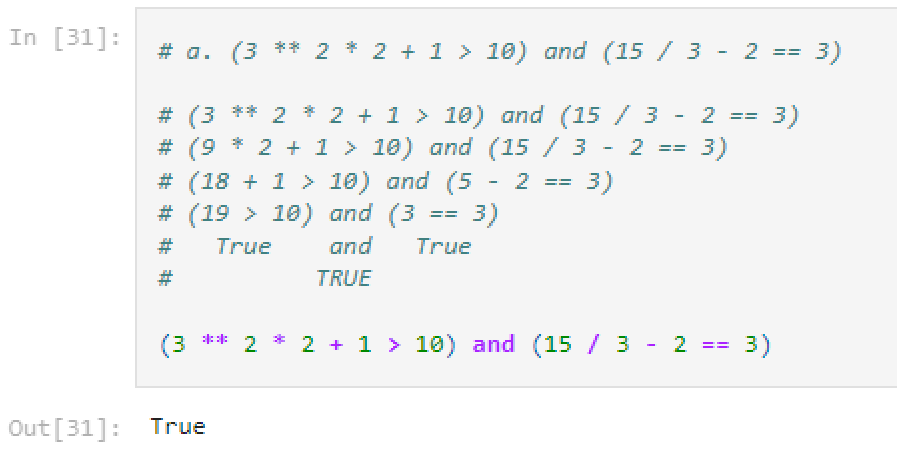
CS521 - Homework 2 - Alexander Bean

1. The PEMDAS Rule

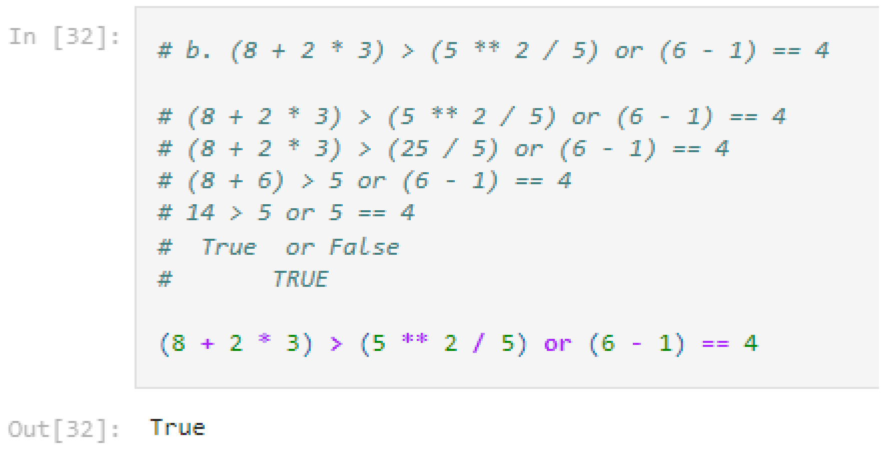
In the following problems (1a-1d), I followed the PEMDAS rule and the Precedence of Relational and Arithmetic Operators table to explain the calculations done. With the operators, I followed the order of priority (not > and > or) to explain the calculations. I broke it down by (mostly) individual steps to clearly show the process, so those steps should provide the necessary explanation.

a. (3 ** 2 * 2 + 1 > 10) and (15 / 3 - 2 == 3)



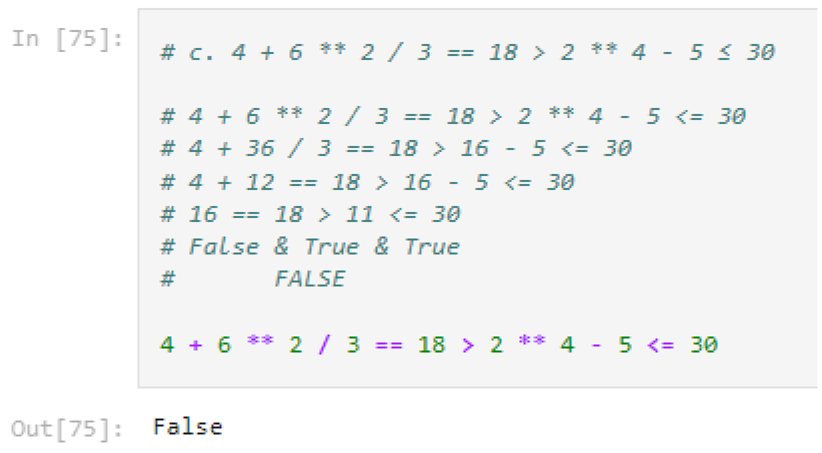
```
In [31]: # a. (3 ** 2 * 2 + 1 > 10) and (15 / 3 - 2 == 3)

         # (3 ** 2 * 2 + 1 > 10) and (15 / 3 - 2 == 3)
         # (9 * 2 + 1 > 10) and (15 / 3 - 2 == 3)
         # (18 + 1 > 10) and (5 - 2 == 3)
         # (19 > 10) and (3 == 3)
         #   True     and   True
         #             TRUE

         (3 ** 2 * 2 + 1 > 10) and (15 / 3 - 2 == 3)

Out[31]: True
```

b. (8 + 2 * 3) > (5 ** 2 / 5) or (6 - 1) == 4

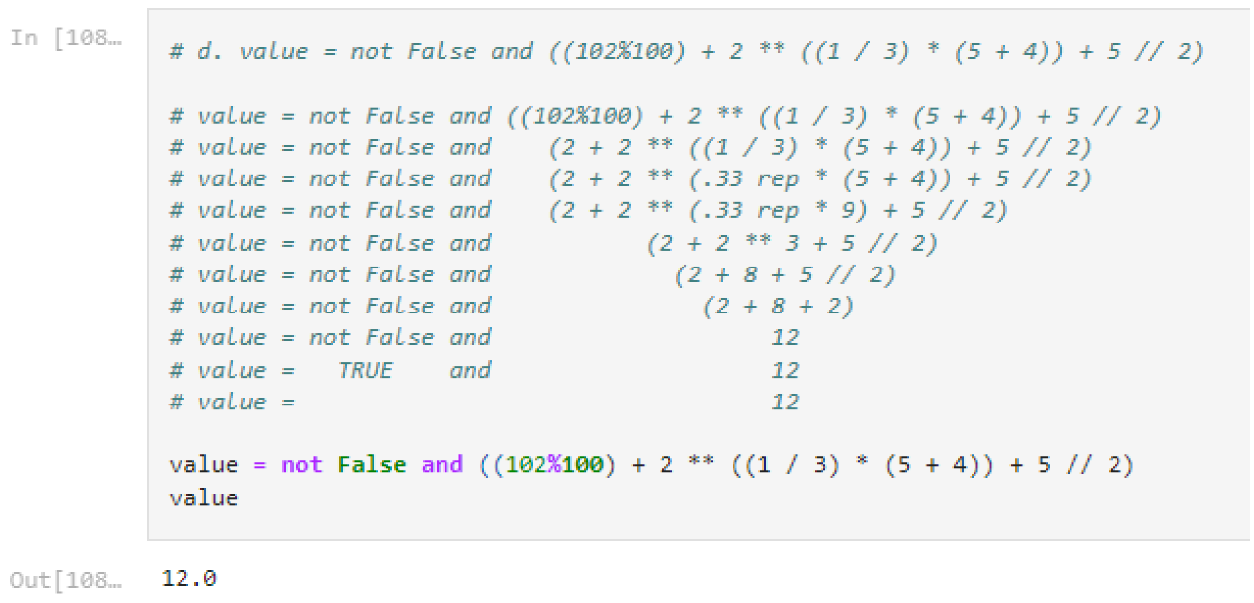


```
In [32]: # b. (8 + 2 * 3) > (5 ** 2 / 5) or (6 - 1) == 4

         # (8 + 2 * 3) > (5 ** 2 / 5) or (6 - 1) == 4
         # (8 + 2 * 3) > (25 / 5) or (6 - 1) == 4
         # (8 + 6) > 5 or (6 - 1) == 4
         # 14 > 5 or 5 == 4
         #  True  or False
         #        TRUE

         (8 + 2 * 3) > (5 ** 2 / 5) or (6 - 1) == 4

Out[32]: True
```

c. 4 + 6 ** 2 / 3 == 18 > 2 ** 4 - 5 ≤ 30


```
In [75]:  # c. 4 + 6 ** 2 / 3 == 18 > 2 ** 4 - 5 ≤ 30

          # 4 + 6 ** 2 / 3 == 18 > 2 ** 4 - 5 <= 30
          # 4 + 36 / 3 == 18 > 16 - 5 <= 30
          # 4 + 12 == 18 > 16 - 5 <= 30
          # 16 == 18 > 11 <= 30
          # False & True & True
          #       FALSE

          4 + 6 ** 2 / 3 == 18 > 2 ** 4 - 5 <= 30

Out[75]:  False
```

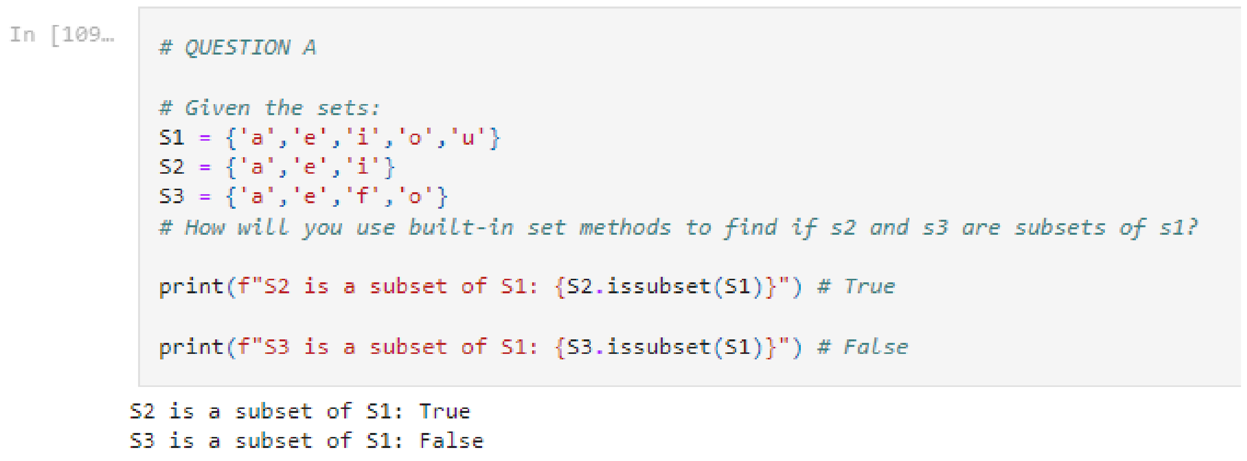
d. **value = not False and ((102%100) + 2 ** ((1 / 3) * (5 + 4)) + 5 // 2)**


```
In [108…  # d. value = not False and ((102%100) + 2 ** ((1 / 3) * (5 + 4)) + 5 // 2)

          # value = not False and ((102%100) + 2 ** ((1 / 3) * (5 + 4)) + 5 // 2)
          # value = not False and    (2 + 2 ** ((1 / 3) * (5 + 4)) + 5 // 2)
          # value = not False and    (2 + 2 ** (.33 rep * (5 + 4)) + 5 // 2)
          # value = not False and    (2 + 2 ** (.33 rep * 9) + 5 // 2)
          # value = not False and            (2 + 2 ** 3 + 5 // 2)
          # value = not False and              (2 + 8 + 5 // 2)
          # value = not False and                (2 + 8 + 2)
          # value = not False and                     12
          # value =   TRUE    and                     12
          # value =                                   12

          value = not False and ((102%100) + 2 ** ((1 / 3) * (5 + 4)) + 5 // 2)
          value

Out[108…  12.0
```

Since ‘not’ is first in the order of relational operations, it is evaluated first. If ‘and’ was evaluated first, then it would mean value = **not** (False and 12), which would produce a completely different result.

2. Sets

Question A

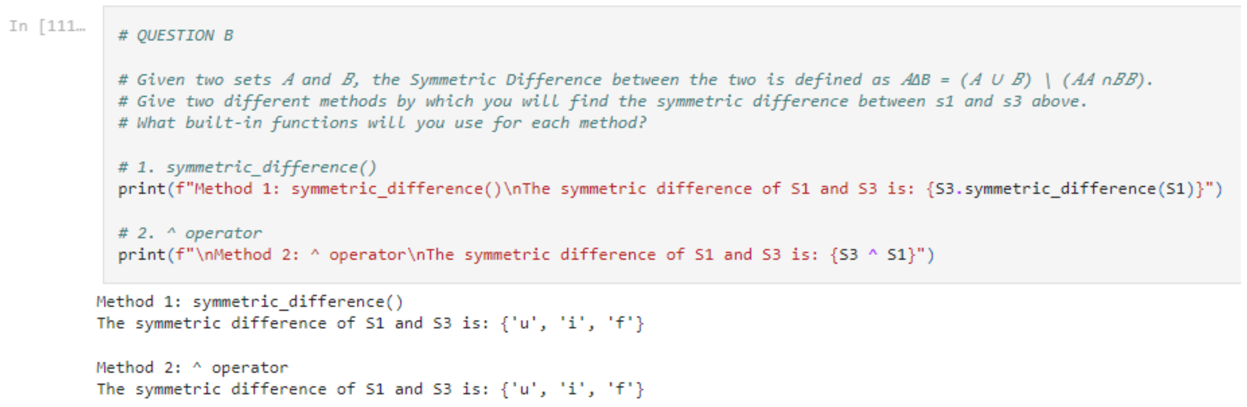


```
In [109…
# QUESTION A

# Given the sets:
S1 = {'a','e','i','o','u'}
S2 = {'a','e','i'}
S3 = {'a','e','f','o'}
# How will you use built-in set methods to find if s2 and s3 are subsets of s1?

print(f"S2 is a subset of S1: {S2.issubset(S1)}") # True

print(f"S3 is a subset of S1: {S3.issubset(S1)}") # False
```

```
S2 is a subset of S1: True
S3 is a subset of S1: False
```

I can use the 'issubset()' method to determine this. This method is used on the supposed substring and contains the comparison string in the parameters. S2 is a subset of S1 because 'a', 'e', and 'i' are all included in S1. S3, however, is NOT a subset–while 'a', 'e', and 'o' are in S1, 'f' is NOT in S1.

Question B



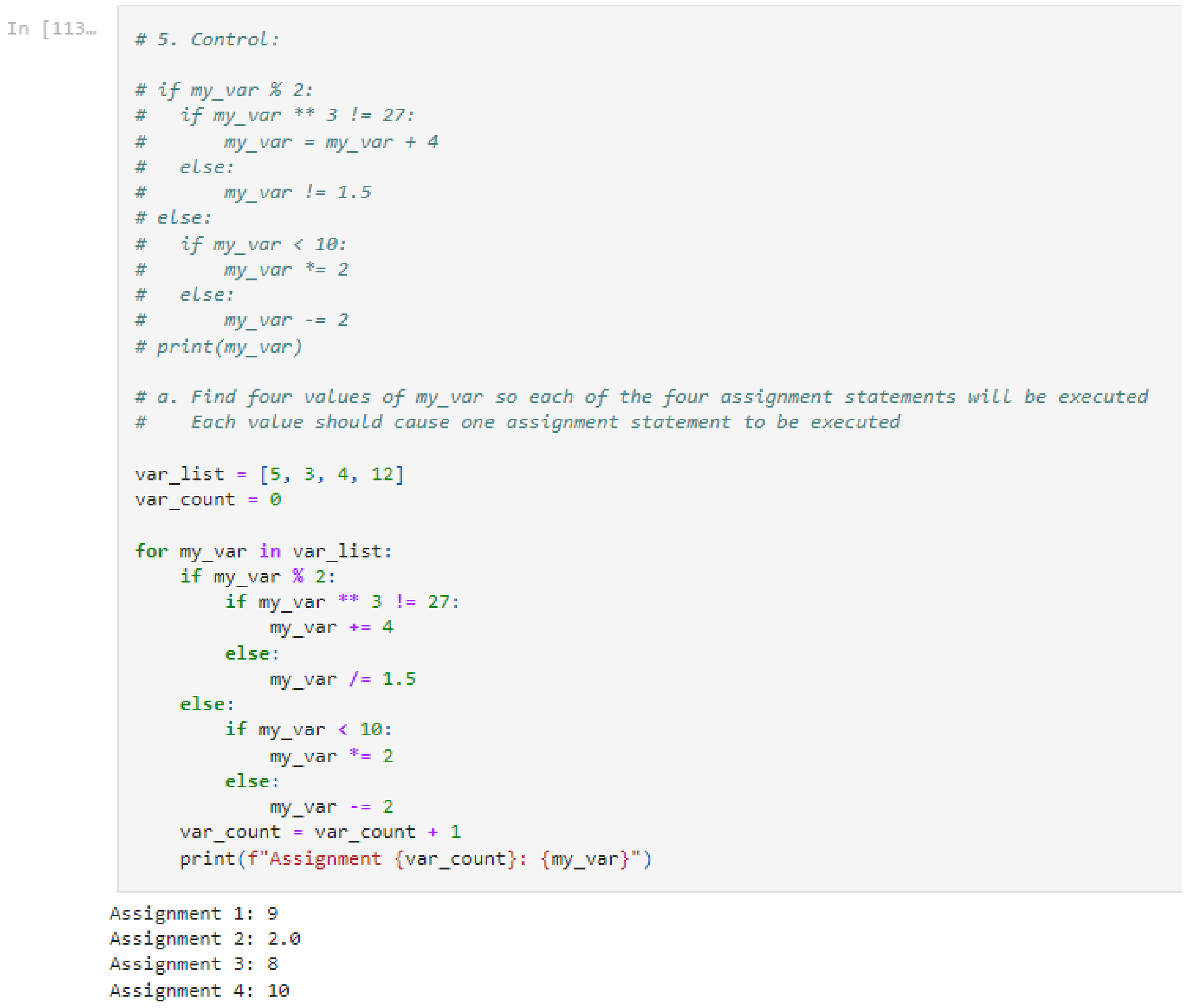
```
In [111…
# QUESTION B

# Given two sets A and B, the Symmetric Difference between the two is defined as AΔB = (A ∪ B) \ (AA ∩BB).
# Give two different methods by which you will find the symmetric difference between s1 and s3 above.
# What built-in functions will you use for each method?

# 1. symmetric_difference()
print(f"Method 1: symmetric_difference()\nThe symmetric difference of S1 and S3 is: {S3.symmetric_difference(S1)}")

# 2. ^ operator
print(f"\nMethod 2: ^ operator\nThe symmetric difference of S1 and S3 is: {S3 ^ S1}")
```

```
Method 1: symmetric_difference()
The symmetric difference of S1 and S3 is: {'u', 'i', 'f'}

Method 2: ^ operator
The symmetric difference of S1 and S3 is: {'u', 'i', 'f'}
```

One method I can use is the symmetric_difference() method. This is used in the same way as issubset() to get the difference in values. I can also use the '^' operator as a shortcut command for symmetric_difference(). It is essentially just shorthand for that method. Both of these methods produce the same result, which is a list of values that are in one of the lists but not both.

3. Control Flow | Questions from Chapter 2

Question 5, Part A:



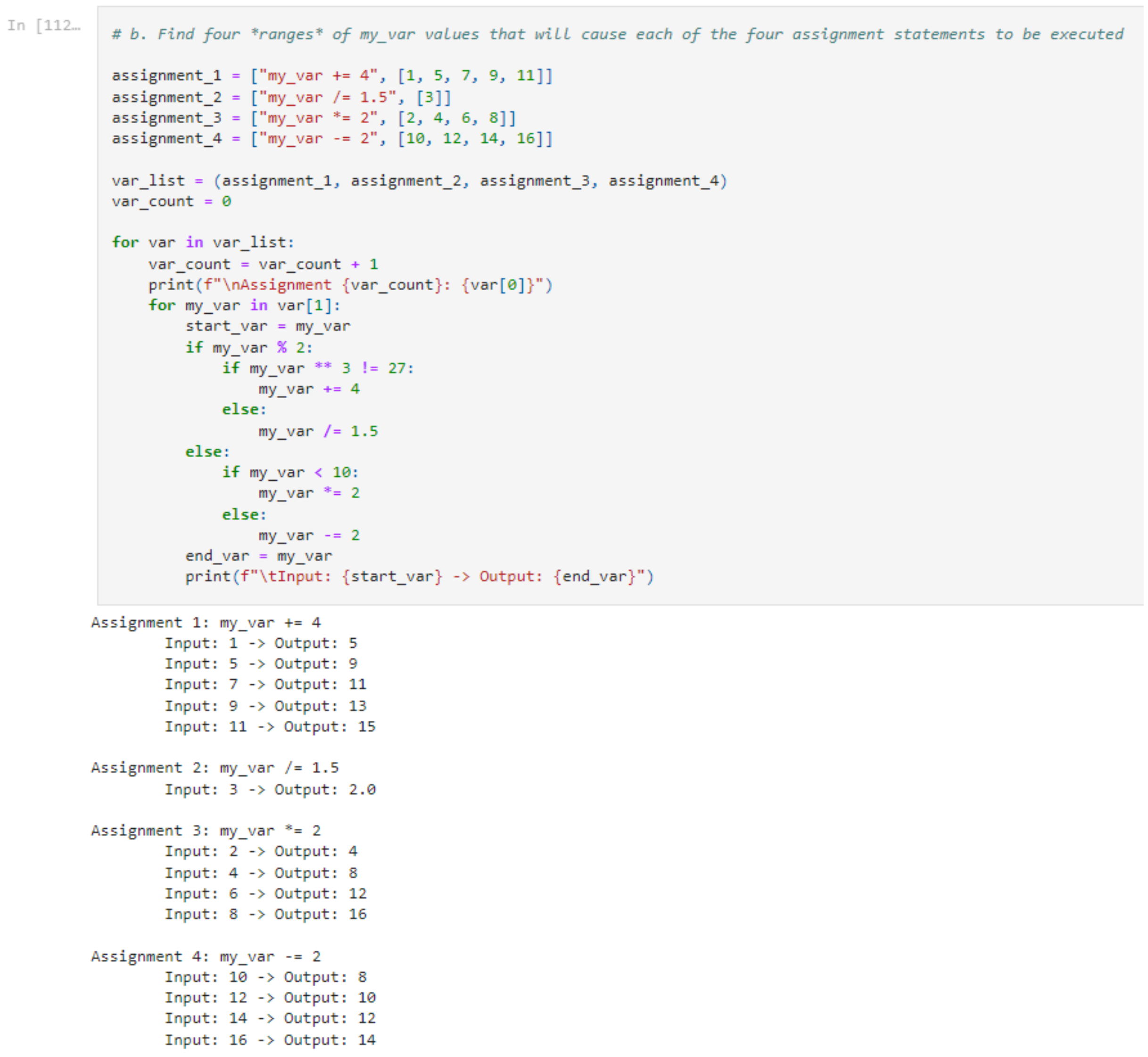
```
In [113…
# 5. Control:

# if my_var % 2:
#   if my_var ** 3 != 27:
#       my_var = my_var + 4
#   else:
#       my_var != 1.5
# else:
#   if my_var < 10:
#       my_var *= 2
#   else:
#       my_var -= 2
# print(my_var)

# a. Find four values of my_var so each of the four assignment statements will be executed
#    Each value should cause one assignment statement to be executed

var_list = [5, 3, 4, 12]
var_count = 0

for my_var in var_list:
    if my_var % 2:
        if my_var ** 3 != 27:
            my_var += 4
        else:
            my_var /= 1.5
    else:
        if my_var < 10:
            my_var *= 2
        else:
            my_var -= 2
    var_count = var_count + 1
    print(f"Assignment {var_count}: {my_var}")
```

```
Assignment 1: 9
Assignment 2: 2.0
Assignment 3: 8
Assignment 4: 10
```

For assignment 1, a valid value is 5, as 5%2 is 1 (True) and 5**3 != 27 is True. For assignment 2, the only valid value is 3. 3%2 is 1 (True) and 3 is the only number where **3 != 27 is False. It is the only number you can cube and get 27. For assignment 3, a valid value is 4, as 4%2 is 0 (False) and 4 < 10 is True. For assignment 4, a valid value is 12, as 10%2 is 0 (False) and 12 < 10 is False.

I wrote this loop to run through my list of examples for all 4 assignments. I used the var_count as a simple loop counter and used the value to identify what assignment the variable was representing.

Question 5, Part B



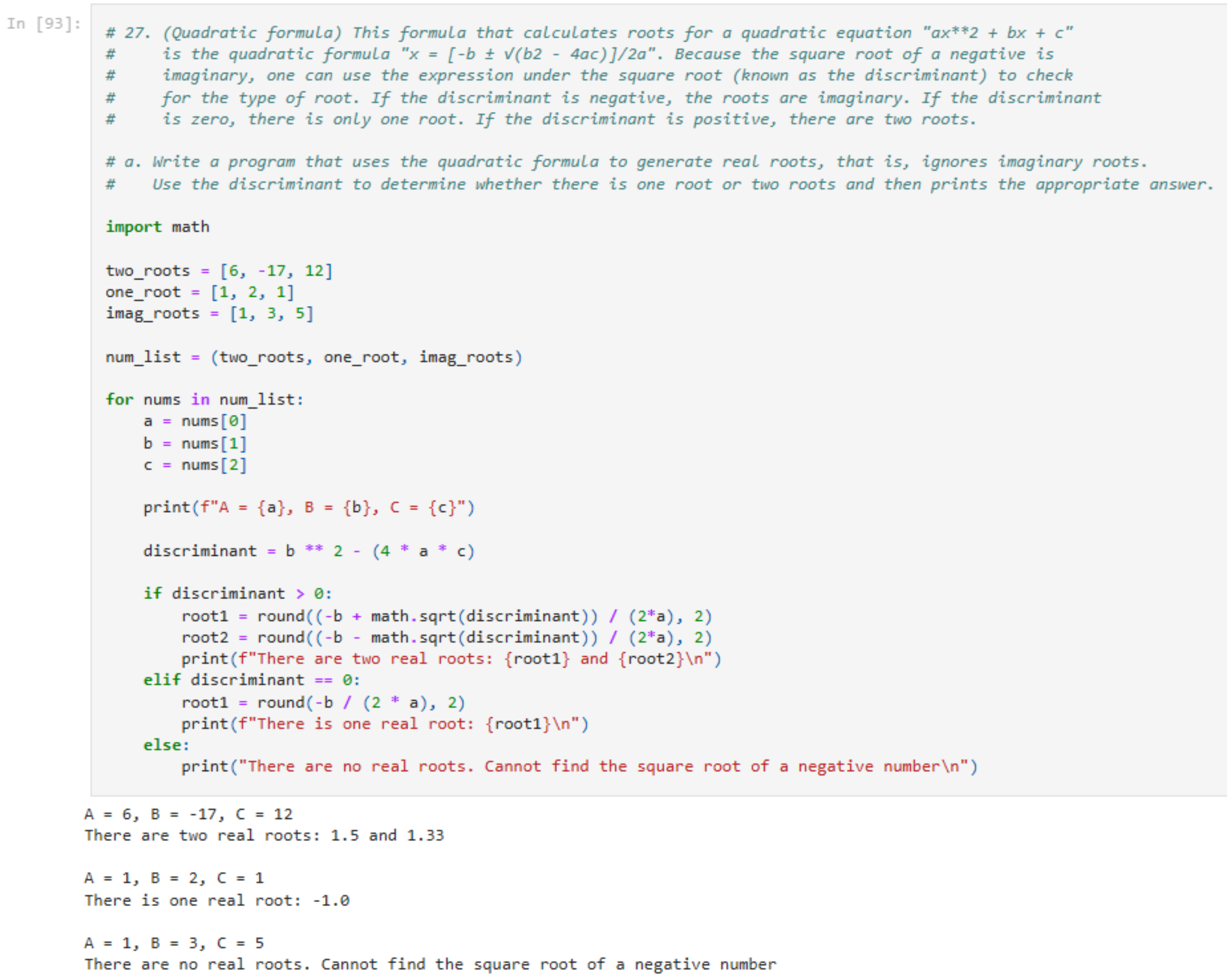
```
In [112…
# b. Find four *ranges* of my_var values that will cause each of the four assignment statements to be executed

assignment_1 = ["my_var += 4", [1, 5, 7, 9, 11]]
assignment_2 = ["my_var /= 1.5", [3]]
assignment_3 = ["my_var *= 2", [2, 4, 6, 8]]
assignment_4 = ["my_var -= 2", [10, 12, 14, 16]]

var_list = (assignment_1, assignment_2, assignment_3, assignment_4)
var_count = 0

for var in var_list:
    var_count = var_count + 1
    print(f"\nAssignment {var_count}: {var[0]}")
    for my_var in var[1]:
        start_var = my_var
        if my_var % 2:
            if my_var ** 3 != 27:
                my_var += 4
            else:
                my_var /= 1.5
        else:
            if my_var < 10:
                my_var *= 2
            else:
                my_var -= 2
        end_var = my_var
        print(f"\tInput: {start_var} -> Output: {end_var}")
```

```
Assignment 1: my_var += 4
        Input: 1 -> Output: 5
        Input: 5 -> Output: 9
        Input: 7 -> Output: 11
        Input: 9 -> Output: 13
        Input: 11 -> Output: 15

Assignment 2: my_var /= 1.5
        Input: 3 -> Output: 2.0

Assignment 3: my_var *= 2
        Input: 2 -> Output: 4
        Input: 4 -> Output: 8
        Input: 6 -> Output: 12
        Input: 8 -> Output: 16

Assignment 4: my_var -= 2
        Input: 10 -> Output: 8
        Input: 12 -> Output: 10
        Input: 14 -> Output: 12
        Input: 16 -> Output: 14
```

For assignment 1, the range is any odd number other than 3 (ex. 1, 5, 6, 9, 11, etc.). For assignment 2, the only number that satisfies this condition is 3. It is the only number that can be cubed and equal 27. For assignment 3, any even number less than 10 (ex. 2, 4, 6, 8) is valid. For assignment 4, any even number greater than (and including) 10 will satisfy the conditions (ex. 10, 12, 14, 16, etc.).

I took my loop a step further in this step and added a nested loop to support the nested lists I assigned to the assignment variables. I did this so I could run through all listed values for each assignment list. I also created *start_var* and *end_var* to track the value of *my_var* before and after the assignment. Technically, *end_var* is not necessary since it will be the same value as *my_var*, but I wanted to keep the naming scheme. While I included much more than necessary in this problem, I was eager to put the indexing, slicing, and loop control concepts we learned into practice and have a nicely presented output.

Question 27, Part A

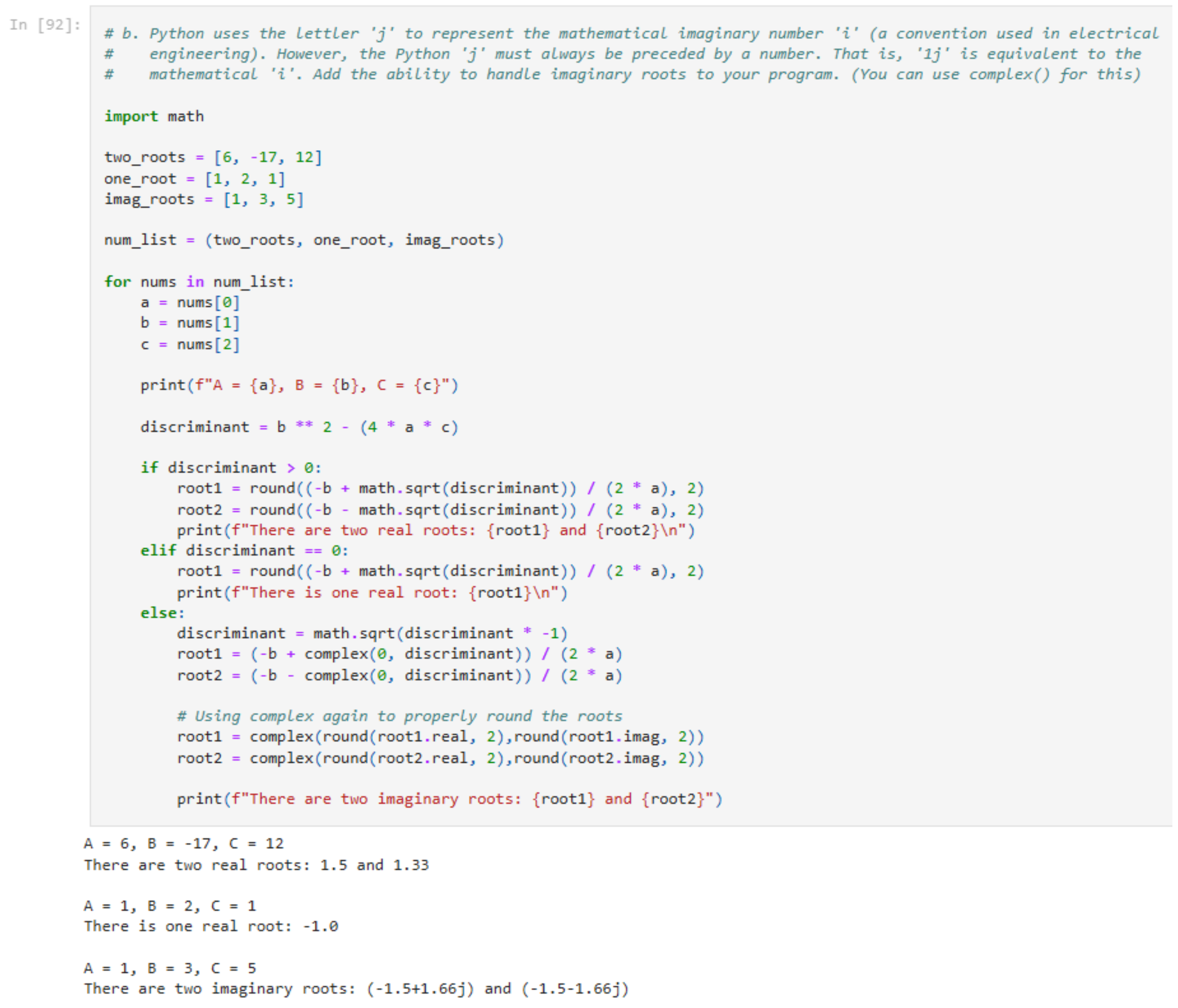


```
In [93]:  # 27. (Quadratic formula) This formula that calculates roots for a quadratic equation "ax**2 + bx + c"
          #     is the quadratic formula "x = [-b ± √(b2 - 4ac)]/2a". Because the square root of a negative is
          #     imaginary, one can use the expression under the square root (known as the discriminant) to check
          #     for the type of root. If the discriminant is negative, the roots are imaginary. If the discriminant
          #     is zero, there is only one root. If the discriminant is positive, there are two roots.

          # a. Write a program that uses the quadratic formula to generate real roots, that is, ignores imaginary roots.
          #    Use the discriminant to determine whether there is one root or two roots and then prints the appropriate answer.

          import math

          two_roots = [6, -17, 12]
          one_root = [1, 2, 1]
          imag_roots = [1, 3, 5]

          num_list = (two_roots, one_root, imag_roots)

          for nums in num_list:
              a = nums[0]
              b = nums[1]
              c = nums[2]

              print(f"A = {a}, B = {b}, C = {c}")

              discriminant = b ** 2 - (4 * a * c)

              if discriminant > 0:
                  root1 = round((-b + math.sqrt(discriminant)) / (2*a), 2)
                  root2 = round((-b - math.sqrt(discriminant)) / (2*a), 2)
                  print(f"There are two real roots: {root1} and {root2}\n")
              elif discriminant == 0:
                  root1 = round(-b / (2 * a), 2)
                  print(f"There is one real root: {root1}\n")
              else:
                  print("There are no real roots. Cannot find the square root of a negative number\n")
```

```
A = 6, B = -17, C = 12
There are two real roots: 1.5 and 1.33

A = 1, B = 2, C = 1
There is one real root: -1.0

A = 1, B = 3, C = 5
There are no real roots. Cannot find the square root of a negative number
```

Similar to the previous solutions, I utilized for loops to run through a list of numbers that would produce specific roots to demonstrate how my code works. I imported the math module and used the square root function to get the square root of the discriminant, which is the number that determines the number of roots.

Question 27, Part B

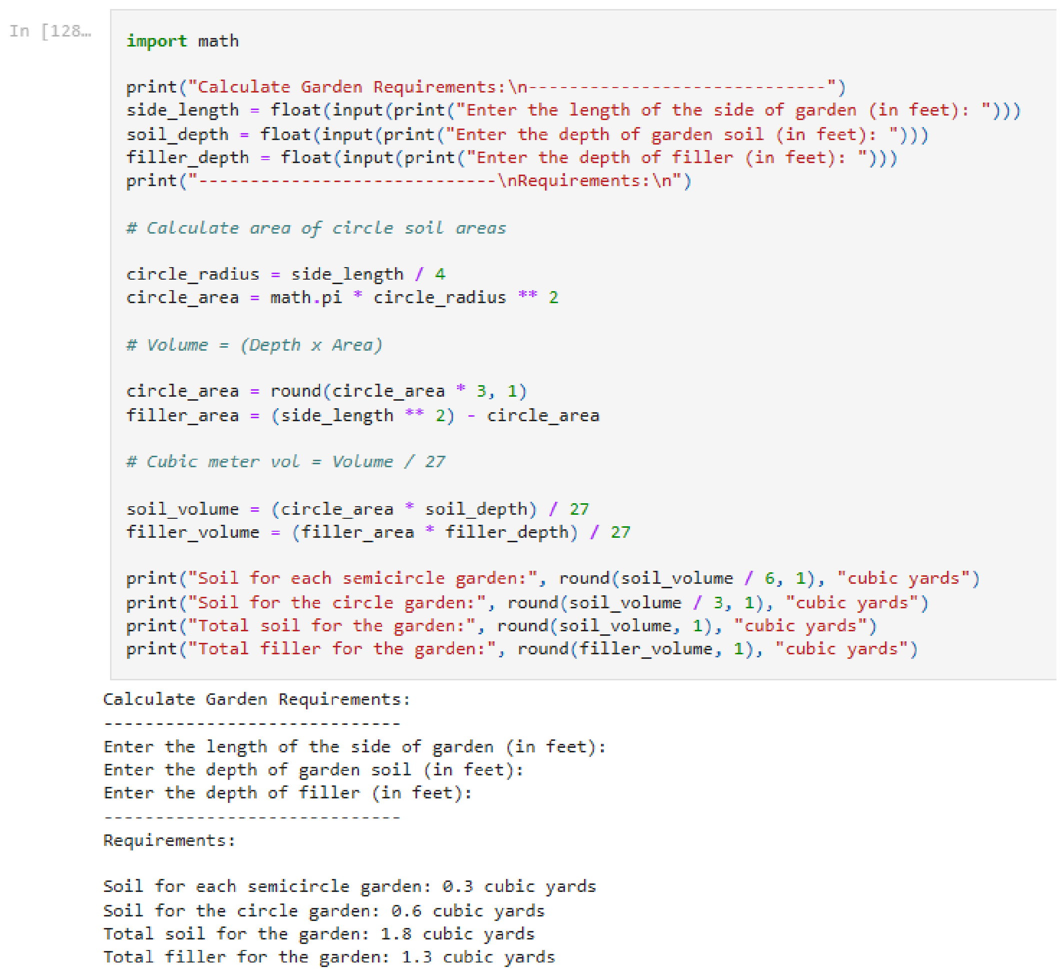


```
In [92]:
# b. Python uses the lettler 'j' to represent the mathematical imaginary number 'i' (a convention used in electrical
#    engineering). However, the Python 'j' must always be preceded by a number. That is, '1j' is equivalent to the
#    mathematical 'i'. Add the ability to handle imaginary roots to your program. (You can use complex() for this)

import math

two_roots = [6, -17, 12]
one_root = [1, 2, 1]
imag_roots = [1, 3, 5]

num_list = (two_roots, one_root, imag_roots)

for nums in num_list:
    a = nums[0]
    b = nums[1]
    c = nums[2]

    print(f"A = {a}, B = {b}, C = {c}")

    discriminant = b ** 2 - (4 * a * c)

    if discriminant > 0:
        root1 = round((-b + math.sqrt(discriminant)) / (2 * a), 2)
        root2 = round((-b - math.sqrt(discriminant)) / (2 * a), 2)
        print(f"There are two real roots: {root1} and {root2}\n")
    elif discriminant == 0:
        root1 = round((-b + math.sqrt(discriminant)) / (2 * a), 2)
        print(f"There is one real root: {root1}\n")
    else:
        discriminant = math.sqrt(discriminant * -1)
        root1 = (-b + complex(0, discriminant)) / (2 * a)
        root2 = (-b - complex(0, discriminant)) / (2 * a)

        # Using complex again to properly round the roots
        root1 = complex(round(root1.real, 2),round(root1.imag, 2))
        root2 = complex(round(root2.real, 2),round(root2.imag, 2))

        print(f"There are two imaginary roots: {root1} and {root2}")
```

```
A = 6, B = -17, C = 12
There are two real roots: 1.5 and 1.33

A = 1, B = 2, C = 1
There is one real root: -1.0

A = 1, B = 3, C = 5
There are two imaginary roots: (-1.5+1.66j) and (-1.5-1.66j)
```

When re-writing the else statement to represent imaginary numbers, I first multiplied the discriminant by -1 to make it a positive integer (since the root of a negative number is imaginary and results in an error). Then I used the complex() function to produce the imaginary number ending in 'j'. Then, to properly round the roots, I used complex again and used the .real and .imag methods to define the real and imaginary numbers in both roots.

4. Flower Garden | Programming Project



In [128…

```python
import math

print("Calculate Garden Requirements:\n-----------------------------")
side_length = float(input(print("Enter the length of the side of garden (in feet): ")))
soil_depth = float(input(print("Enter the depth of garden soil (in feet): ")))
filler_depth = float(input(print("Enter the depth of filler (in feet): ")))
print("-----------------------------\nRequirements:\n")

# Calculate area of circle soil areas

circle_radius = side_length / 4
circle_area = math.pi * circle_radius ** 2

# Volume = (Depth x Area)

circle_area = round(circle_area * 3, 1)
filler_area = (side_length ** 2) - circle_area

# Cubic meter vol = Volume / 27

soil_volume = (circle_area * soil_depth) / 27
filler_volume = (filler_area * filler_depth) / 27

print("Soil for each semicircle garden:", round(soil_volume / 6, 1), "cubic yards")
print("Soil for the circle garden:", round(soil_volume / 3, 1), "cubic yards")
print("Total soil for the garden:", round(soil_volume, 1), "cubic yards")
print("Total filler for the garden:", round(filler_volume, 1), "cubic yards")
```

```
Calculate Garden Requirements:
-----------------------------
Enter the length of the side of garden (in feet):
Enter the depth of garden soil (in feet):
Enter the depth of filler (in feet):
-----------------------------
Requirements:

Soil for each semicircle garden: 0.3 cubic yards
Soil for the circle garden: 0.6 cubic yards
Total soil for the garden: 1.8 cubic yards
Total filler for the garden: 1.3 cubic yards
```

In this problem, I did not end up using the int() function, but I did use the float(), round(), print(), and input() functions in calculating the garden requirements.

My first step was to calculate the area of the circle/semicircle areas. When reviewing the information for the assignment, I noticed that the side length is equal to the length of one perfect circle and two semicircles–meaning the radius is a quarter of the side length. I imported the math module to use the pi variable to get the area of the circles. I then calculated the volume of the circle area and filler area, then converted both of those values to cubic yards

I used the total soil as the basis to calculate the soil for each circle and semicircle. With one perfect circle and four semicircles, I determined there are 3 whole circles / 6 semicircles, so I divided the total soil by those figures to get the soil needed for each circle and semicircle. I also rounded all of the calculations to 1 decimal place. With all of that done, the program succeeds with the example test case.