Answers

1. **c**. If we did `doc.find_all("b")`, we would have gotten all three, but we're only searching within the i `element`, so we only get the one.
2. **a**. We get a random number immediately after seeding, and we use the same seed both times, meaning we'll get the same number twice. `choice(5)` returns 0, 1, 2, 3, or 4; so we rule out `5 5`. That leaves `0 0`.
3. **e**. This is tricky! We're appending `items` to itself! There's actually nothing wrong with this. This does mean the last enty in `items` will refer to the same list object referred to by `items` itself. In other words, `items[2]`, `items[-1]`, and `items` all refer to the same object. Thus, you can simplify `items[-1][2][0]` to `items[2][0]`, then to `items[0]`. Similarly, `items[2][-1][0]` reduces to `items[0]`.
4. **b**. Review the vocabulary here as necessary: https://tyler.caraza-harter.com/cs301/spring19/materials/lec-24.pdf (slide 60 and those around it).
5. **b**. Review https://tyler.caraza-harter.com/cs301/spring19/materials/lec-22.pdf as necessary.
6. **c**. Only WHERE and HAVING are SQL clauses. HAVING is for filtering groups, and we don't even have a GROUP BY, so the answer can only be WHERE.
7. **c**. 75 is the average of the P1 scores and 55 is the average of the P2 scores, so we want to compute the average per project (so we need to GROUP BY project).
8. **c**. When trying to understand a complex expression like this, try running portions of it in a notebook. What is `letters`? What does `letters.value_counts()` evaluate to? What does `letters.value_counts().value_counts()` evaluate to? What does `letters.value_counts().value_counts()[3]` evaluate to?
9. **d**. As in 8, break this down to understand. What is `df["y"] > df["x"]`? What is `df["z"] > 7`? What is `(df["y"] > df["x"]) | (df["z"] > 7)`?
10. **b**. Review the behavior of this function as necessary: https://tyler.caraza-harter.com/cs301/spring19/materials/code/lec-37/line-and-bar.html