Functions

Solve the following exercises and upload your solutions to Moodle until the specified due date.

Important Information!

Please try to *exactly match the output* given in the examples (naturally, the input can be different). We are running automated tests to aid in the correction and grading process, and deviations from the specified output lead to a significant organizational overhead, which we cannot handle in the majority of the cases due to the high number of submissions.

Make sure to use the *exact filenames* that are specified for each individual exercise. Also, use the provided unit tests to check your scripts before submission (see the slides Handing in Assignments on Moodle).

It is of *particular importance* in this assignment to wrap the printing that you see in the example outputs in `if __name__=='__main__:'`, as your exercises essentially only consist of a function definition. Example - let's say the task is to write a function that doubles the float value that is passed:

```
def double(var: float) -> float:
    return var*2

if __name__ == '__main__':
    print(double(3.4))
```

Unless explicitly stated otherwise, you can assume correct user input and correct arguments. You are *not allowed* to use any concepts and modules that have not yet been presented in the lecture.

Feel free to copy the example text from the assignment sheet, and then change it according to the exercise task.

Exercise 1 – Submission: `a4_ex1.py` 25 Points

Write a function `aggregate(*args, **kwargs) -> dict` that counts the frequency of each keyword. The function can take the following arguments:

- An arbitrary number of keywords as positional arguments (args). The frequencies of these arguments needs to be counted.
- An arbitrary number of keyword-frequency pairs as keyword arguments (kwargs). The frequency that is passed here needs to be added to the frequency count of the positional arguments.

The output of the function is a dictionary where the keys are the unique input keywords (both args and kwargs), and the values are the counted frequencies (occurrences of positional arguments added to the values of corresponding keyword arguments - if there are any).

Below are some examples:

```
print(aggregate())
{}

print(aggregate(a=5, b=10, d=4))
{'a': 5, 'b': 10, 'd': 4}

print(aggregate('a', 'b', 'c', 'b', 'c'))
{'a': 1, 'b': 2, 'c': 2}

print(aggregate('a', 'b', 'c', 'b', 'c', a=5, b=10, d=4))
{'a': 6, 'b': 12, 'd': 4, 'c': 2}
```

Exercise 2 – Submission: `a4_ex2.py` 25 Points

Write a function `compose(*functions)` that returns a composed function from an arbitrary number of functions passed to the function `compose()` as positional arguments. For simplicity, assume that the input functions each take only one parameter, e.g. if the input functions $f(x)$, $g(x)$, $h(x)$ are passed to the function as `compose(f, g, h)`, the returned function should evaluate to $f(g(h(x)))$. Note: do not use decorators to solve this exercise.

The parameter that is passed to the composed function can be any object in principle. Naturally it depends on what the individual functions do. Let's assume for example that we have the following three functions:

```
def add_one(x):
    return x + 1

def square(x):
    return x * x

def half(x):
    return x / 2
```

The `compose()` function can now be used as follows. If `composed_f()` does not contain any functions it should just return the argument that is passed. Otherwise the result of the nested function call has to be returned. Note the evaluation order: here $f(x)$ is `halve()`, $g(x)$ is `square()` and $h(x)$ is `add_one()`. Which means that $f(g(h(3)))$ results in the value 8.0.

```
composed_f = compose()
result = composed_f(3)                          # result = 3

composed_f = compose(square)
result = composed_f(3)                          # result = 9

composed_f = compose(halve, square, add_one)
result = composed_f(3)                          # result = 8.0
```

Exercise 3 – Submission: `a4_ex3.py` 25 Points

Write a function `multiply_matrix(matrix1: list, matrix2: list) -> list` that multiplies two input matrices, `matrix1` and `matrix2`, and returns the resulting matrix. The matrices are represented as 2D nested lists. Assume that the input matrices are valid, meaning that the matrix values are numbers and all rows (inner lists) have the same length. The function should check if the two matrices are compatible for multiplication. If they are not, the function returns `None`. Below are some examples:

```
matrix1 = [[1, 2, 3], [4, 5, 6]]                     # matrix size 2x3
matrix2 = [[7, 8], [9, 10]]                          # matrix size 2x2
result_matrix = multiply_matrix(matrix1, matrix2)
print(result_matrix)                                 # result: None

matrix1 = [[1, 2, 3], [4, 5, 6]]                     # matrix size 2x3
matrix2 = [[7, 8], [9, 10], [11, 12]]                # matrix size 3x2
result_matrix = multiply_matrix(matrix1, matrix2)
print(result_matrix)                                 # result: [[58, 64], [139, 154]]
```

Exercise 4 – Submission: `a4_ex4.py` 25 Points

Write a function `check_parentheses(s: str) -> bool` to check if the input string `s`, which contains a sequence of opening parentheses '(' and closing parentheses ')', is valid. A string `s` is considered valid if it satisfies the following conditions:

- Every opening parenthesis pairs with a corresponding closing parenthesis.
- No closing parenthesis appears before its corresponding opening parenthesis.
- There can be spaces in the string (as you can see in the examples).

Below are some examples:

```
print(check_parentheses(""))                          # result: True

print(check_parentheses("("))                         # result: False

print(check_parentheses(")"))                         # result: False

print(check_parentheses(")("))                        # result: False

print(check_parentheses("()"))                        # result: True

print(check_parentheses("( (()) ()()) )"))            # result: False

print(check_parentheses("( (()) (()()) )"))           # result: True

print(check_parentheses("( (()) (()()) ) ("))         # result: False

print(check_parentheses("( (()) (()()) ) ()"))        # result: True
```