
  - set the caption, and
  - create a Font object that will be used to display the score and the instructions on the window.
- The objects that are created by these function calls will be stored in global variables so that they can be used in other functions. But they are basically constants since the value in them is never changed.

- Lines 55 to 58 will load sound files so that Simulate can play sound effects as the player clicks on each button. The pygame.mixer.Sound() constructor function will return a Sound object, which we store in the variables BEEP1 to BEEP4 which were made into global variables on line 42.

Some Local Variables Used in This Program (1)

```
60.     # Initialize some variables for a new game
61.     pattern = [] # stores the pattern of colors
62.     currentStep = 0 # the color the player must push next
63.     lastClickTime = 0 # timestamp of the player's last button push
64.     score = 0
65.     # when False, the pattern is playing. when True, waiting for the
player to click a colored button:
66.     waitingForInput = False
```

- The pattern variable will be a list of color values (either YELLOW, RED, BLUE, or GREEN) to keep track of the pattern that the player must memorize.
  - As the player finishes each round, a new random color is added to the end of the list.
- The currentStep variable will keep track of which color in the pattern list the player has to click next. If currentStep was 0 and pattern was [GREEN, RED, RED, YELLOW], then the player would have to click the green button. If they clicked on any other button, the code will cause a game over.

Some Local Variables Used in This Program (2)

- There is a TIMEOUT constant that makes the player click on next button in the pattern within a number of seconds, otherwise the code causes a game over. In order to check if enough time has passed since the last button click, the lastClickTime variable needs to keep track of the last time the player clicked on a button. (module - time and a time.time() function)
- The score variable keeps track of the score.
- There are also two modes that the program will be in:
  - The program is playing the pattern of buttons for the player (waitingForInput is set to False),
  - The program has finished playing the pattern and is waiting for the user to click the buttons in the correct order (waitingForInput is set to True).

Drawing the Board and Handling Input

```
 68.     while True: # main game loop
 69.         clickedButton = None # button that was clicked (set to YELLOW,
RED, GREEN, or BLUE)
 70.         DISPLAYSURF.fill(bgColor)
 71.         drawButtons()
 72.
 73.         scoreSurf = BASICFONT.render('Score: ' + str(score), 1, WHITE)
 74.         scoreRect = scoreSurf.get_rect()
 75.         scoreRect.topleft = (WINDOWWIDTH - 100, 10)
 76.         DISPLAYSURF.blit(scoreSurf, scoreRect)
 77.
 78.         DISPLAYSURF.blit(infoSurf, infoRect)
```

- Line 68 - the start of the main game loop.
- The clickedButton will be reset to None at the beginning of each iteration. If a button is clicked during this iteration, then clickedButton will be set to one of the color values to match the button (YELLOW, RED, GREEN, or BLUE).
- The fill() method - to repaint the entire display Surface so that we can start drawing from scratch. The four colored buttons are drawn with a call to the drawButtons(). Then the text for the score is created on lines 73 to 76.
- Text - tells the player what their current score is. The text for the score changes. It starts off as 'Score: 0' and then becomes 'Score: 1' and then 'Score: 2' and so on.

Checking for Mouse Clicks

```
80.         checkForQuit()
81.         for event in pygame.event.get(): # event handling loop
82.             if event.type == MOUSEBUTTONUP:
83.                 mousex, mousey = event.pos
84.                 clickedButton = getButtonClicked(mousex, mousey)
```

- Line 80 does a quick check for any QUIT events
- Line 81 is the start of the event handling loop
- The XY coordinates of any mouse clicks will be stored in the mousex and mousey variables.
  - If the mouse click was over one of the four buttons, then our getButtonClicked() function will return a Color object of the button clicked (otherwise it returns None).

Checking for Keyboard Presses

```
85.             elif event.type == KEYDOWN:
86.                 if event.key == K_q:
87.                     clickedButton = YELLOW
88.                 elif event.key == K_w:
89.                     clickedButton = BLUE
90.                 elif event.key == K_a:
91.                     clickedButton = RED
92.                 elif event.key == K_s:
93.                     clickedButton = GREEN
```

- Lines 85 to 93 check for any KEYDOWN events (created when the user presses a key on the keyboard). The Q, W, A, and S keys correspond to the buttons because they are arranged in a square shape on the keyboard.
  - The Q key is in the upper left of the four keyboard keys, just like the yellow button on the screen is in the upper left, so we will make pressing the Q key the same as clicking on the yellow button.
  - We can do this by setting the clickedButton variable to the value in the constant variable YELLOW. We can do the same for the three other keys.
- This way, the user can play Simulate with either the mouse or keyboard.

The Two States of the Game Loop

```
 97.         if not waitingForInput:
 98.             # play the pattern
 99.             pygame.display.update()
100.             pygame.time.wait(1000)
101.             pattern.append(random.choice((YELLOW, BLUE, RED, GREEN)))
102.             for button in pattern:
103.                 flashButtonAnimation(button)
104.                 pygame.time.wait(FLASHDELAY)
105.             waitingForInput = True
```

- There are two different “modes” or “states” that the program can be in.
  - waitingForInput - False, the program will be displaying the animation for the pattern.
  - waitingForInput - True, the program will be waiting for the user to select buttons.
- Lines 97 to 105 - the case where the program displays the pattern animation. Since this is done at the start of the game or when the player finishes a pattern, line 101 will add a random color to the pattern list to make the pattern one step longer. Then lines 102 to 104 loops through each of the values in the pattern list and calls flashButtonAnimation() which makes that button light up. After it is done lighting up all the buttons in the pattern list, the program sets the waitingForInput variable to True.

Figuring Out if the Player Pressed the Right Buttons (1)

```
106.        else:
107.            # wait for the player to enter buttons
108.            if clickedButton and clickedButton == pattern[currentStep]:
109.                # pushed the correct button
110.                flashButtonAnimation(clickedButton)
111.                currentStep += 1
112.                lastClickTime = time.time()
```

- Line 106 - If waitingForInput is True
- Line 108 - checks if the player has clicked on a button during this iteration of the game loop and if that button was the correct one.
  - The currentStep variable keeps track of the index in the pattern list for the button that the player should click on next.
  - Example: the pattern - [YELLOW, RED, RED], the currentStep variable - 0 (like it would be when the player first starts the game), then the correct button for the player to click would be pattern[0] (the yellow button).
- If the player has clicked on the correct button -> flash the button the player clicked by calling flashButtonAnimation() then, increase the currentStep to the next step, and then update the lastClickTime variable to the current time.

Figuring Out if the Player Pressed the Right Buttons (2)

```
114.                 if currentStep == len(pattern):
115.                     # pushed the last button in the pattern
116.                     changeBackgroundAnimation()
117.                     score += 1
118.                     waitingForInput = False
119.                     currentStep = 0 # reset back to first step
```

- Lines 114 to 119 are inside the else statement that started on line 106. We know the player clicked on a button and also it was the correct button. Line 114 checks if this was the last correct button in the pattern list by checking if the integer stored in currentStep is equal to the number of values inside the pattern list.
- If this is True, then we want to change the background color by calling our changeBackgroundAnimation(). This is a simple way to let the player know they have entered the entire pattern correctly. The score is incremented, currentStep is set back to 0, and the waitingForInput variable is set to False so that on the next iteration of the game loop the code will add a new Color value to the pattern list and then flash the buttons.

Figuring Out if the Player Pressed the Right Buttons (3)

```
121.             elif (clickedButton and clickedButton != pattern[currentStep])
or (currentStep != 0 and time.time() - TIMEOUT > lastClickTime):
```

- Line 121 - If the player did not click on the correct button or the player has waited too long to click on a button. Either way, we need to show the “game over” animation and start a new game.
- In order to “time out”, it must not be the player’s first button click. But once they’ve started to click buttons, they must keep clicking the buttons quickly enough until they’ve entered the entire pattern (or have clicked on the wrong pattern and gotten a —game over‖). If currentStep != 0 is True, then we know the player has begun clicking the buttons.

Epoch Time (1)

```
>>> import time
>>> time.time()
1320460242.118
```

- Also in order to —time outll, the current time (returned by time.time()) minus four seconds (because 4 is stored in TIMEOUT) must be greater than the last time clicked a button (stored in lastClickTime).
- The reason why time.time() - TIMEOUT > lastClickTime works has to do with how epoch time works. Epoch time (also called Unix epoch time) is the number of seconds it has been since January 1st, 1970. This date is called the Unix epoch.

Epoch Time (2)

```
122.                # pushed the incorrect button, or has timed out
123.                gameOverAnimation()
124.                # reset the variables for a new game:
125.                pattern = []
126.                currentStep = 0
127.                waitingForInput = False
128.                score = 0
129.                pygame.time.wait(1000)
130.                changeBackgroundAnimation()
```

- Going back to line 121, if time.time() - TIMEOUT > lastClickTime evaluates to True, then it has been longer than 4 seconds since time.time() was called and stored in lastClickTime. If it evaluates to False, then it has been less than 4 seconds.
- If either the player clicked on the wrong button or has timed out, the program should play the —game overll animation and then reset the variables for a new game. This involves setting the pattern list to a blank list, currentStep to 0, waitingForInput to False, and then score to 0. A small pause and a new background color will be set to indicate to the player the start of a new game, which will begin on the next iteration of the game loop.

Drawing the Board to the Screen

```
132.            pygame.display.update()
133.            FPSCLOCK.tick(FPS)
```

- Just like the other game programs, the last thing done in the game loop is drawing the display Surface object to the screen and calling the tick() method.

Same Old terminate() Function

```
136. def terminate():
137.     pygame.quit()
138.     sys.exit()
139.
140.
141. def checkForQuit():
142.     for event in pygame.event.get(QUIT): # get all the QUIT events
143.         terminate() # terminate if any QUIT events are present
144.     for event in pygame.event.get(KEYUP): # get all the KEYUP events
145.         if event.key == K_ESCAPE:
146.             terminate() # terminate if the KEYUP event was for the Esc key
147.         pygame.event.post(event) # put the other KEYUP event objects back
```

- The terminate() and checkForQuit() functions were used and explained in the Sliding Puzzle chapter, so we will skip describing them again.

Reusing The Constant Variables

```
150. def flashButtonAnimation(color, animationSpeed=50):
151.     if color == YELLOW:
152.         sound = BEEP1
153.         flashColor = BRIGHTYELLOW
154.         rectangle = YELLOWRECT
155.     elif color == BLUE:
156.         sound = BEEP2
157.         flashColor = BRIGHTBLUE
158.         rectangle = BLUERECT
159.     elif color == RED:
160.         sound = BEEP3
161.         flashColor = BRIGHTRED
162.         rectangle = REDRECT
163.     elif color == GREEN:
164.         sound = BEEP4
165.         flashColor = BRIGHTGREEN
166.         rectangle = GREENRECT
```

- Depending on which Color value is passed as an argument for the color parameter, the sound, color of the bright flash, and rectangular area of the flash will be different.
- Line 151 to 166 sets three local variables differently depending on the value in the color parameter: sound, flashColor, and rectangle.

Animating the Button Flash (1)

```
168.     origSurf = DISPLAYSURF.copy()
169.     flashSurf = pygame.Surface((BUTTONSIZE, BUTTONSIZE))
170.     flashSurf = flashSurf.convert_alpha()
171.     r, g, b = flashColor
172.     sound.play()
```

- Brightening up - on each frame of the animation, the normal board is drawn and then on top of that, the bright color version of the button that is flashing is drawn over the button. The alpha value of the bright color starts off at 0 for the first frame of animation, but then on each frame after the alpha value is slowly increased until it is fully opaque and the bright color version completely paints over the normal button color.
- Button dimming – same as previous, but the alpha value is decreasing each frame.
- The sound will be playing during the button flash animation

Animating the Button Flash (2)

```
173.     for start, end, step in ((0, 255, 1), (255, 0, -1)): # animation loop
174.         for alpha in range(start, end, animationSpeed * step):
175.             checkForQuit()
176.             DISPLAYSURF.blit(origSurf, (0, 0))
177.             flashSurf.fill((r, g, b, alpha))
178.             DISPLAYSURF.blit(flashSurf, rectangle.topleft)
179.             pygame.display.update()
180.             FPSCLOCK.tick(FPS)
181.     DISPLAYSURF.blit(origSurf, (0, 0))
```

Drawing the Buttons

```
184. def drawButtons():
185.     pygame.draw.rect(DISPLAYSURF, YELLOW, YELLOWRECT)
186.     pygame.draw.rect(DISPLAYSURF, BLUE,   BLUERECT)
187.     pygame.draw.rect(DISPLAYSURF, RED,    REDRECT)
188.     pygame.draw.rect(DISPLAYSURF, GREEN,  GREENRECT)
```

- Since each of the buttons is just a rectangle of a certain color in a certain place, we just make four calls to pygame.draw.rect() to draw the buttons on the display Surface.
- The Color object and the Rect object we use to position them never change, which is why we stored them in constant variables like YELLOW and YELLOWRECT.

Animating the Background Change (1)

```
191. def changeBackgroundAnimation(animationSpeed=40):
192.     global bgColor
193.     newBgColor = (random.randint(0, 255), random.randint(0, 255),
random.randint(0, 255))
194.
195.     newBgSurf = pygame.Surface((WINDOWWIDTH, WINDOWHEIGHT))
196.     newBgSurf = newBgSurf.convert_alpha()
197.     r, g, b = newBgColor
198.     for alpha in range(0, 255, animationSpeed): # animation loop
199.         checkForQuit()
200.         DISPLAYSURF.fill(bgColor)
201.
202.         newBgSurf.fill((r, g, b, alpha))
203.         DISPLAYSURF.blit(newBgSurf, (0, 0))
204.
205.         drawButtons() # redraw the buttons on top of the tint
206.
207.         pygame.display.update()
208.         FPSCLOCK.tick(FPS)
209.     bgColor = newBgColor
```

Animating the Background Change (2)

- The background color change animation happens whenever the player finishes entering the entire pattern correctly. On each iteration through the loop (starting from line 198) the entire display Surface has to be redrawn:
  - Line 200 fills in the entire display Surface (stored in DISPLAYSURF) with the old background color (which is stored in bgColor).
  - Line 202 fills in a different Surface object (stored in newBgSurf) with the new background color's RGB values (and the alpha transparency value changes on each iteration since that is what the for loop on line 198 does).
  - Line 203 then draws the newBgSurf Surface to the display Surface in DISPLAYSURF. The reason we didn't just paint our semitransparent new background color on DISPLAYSURF to begin with is because the fill() method will just replace the color on the Surface, whereas the blit() method will blend the colors.
  - Now that we have the background the way we want it, we'll draw the buttons over it with a call to drawButtons() on line 205.
  - Line 207 and 208 then just draws the display Surface to the screen and adds a pause.

The Game Over Animation (1)

```
212. def gameOverAnimation(color=WHITE, animationSpeed=50):
213.     # play all beeps at once, then flash the background
214.     origSurf = DISPLAYSURF.copy()
215.     flashSurf = pygame.Surface(DISPLAYSURF.get_size())
216.     flashSurf = flashSurf.convert_alpha()
217.     BEEP1.play() # play all four beeps at the same time, roughly.
218.     BEEP2.play()
219.     BEEP3.play()
220.     BEEP4.play()
221.     r, g, b = color
222.     for i in range(3): # do the flash 3 times
```

- Each of the iterations of the for loop on the line 223 and the next lines will perform a flash. To have three flashes done, we put all of that code in a for loop that has three iterations. If you want more or fewer flashes, then change the integer that is passed to range() on line 222.

The Game Over Animation (2)

```
223.         for start, end, step in ((0, 255, 1), (255, 0, -1)):
224.             # The first iteration in this loop sets the following for loop
225.             # to go from 0 to 255, the second from 255 to 0.
226.             for alpha in range(start, end, animationSpeed * step): #
animation loop
227.                 # alpha means transparency. 255 is opaque, 0 is invisible
228.                 checkForQuit()
229.                 flashSurf.fill((r, g, b, alpha))
230.                 DISPLAYSURF.blit(origSurf, (0, 0))
231.                 DISPLAYSURF.blit(flashSurf, (0, 0))
232.                 drawButtons()
233.                 pygame.display.update()
234.                 FPSCLOCK.tick(FPS)
```

- This animation loop works the same as the previous flashing animation code (Animating the Background Change). The copy of the original Surface object stored in origSurf is drawn on the display Surface, then flashSurf (which has the new flashing color painted on it) is blitted on top of the display Surface. After the background color is set up, the buttons are drawn on top on line 232. Finally the display Surface is drawn to the screen with the call to pygame.display.update().
- The for loop on line 226 adjusts the alpha value for the color used for each frame of animation (increasing at first, and then decreasing).

Converting from Pixel Coordinates to Buttons

```
238. def getButtonClicked(x, y):
239.     if YELLOWRECT.collidepoint( (x, y) ):
240.         return YELLOW
241.     elif BLUERECT.collidepoint( (x, y) ):
242.         return BLUE
243.     elif REDRECT.collidepoint( (x, y) ):
244.         return RED
245.     elif GREENRECT.collidepoint( (x, y) ):
246.         return GREEN
247.     return None
248.
249.
250. if __name__ == '__main__':
251.     main()
```

- The getButtonClicked() function simply takes XY pixel coordinates and returns either the values YELLOW, BLUE, RED, or GREEN if one of the buttons was clicked, or returns None if the XY pixel coordinates are not over any of the four buttons.

The Game Over Animation (2)

- The code for getButtonClicked() ends with a “return None” statement on line 247.
- Normally when a function reaches the end and returns the None value implicitly (that is, there is no return statement outright saying that it is returning None) the code that calls it doesn’t care about the return value.
- When getButtonClicked() returns None, it means that the coordinates that were passed to it were not over any of the four buttons. To make it clear that in this case the value None is returned from getButtonClicked(), we have the return None line at the end of the function.
- To make your code more readable, it is better to have your code be explicit (that is, clearly state something even if it might be obvious) rather than implicit (that is, leaving it up to the person reading code to know how it works without outright telling them).
- “Explicit is better than implicit” - one of the Python Koans.