Recursion and Generators

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: `a6_ex1.py` 25 Points

Write a generator function `numerical_sequence()` that can return an infinite sequence of numbers n_i which are determined as $n_i = \sum_1^i i$, $i \geq 1$.

Example code execution:

```
sequence = numerical_sequence()
for i in range(1, 11):
    print(f'number at {i}: {next(sequence)}')
```

Expected result:

```
number at 1: 1
number at 2: 3
number at 3: 6
number at 4: 10
number at 5: 15
number at 6: 21
number at 7: 28
number at 8: 36
number at 9: 45
number at 10: 55
```

Exercise 2 – Submission: `a6_ex2.py` 25 Points

Using recursion, write a function `power_set(L: list) -> set[tuple]` that generates and returns all possible subsets of an input set `L`, passed as a list. Each subset in the result is represented as a tuple of elements in ascending order (the subsets; not the overall set). Assume that `L` is valid, meaning it is not `None` and its elements are distinct and not `None`.

Example code execution:

```
print(power_set([]))

print(power_set([3, 1, 2]))

print(power_set(['c', 'a', 'b']))
```

Expected output:

```
result: {()}

result: {(1, 3), (2,), (1, 2), (1, 2, 3), (2, 3), (1,), (), (3,)}

result: {('a', 'b', 'c'), ('c',), ('a', 'c'),
         ('b', 'c'), ('a',), ('b',), (), ('a', 'b')}
```

Hints: note that the `power_set` function is not necessarily recursive itself, but it can call another function that handles the recursion. Refer to Table 1 for the recursion principle of the power set generation.

Input set	Power set
[]	{()}
[**a**]	{()}+ {(**a**)} = {(), (a)}
[a, **b**]	{(), (a)}+ {(**b**), (a, **b**)} = {(), (a), (b), (a, b)}
[a, b, **c**]	{(), (a), (b), (a, b)}+ {(**c**), (a, **c**), (b, **c**), (a, b, **c**)} = {(), (a), (b), (a, b), (c), (a, c), (b, c), (a, b, c)}

Table 1: Recursion principle for the power set generation.

Exercise 3 – Submission: `a6_ex3.py` 25 Points

Using recursion, write a function `flatten_dict(d: dict) -> dict` that flattens an input (nested) dictionary `d` and returns the flattened dictionary. Keys in the flattened dictionary are constructed by concatenating relevant keys form the input dictionary, using '.' character. Assume that the input dictionary is not `None`.

Example code execution:

```
d1 = {}
d2 = {'a':1, 'b':2}
d3 = {
    'a':1, 'b':2,
    'c':{
        'a': 3,
        'b': {
            'a': 4,
            'b': 5
        }
    },
    'd': 6
}

print(flatten_dict(d1))

print(flatten_dict(d2))

print(flatten_dict(d3))
```

Expected output:

```
result: {}

result: {'a': 1, 'b': 2}

result: {'a': 1, 'b': 2, 'c.a': 3, 'c.b.a': 4, 'c.b.b': 5, 'd': 6}
```

Note that the `flatten_dict` function is not necessarily recursive itself, but it can call another function that handles the recursion.

Exercise 4 – Submission: `a6_ex4.py` 25 Points

Using recursion, write a function `permute(s: str) -> set[str]` that generates and returns a set of all permutations of characters from the input string `s`. A permutation is represented as a string of the same length as `s`. Assume that `s` is not `None`.

Example code execution:

```
print(permute(''))

print(permute('a'))

print(permute('ab'))

print(permute('abc'))
```

Expected output:

```
result: {''}

result: {'a'}

result: {'ba', 'ab'}

result: {'cba', 'abc', 'bac', 'acb', 'cab', 'bca'}
```

Hints: Refer to Table 2 for the recursion principle of the permutation generation.

Input strings	Permutations
”	{”}
’**a**’	”⟶ {’**a**’}
’a**b**’	’a’⟶ {’**b**a’, ’a**b**’}
’ab**c**’	’ba’⟶ {’**c**ba’, ’b**c**a’, ’ba**c**’} ’ab’⟶ {’**c**ab’, ’a**c**b’, ’ab**c**’}

Table 2: Recursion principle for the permutation generation.