3. WRITTEN RESPONSES

3 a.

3.a.i.

The program's purpose is to provide the user entertainment and amusement through finding a fictional pet that is personalized to their interests based on questions provided.

3.a.ii.

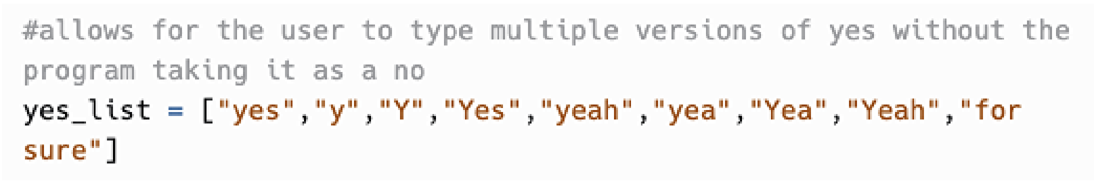
The program uses the user's inputs in order to ask first if they want to participate, and then four multiple-choice questions about their interests. The code then stores the answers in a list, adds a certain number of points that are coordinated to each answer letter (A, B, C, D), and then totals the points for a score. The final pet is determined by the number range of the final score.

3.a.iii.

The input of the program is the user typing their name, then a yes or no depending on if they want to play, and then multiple-choice answers depending on what their interests are. The output is the personalized pet generated as a result of the procedure points_calc.

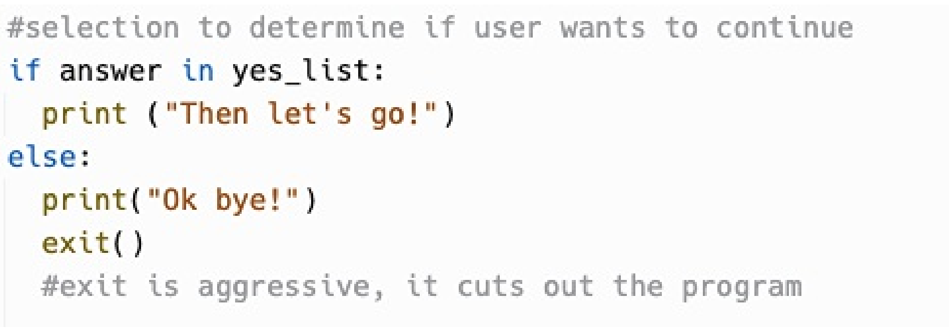
3 b.

3.b.i.



```
#allows for the user to type multiple versions of yes without the
program taking it as a no
yes_list = ["yes","y","Y","Yes","yeah","yea","Yea","Yeah","for
sure"]
```

3.b.ii.



```
#selection to determine if user wants to continue
if answer in yes_list:
  print ("Then let's go!")
else:
  print("Ok bye!")
  exit()
  #exit is aggressive, it cuts out the program
```

3.b.iii.

The list used in the program is named yes_list.

3.b.iv.

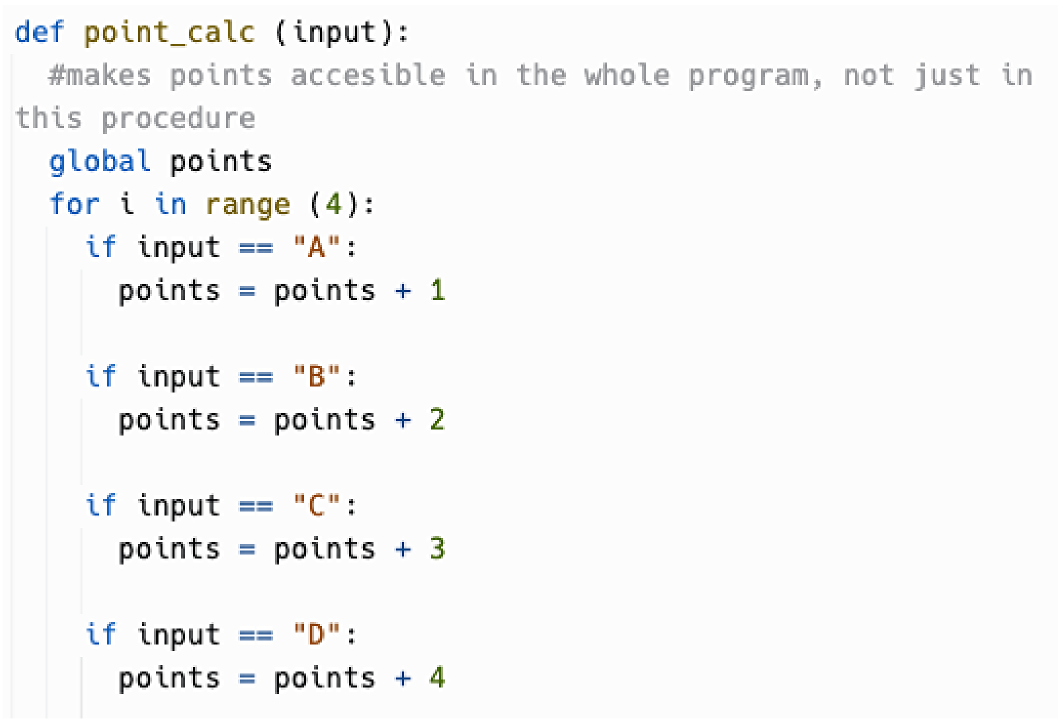
The list stores multiple versions of yes the user might input if they want to continue the program. If the program enters any other answer that is not in the list, then the program terminates and they don't get their personalized pet.

3.b.v.

The list "yes_list" manages complexity because the user could have multiple inputs for the value of yes (the program would stop if the user didn't answer the one specific value of yes). I could have assigned a variable for each yes possibility, which would require an if-statement for each variable to check for equivalency to the user's answer. With the nine options in my list, that would make the code more complex than needed. The list makes the program simpler because if adding more options for yes is easy, I can type them into the list and the program will still run the same.

3 c.

3.c.i.



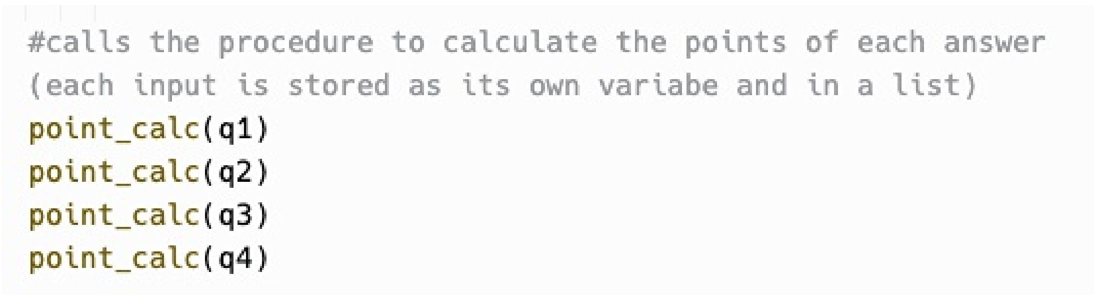
```
def point_calc (input):
  #makes points accesible in the whole program, not just in
this procedure
  global points
  for i in range (4):
    if input == "A":
      points = points + 1

    if input == "B":
      points = points + 2

    if input == "C":
      points = points + 3

    if input == "D":
      points = points + 4
```

3.c.ii.



```
#calls the procedure to calculate the points of each answer
(each input is stored as its own variabe and in a list)
point_calc(q1)
point_calc(q2)
point_calc(q3)
point_calc(q4)
```

3.c.iii.

The procedure, point_calc, adds to the overall functionality because it is integral to output the personalized pet for the user, which is the main point of the program. Since point_calc is necessary to calculate points to determine the pet, it makes the program more efficient by having the procedure written in one place rather than repeated for each of the user answers.

3.c.iv.

The procedure, point_calc, takes one input as its parameter. It makes the existing variable, points, global. Then it evaluates each if-statement four times (due to the “for in range (4)” loop) to see if it applies to the argument. There are four if-statements each for if the argument equals A, B, C, or D. For the one it applies to, the assigned number of points is added to “points”. Each argument will only apply to one of the if-statements, so when it repeats the other three loops the same number of points will be added each time. There is no need for a return value because “points” stores the total points. The procedure demonstrates sequence with the running of the code in order. The if-statements demonstrate selection, and the for loop includes iteration as the procedure repeats the same code four times. In order to increase the range of numbers to determine each pet, I run through the loop multiple times.

3 d.

3.d.i.

First call:

An example argument is q1, the variable assigned to store the answer to the first question. In this example, q1= A (meaning the user picked A).

Second call:

Another argument is q2, the variable assigned to store the answer to the second question. In this example, q2= B.

3 d.ii.

Condition(s) tested by first call:

The condition being tested is if q1 is equal to the string in each statement. Since A is the value of q1, this would activate only the first if-statement. Since the condition is true, the specified number of points is added to “points” (which is one point).

Condition(s) tested by second call:

The condition being tested is if q2 is equal to the string in each statement. Since B is the value of q2, this would activate only the second if-statement. Since the condition is true, the specified number of points is added to “points” (which is two points).

3.d.iii.

Results of the first call:

The result of the parameter q1 is 4 points stored in “points”. The for loop runs through the if-statements four total times, resulting in the “A” if-statement to be activated four times, thus adding one point each time.

Results of the second call:

The result of the parameter q2 is 8 points stored in “points”. The for loop runs through the if-statements four total times, resulting in the “B” if-statement to be activated four times, thus adding two points each time.

The final points then fall in a range that determines the pet.