CS521 - Homework 4 - Alexander Bean

1. List Comprehension with Multiple Conditions

In [40]:
```
list = [num for num in range(1, 1001) if num % 3 == 0 or str(num)[len(str(num))-1::] == '7']

print(list)
```

```
[3, 6, 7, 9, 12, 15, 17, 18, 21, 24, 27, 30, 33, 36, 37, 39, 42, 45, 47, 48, 51, 54, 57, 60, 63, 66, 67, 69, 72, 75, 77, 78,
81, 84, 87, 90, 93, 96, 97, 99, 102, 105, 107, 108, 111, 114, 117, 120, 123, 126, 127, 129, 132, 135, 137, 138, 141, 144, 14
7, 150, 153, 156, 157, 159, 162, 165, 167, 168, 171, 174, 177, 180, 183, 186, 187, 189, 192, 195, 197, 198, 201, 204, 207, 21
0, 213, 216, 217, 219, 222, 225, 227, 228, 231, 234, 237, 240, 243, 246, 247, 249, 252, 255, 257, 258, 261, 264, 267, 270, 27
3, 276, 277, 279, 282, 285, 287, 288, 291, 294, 297, 300, 303, 306, 307, 309, 312, 315, 317, 318, 321, 324, 327, 330, 333, 33
6, 337, 339, 342, 345, 347, 348, 351, 354, 357, 360, 363, 366, 367, 369, 372, 375, 377, 378, 381, 384, 387, 390, 393, 396, 39
7, 399, 402, 405, 407, 408, 411, 414, 417, 420, 423, 426, 427, 429, 432, 435, 437, 438, 441, 444, 447, 450, 453, 456, 457, 45
9, 462, 465, 467, 468, 471, 474, 477, 480, 483, 486, 487, 489, 492, 495, 497, 498, 501, 504, 507, 510, 513, 516, 517, 519, 52
2, 525, 527, 528, 531, 534, 537, 540, 543, 546, 547, 549, 552, 555, 557, 558, 561, 564, 567, 570, 573, 576, 577, 579, 582, 58
5, 587, 588, 591, 594, 597, 600, 603, 606, 607, 609, 612, 615, 617, 618, 621, 624, 627, 630, 633, 636, 637, 639, 642, 645, 64
7, 648, 651, 654, 657, 660, 663, 666, 667, 669, 672, 675, 677, 678, 681, 684, 687, 690, 693, 696, 697, 699, 702, 705, 707, 70
8, 711, 714, 717, 720, 723, 726, 727, 729, 732, 735, 737, 738, 741, 744, 747, 750, 753, 756, 757, 759, 762, 765, 767, 768, 77
1, 774, 777, 780, 783, 786, 787, 789, 792, 795, 797, 798, 801, 804, 807, 810, 813, 816, 817, 819, 822, 825, 827, 828, 831, 83
4, 837, 840, 843, 846, 847, 849, 852, 855, 857, 858, 861, 864, 867, 870, 873, 876, 877, 879, 882, 885, 887, 888, 891, 894, 89
7, 900, 903, 906, 907, 909, 912, 915, 917, 918, 921, 924, 927, 930, 933, 936, 937, 939, 942, 945, 947, 948, 951, 954, 957, 96
0, 963, 966, 967, 969, 972, 975, 977, 978, 981, 984, 987, 990, 993, 996, 997, 999]
```

To achieve this, I used the range() function to check all numbers from 1 to 1000. I identified the max as 1001 because range() checks up to but not including the max.

I used an if statement with two conditions bound by the “or” operator. For the first condition: If the remainder of number % 3 == 0, it means it is evenly divisible by 3. If this is true, the number meets the criteria and is included in the list.

To check the second criteria (ending in 7), I turned the number into a string so I could index/slice it. With indexing, I start the index from the length of the string - 1, which will always return the last digit of the string. If the last character is 7, then it meets the criteria and is included in the list.

The resulting list contains all numbers evenly divisible by 3 or ending in number 7.

2. Nested List Comprehension

In [41]:
```
matrix = [ [1, 2, 3], [4, 5, 6], [7, 8, 9], [10, 11, 12] ]

transposed_matrix = [[list[index] for list in matrix] for index in range(len(matrix[0]))]

print("Original:\t", matrix,
      "\nTransposed:\t", transposed_matrix)
```

```
Original:       [[1, 2, 3], [4, 5, 6], [7, 8, 9], [10, 11, 12]]
Transposed:     [[1, 4, 7, 10], [2, 5, 8, 11], [3, 6, 9, 12]]
```

In the previous homeworks, I used enumerate() to identify indices in the matrix. However, I could not get it to work as effectively with nested list comprehension. When looking through class notes for a solution, I found that I can use range() and len() to identify the indices instead.

In this scenario, len() will identify the length of the list (which is 3). Range() will then take the length and use that as the range maximum. I can use the range numbers (0, 1, and 2) as indices to identify numbers in the matrix list and to insert numbers into the correct sublist in the transposed matrix list.

By adding the first "for" statement in brackets, I identify that these items will be grouped in a list based on the expression. This way, items of the same sublist index will be placed in the correct sublists in transposed_matrix. This produces the expected result.

3. Nested List Comprehension (Cont.)

In [42]:
```
matrix = [ [1, 2, 3], [4, 5, 6], [7, 8, 9] ]

flattened_matrix = [list[index] for list in matrix for index in range(len(matrix[0]))]

print("Original:\t", matrix,
      "\nFlattened:\t", flattened_matrix)
```

```
Original:       [[1, 2, 3], [4, 5, 6], [7, 8, 9]]
Flattened:      [1, 2, 3, 4, 5, 6, 7, 8, 9]
```

This code is 99% the same as the previous solution. By removing the brackets on the first "for" statement, the resulting list items are not placed into a list when iterated. Instead, they are processed and appended right into flattened_matrix. By processing it this way, all of the items are placed in the same list and effectively flattens the matrix.

4. Dictionary Comprehension

In [43]:
```
string = "supercalifragilisticexpialidocious"

dict = {char:string.count(char) for char in string}

print("Input:\t", string,
      "\nOutput:\t", dict)
```

```
Input:   supercalifragilisticexpialidocious
Output:  {'s': 3, 'u': 2, 'p': 2, 'e': 2, 'r': 2, 'c': 3, 'a': 3, 'l': 3, 'i': 7, 'f': 1, 'g': 1, 't': 1, 'x': 1, 'd': 1,
'o': 2}
```

In the expression, I identified the key (character) and value (count of character) to be assigned from the string. With dictionary comprehension, the dictionary is created within the statement so there is no need to define “dict = {}” beforehand.

In the print statement, I listed the input and output. The output orders the characters based on order of appearance and lists the number of occurrences within the string.

5. Search Algorithms

In [640…

```
from random import randint

# Creating random array and sort values ascending

array = [randint(0, 500) for num in range(1, 1001)]
array.sort()

# print("Array:\t", array, "\n")

# Assigning variables for the algorithm calculations & result

target_num = 27
min = 0
max = len(array) - 1

alg_result = (f"The number {target_num} cannot be found in the list.")

# Algorithm

while (min <= max):
    search_interval = (min + max) // 2
    if array[search_interval] == target_num:
        max = search_interval - 1
        alg_result = (f"The number {array[search_interval]} can be found in the list, located at index {search_interval}.")
    elif array[search_interval] > target_num:
        max = search_interval - 1
    else:
        min = search_interval + 1

print("Result:\t", alg_result)

# print(array[search_interval - 1])
# print(array[search_interval + 1])
```

```
Result:  The number 27 can be found in the list, located at index 54.
```

I used list comprehension to create a randomized list and used the sort() method to arrange them in ascending order. Then I assigned variables for ease of coding and readability. I created the alg_result variable to track the algorithm result. It's initialized with a "negative" result, and, if the target number is found within the array, the value will be reassigned with a "positive" result listing the index of the value.

The algorithm is based in a while loop, where the minimum search index is less than or equal to the maximum search index. This establishes a range within the array to check for 27. Following the question instructions, I halved the interval using truncated division to produce an indexable integer.

The if-elif-else statement is based on whether the value of the halfway index is equal to, greater than, or less to (implied in else) the target index. If the halfway value is greater than the target, the maximum is reduced to the halfway value index - 1. If it's less, the minimum is increased to the halfway value index + 1. Each iteration will identify a new range and which side (left or right) to check until the target is found.

I noticed that, sometimes, the middle instance would be listed instead of the first occurrence. To resolve this, I added the index maximum reassignment to the "if" statement. By adding this, the program will keep subtracting through the indices until the first instance is found. If it goes too far below, the else statement will add to the interval until, ultimately, the first occurrence is found. I also commented two print statements to check the neighboring indexes and to validate my program.