SI@UCF Computer Science Camp
Object Oriented Python and pyGame
Quiz #2 Solutions
Date: 6/17/2026

1) (20 pts) The first version of the fruit game shown in class, each fruit appeared equally. Remember that the fruits were numbered 0, 1, 2 and 3 and we just generated a random integer in between 0 and 3, inclusive and used that to determine which fruit was chosen. In other games, we might have a different number of objects and we may want each one to be generated with different probabilities. Write a function called `returnRandomObject`, which takes in a list of percentages that each object is randomly selected. For example, if the list [23, 37, 5, 10, 25] is passed to the function, this means the function should return 0 about 23% of the time, 1 about 37% of the time 2 about 5% of the time, 3 about 10% of the time and 4 about 25% of the time. Complete the function so that it takes in a list, `percentages`, of positive integers which adds up to exactly 100 and returns an integer in between 0 and `len(percentages)-1`, inclusive, according to the criteria described above.

```
# Random class method
# random.randint(a, b) returns a random integer in between a and b, inclusive.

def returnRandomObject(percentages):

    x = random.randint(0,99)                    // 6 pts
    for i in range(len(percentages)):           // 2 pts

        if x < percentages[i]:                  // 6 pts
            return i

        x -= percentages[i]                     // 6 pts

    return 0                                    // 0 pts
```

Grading Note: There are many ways to do this, this is just one, so you have to read the code carefully.

2) (10 pts) In all 3 versions of the Fruit Game, the move function was written as follows:

```
def move(items):
    for item in items:
        item.move()
```

How would the behavior of the game change if the function were changed as follows:

```
def move(items):
    for item in items:
        item.move()
        item.bounceLeft()
        item.bounceRight()
```

Each object in the game, both the fruit and the bombs, as they fall, would bounce off the left and right sides instead of going off the side of the screen. This would allow the player to capture some fruit they might have missed out on otherwise, but also expose them to bombs that weren't a threat.

Grading: Grader's discretion!

3) (10 pts) In recitation, a Rock-Paper-Scissors program was shown to demonstrate the use of pyGame's built in Sprite class. Describe what occurred when the program was executed and the type of behavior observed amongst all of the objects shown on the screen over time.

The game was initialized to have roughly 1/3 of each object. The objects appeared on screen and would move randomly once the simulation/program started. Whenever two objects collided, the loser in the traditional rock paper scissors match up would change to be the winner. The behavior was cyclic. One of the three would end up increasing in percentage of objects on the screen, but eventually when there were too many of one object on screen, they would start losing battles to their weakness and things would even out slowly. This type of oscillation occurred over and over again, in a random manner. (Namely, if rock dominated and then came back to equilibrium, any one of the three might later dominate. Rock was neither more or less likely to dominate again based on what had previously occurred.)

Grading: Grader's discretion!

4) (20 pts) The following function is from the last version of the fruit game. Answer the questions after the code about it. Several lines are numbered to aid in the questions. Answer each question independently of the others.

```
def update(filename, best, name, score):

    newitem = [score, name]                          # line 3
    best.append(newitem)                             # line 4

    best.sort(reverse=True)                          # line 6

    newscores = open(filename, "w")                  # line 8

    if len(best) > MAXSCORES:                        # line 9
        del best[-1]                                 # line 10

    cnt = len(best)                                  # line 12

    newscores.write(str(cnt)+"\n")                   # line 14
    for i in range(cnt):                             # line 15
        newscores.write(best[i][1]+"\t"+str(best[i][0])+"\n") # line 16
    newscores.close()                                # line 17
```

(a) (5 pts) What would happen if we removed `reverse=True` from line 6?

The scores would be stored in the file from lowest to highest. If this was the code from the beginning of the game when there were no scores, then the scores saved in the file would be the lowest MAXSCORES (or fewer) number of scores.

(b) (5 pts) Explain what line 10 is doing.

Independent of part (a), this line is removing the last item in the list best, which would correspond to the MAXSCORES+1 best score of those available. (Which also means the worst...) This line only triggers when we already have the maximum number of best scores stored, and adding a new score kicks out the lowest of all of the scores from being stored.

(c) (5 pts) On line 16, why do we write `best[i][1]` before we write `best[i][0]`?

best[i][1] stores the name and best[i][0] stores the score. It looks more natural to print the name before the score, but for sorting purposes, we had to store the score first, so that we could sort by score.

(d) (5 pts) What would the effect of removing lines 9 and 10 have?

Then all scores for the game would be stored in the file in sorted order. In practice, since there's limited room on the pyGame surface, eventually names that are in the file just wouldn't show up on the visible screen, because the pyGame code would ask to draw that text outside of the bounds of the surface.

Grading: Grader's discretion!

5) (20 pts) In the Sushi Game, we added a feature not in previous games which allowed the user to enter their name in one of the pyGame surfaces. Here is the relevant code (note some code not necessary for this question is omitted):

```
if event.type == KEYDOWN:

   if event.unicode >= 'a' and event.unicode <= 'z' and len(buffer) < 15:
       buffer = buffer + event.unicode

   if event.unicode >= 'A' and event.unicode <= 'Z' and len(buffer) < 15:
       buffer = buffer + event.unicode

   if event.key == K_BACKSPACE and len(buffer) > 0:
       buffer = buffer[:len(buffer)-1]

   if event.key == K_RETURN:
       return buffer
```

(a) (10 pts) If we wanted the names of players to contain digits, what code should we add to the excerpt above? (Just write the line(s) below and draw an arrow showing where you would insert it.

```
   if event.unicode >= '0' and event.unicode <= '9' and len(buffer) < 15:
       buffer = buffer + event.unicode
```

Can be inserted anywhere as a separate if statement added to the four if statements that are already inside the KEYDOWN if.

Grading: 1 pt if, 2 pts each expression in the and, 3 pts for buffer change

(b) (10 pts) Another way to limit the length of the string returned from this section of the code would be to allow the string to grow longer than 15 characters, but when the string is returned, just return the first 15 characters. (With this solution, the len(buffer) < 15 would be removed from the first set of ifs.) Rewrite the body of the last if to correctly implement this idea.

```
   if event.key == K_RETURN:

        if len(buffer) <= 15:              // 3 pts
            return buffer                  // 3 pts
        return buffer[:15]                 // 4 pts
```

Grading Note: Apparently just return buffer[:15] works because Python allows for slicing out of bounds without any sort of error occurring. (It also does what one might expect.

6) (15 pts) After class on Tuesday, the minesweeper code was slightly refactored, so that in the main file, the code was updated to look like this:

```
if pygame.mouse.get_pressed()[2]:

    sq = minefunc.getXY(event.pos)

    if myMine != None and sq != None and
                          myMine.getStatus(sq[0], sq[1]) != minefunc.SHOWING:

        myMine.flipStatus(sq[0], sq[1])
```

The effect of the last line of code was changing the status of the square at (sq[0], sq[1]), either from COVERED to FLAGGED or FLAGGED to COVERED.

A couple notes: The instance variable in the mineclass that stores the status of each square is `show`. The possible values of `show[x][y]` are `minefunc.FLAGGED`, `minefunc.COVERED` or `minefunc.SHOWING`.

Complete the flipStatus method in the mineclass:

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

    myStatus = self.getStatus(x, y)

    if myStatus == minefunc.SHOWING:
        return

    # Code goes here, should be exactly 4 lines.

    if myStatus == minefunc.COVERED:            // 3 pts
        self.show[x][y] = minefunc.FLAGGED      // 5 pts
    else:                                       // 2 pts
        self.show[x][y] = minefunc.COVERED      // 5 pts
```

9) (5 pts) The nickname of Ghana's World Cup team is the Black Stars. What symbol appears in black on Ghana's flag?

Star (Give to All)