EIT, 306387, Gabriel Ćwiek, MAPT 2, lab 9, https://github.com/Astaree/MAPT_2

Ex.1:

```
color_list = ["Red", "Green", "Blue"]
print(color_list[0])
mixed_list = ["Apple", 5, True, 3.14]
print(mixed_list)
num_items = len(color_list)
print(num_items)
item = "Green"
print(item in color_list)
zeros_list = [0] * 100
print(zeros_list)
digit_names = ["One", "Two", "Three", "Four", "Five"]
combined_list = color_list + digit_names
print(combined_list)
combined_list[0] = "Earth"
combined_list[-1] = ["Monday", "Tuesday", "Wednesday", "Thursday", "Friday",
"Saturday", "Sunday"]
print(combined_list)
combined_list.append("Mercury")
print(combined_list)
```

Ex. 2 - Tuples are immutable, so you cannot append items to them:

```
my_tuple = ("red", "green", "red")
first_element = my_tuple[0]
print("First element:", first_element)
tuple_length = len(my_tuple)
print("Tuple length:", tuple_length)
red_count = my_tuple.count("red")
print("Number of 'red' occurrences:", red_count)
red_index = my_tuple.index("red")
print("Index of 'red':", red_index)
# Tuples are immutable, so you cannot append items to them
```

Ex. 3:

a. L1 = [1, 2, 3, 5] L2 = L1 Here, a list `L1` is created with elements [1, 2, 3, 5]. Then, `L2` is assigned the reference to `L1`, which means both `L1` and `L2` are pointing to the same list.
b. print(L2) Output: [1, 2, 3, 5] This prints the contents of `L2`, which is the same as `L1` since they reference the same list.
c. print(id(L1)) Output: <memory address 1> This prints the memory address of `L1` using the `id()` function.
d. print(id(L2)) Output: <memory address 1> This prints the memory address of `L2`, which is the same as `L1` since they reference the same list.
e. L2[2:] = [0, 0, 0, 0, 0, 0] This modifies the list referenced by `L2`. It replaces elements from index 2 onwards with [0, 0, 0, 0, 0, 0].

f. print(id(L2)) Output: <memory address 1> The memory address of `L2` remains the same after the modification because the modification was done in place, and `L2` still refers to the same list as `L1`.
g. print(L1) Output: [1, 2, 0, 0, 0, 0, 0, 0] This prints the contents of `L1` after the modification. Since `L1` and `L2` reference the same list, the changes made to `L2` are reflected in `L1`.
h. print(L2) Output: [1, 2, 0, 0, 0, 0, 0, 0] This prints the contents of `L2` after the modification. As mentioned earlier, `L2` refers to the same list as `L1`, so it reflects the changes made.
i. L1 = [1, 2, 3, 5] L2 = L1 Here, a new list `[1, 2, 3, 5]` is assigned to `L1`, and `L2` is assigned the reference to `L1`.
j. print(id(L1)) Output: <memory address 2> This prints the memory address of the new `L1` list.
k. print(id(L2)) Output: <memory address 2> This prints the memory address of `L2`, which is the same as the new `L1` list because they reference the same list.
l. L2 = [0, 0, 0, 0, 0] Here, `L2` is assigned a new list `[0, 0, 0, 0, 0]`, which creates a separate list object.
m. print(L1) Output: [1, 2, 3, 5] This prints the contents of the new `L1` list, which remains unchanged.
n. print(L2) Output: [0, 0, 0, 0, 0] This prints the contents of the new `L2` list.
o. print(id(L2)) Output: <memory address 3> This prints the memory address of the new `L2` list, which is different from the memory address of `L1` and the previous `L2` list.
p. Conc - assigning a list to a new variable does not create a copy of the list. Instead, it creates a new reference to the same list object. To create a copy of a list, you can use the `copy()` method or the slicing operation `[:]`.

Ex. 4:

a. T1 = (1, 2, 3) Here, a tuple T1 is created with elements (1, 2, 3).
b. print('T2 = T1') This simply prints the string 'T2 = T1'.
c. T2 = T1 T2 is assigned the reference to the same tuple object as T1. Since tuples are immutable, T2 and T1 will always refer to the same tuple object.
d. print('addresses of the T1 and T2:') This prints the string 'addresses of the T1 and T2:'.
e. print(id(T1)) Output: <memory address 1> This prints the memory address of T1 using the id() function.
f. print(id(T2), '\n') Output: <memory address 1> This prints the memory address of T2, which is the same as T1 since they reference the same tuple object.
g. L1 = [1, 2, 3, 5] Here, a list L1 is created with elements [1, 2, 3, 5].
h. T1 = (1, 2, 3) T1 is reassigned to a new tuple object (1, 2, 3). This creates a new tuple object with a different memory address.
i. print('address of the L1:') This prints the string 'address of the L1:'.
j. print(id(L1), '\n') Output: <memory address 2> This prints the memory address of L1, which is different from the memory address of the previous tuple objects.
k. print('address of the T1:') This prints the string 'address of the T1:'.
l. print(id(T1), '\n') Output: <memory address 3> This prints the memory address of the new T1 tuple object, which is different from the memory address of the previous tuple objects.
m. print('T2 = L1') This prints the string 'T2 = L1'.

n. T2 = L1 T2 is assigned the reference to the list object L1. Unlike tuples, lists are mutable, so changes made to T2 will also affect L1.
o. print('address of the T2') This prints the string 'address of the T2'.
p. print(id(T2), '\n') Output: <memory address 2> This prints the memory address of T2, which is the same as L1 since they reference the same list object.

Ex 5:

```
class squareDict:
    def __init__(self, n):
        self.square = {i: i*i for i in range(1, n+1)}

    def printDict(self):
        for key, value in self.square.items():
            print(f"Key: {key}, Value: {value}, Type: {type(value)}")

if __name__ == "__main__":
    n = 5
    obj = squareDict(n)
    obj.printDict()

Out:
Key: 1, Value: 1, Type: <class 'int'>
Key: 2, Value: 4, Type: <class 'int'>
Key: 3, Value: 9, Type: <class 'int'>
Key: 4, Value: 16, Type: <class 'int'>
Key: 5, Value: 25, Type: <class 'int'>
```

Ex. 6:

```
def findDiv(n:int,m:int,div:int, ndiv:int):
    res = []
    for i in range(n,m+1):
        if i%div==0 and i%ndiv!=0:
            res.append(i)
    return res

if __name__ == "__main__":
    n = 2000
    m = 3200
    div = 7
    ndiv = 5
    r = findDiv(n,m,div,ndiv)
    print(r)
    print(f"Found: {len(r)}")

Out (skipped some results):

2506, 2513, 2527, 2534, 2541, 2548, 2562, 2569, 2576, 2583, 2597, 2604, 2611,
2618, 2632, 2639, 2646, 2653, 2667, 2674, 2681, 2688, 2702, 2709, 2716, 2723,
2737, 2744, 2751, 2758, 2772, 2779, 2786, 2793, 2807, 2814, 2821, 2828, 2842,
2849, 2856, 2863, 2877, 2884, 2891, 2898, 2912, 2919, 2926, 2933, 2947, 2954,
2961, 2968, 2982, 2989, 2996, 3003, 3017, 3024, 3031, 3038, 3052, 3059, 3066,
3073, 3087, 3094, 3101, 3108, 3122, 3129, 3136, 3143, 3157, 3164, 3171, 3178,
3192, 3199]
Found: 138
```

Ex. 7 – conc - assigning a new value to a parameter variable does not affect the variable passed as an argument, while modifying a list parameter variable can change the original list object. Making a copy of the list using the `copy()` method creates a separate list object that can be modified independently.:

```
Out:
var_inside myfun1:2
var_outside:1
listA_outside before:[0, 1, 2, 3, 4, 5, 6, 7]
list_inside myfun2:[0, 1, 0, 3, 4, 5, 6, 7]
listA_outside after:[0, 1, 0, 3, 4, 5, 6, 7]
listB_outside after:[0, 1, 0, 3, 4, 5, 6, 7]
list_inside myfun3:[0, 1, 0, 0, 4, 5, 6, 7]
listA_outside after:[0, 1, 0, 0, 4, 5, 6, 7]
listC_outside after:[0, 1, 0, 3, 4, 5, 6, 7]
```

Ex 8:

```
# Exercise 8_1: Temperature Conversion
temperature = input("Enter the temperature (e.g., 21C or 70F): ")
# Extract the numerical value and scale from the input
value = float(temperature[:-1]) # Extract all but the last character
scale = temperature[-1] # Extract the last character
converted_temperature = None
if scale == 'C':
    converted_temperature = (value * 9/5) + 32
    converted_scale = 'F'
elif scale == 'F':
    converted_temperature = (value - 32) * 5/9
    converted_scale = 'C'
if converted_temperature is not None:
    print(f"The converted temperature is:
{converted_temperature}{converted_scale}")
# Exercise 8_2: Season Name
day_number = int(input("Enter the day number: "))
month_number = int(input("Enter the month number: "))
season = None
if (month_number == 1 and day_number >= 1) or (month_number == 2 and
day_number <= 28):
    season = "Winter"
elif (month_number == 3 and day_number >= 1) or (month_number == 4 and
day_number <= 30):
    season = "Spring"
elif (month_number == 5 and day_number >= 1) or (month_number == 6 and
day_number <= 30):
    season = "Summer"
elif (month_number == 7 and day_number >= 1) or (month_number == 8 and
day_number <= 31):
    season = "Autumn"
elif (month_number == 9 and day_number >= 1) or (month_number == 10 and
day_number <= 31):
    season = "Autumn"
elif (month_number == 11 and day_number >= 1) or (month_number == 12 and
day_number <= 31):
    season = "Winter"
if season is not None:
    print(f"The season for the given date is: {season}")
out:
Enter the temperature (e.g., 21C or 70F): 21C
The input temperature is: 21C, type: <class 'str'>
The converted temperature is: 69.8F
Enter the day number: 12
Enter the month number: 5
The season for the given date is: Summer
```