Problem Set 9 - Functions and recursions

Hello student,

the goal of this problem-set is to work with functions, and tip your toes in the use of recursions.

Recursion are a powerful tool to solve a complex problem using a simple algorithm.

Assignment 9.1 - Recursive Pi (3 points)

In assignment 5.2 you were asked to develop a program capable of approximate Π using iterations. In this exercise you are asked to do the same, but using recursion.

Ask the user for the number of iterations and calculate its value.

The output should look like this:

```
Insert the number of iterations: 1
After 1 iterations Pi: 4
```

```
Insert the number of iterations: 42
After 42 iterations Pi: 3.117786501758878
```

Hint: The formula to approximate π is: $\pi = 4 - \frac{4}{3} + \frac{4}{5} - \frac{4}{7} + \frac{4}{9} - \frac{4}{11} + ...$

Hint 2: Remember if you get a `RecursionError: maximum recursion depth exceeded in comparison` you can increase (resonably) your limit with:

```
import sys
sys.setrecursionlimit(10000)
```

Assignment 9.2 - Fees calculator (3 points)

You are a tax officer at the Swiss Post and you are still calculating taxes by hand. However, you are a smart tax officer, and you want to create a small program that will help you in your calculations.

You have the approval from your boss Ruth Olga Bucher, but she is interested and prepared a main function for you. Good girl!

However, she does not know how to do the calculation. She was able to find the explanations on the Swiss Post's website where you read:

> The customs clearance fee depends on the origin of the consignment and the value of the goods sent. For consignments from Germany, France, Austria and Italy, the basic fee is CHF 11.50 while for consignments from any other country, the basic fee is CHF 16.00. A surcharge corresponding to 3
>
> Examples:
>
> A parcel from France with a value of CHF 100.00:
>
> Basic fee: CHF 11.50 Surcharge: CHF 3.00 (3% of the declared value of the goods) Customs clearance fee: CHF 14.50
>
> A parcel from the US with a value of CHF 300.00:
>
> Basic fee: CHF 16.00 Surcharge: CHF 9.00 (3% of the declared value of the goods) Customs clearance fee: CHF 25.00

Some orders have a special value multiplier to be used instead of the normal 3%. You have freedom to write the function to calculate the code, however the main is defined by your boss, and is the following:

```
orders = [
    {'order_id': 594526, 'origin': 'Austria', 'value': 185, 'multiplier': 1},
    {'order_id': 559006, 'origin': 'France', 'value': 313, 'multiplier': 1},
    {'order_id': 828446, 'origin': 'France', 'value': 299, 'multiplier': 14},
    {'order_id': 855966, 'origin': 'Moldavia', 'value': 109, 'multiplier': 1},
    {'order_id': 673246, 'origin': 'Russia', 'value': 252, 'multiplier': 16},
    {'order_id': 364126, 'origin': 'Austria', 'value': 452, 'multiplier': 13},
    {'order_id': 224286, 'origin': 'Austria', 'value': 242, 'multiplier': 1},
    {'order_id': 312926, 'origin': 'Moldavia', 'value': 481, 'multiplier': 1},
    {'order_id': 613406, 'origin': 'China', 'value': 353, 'multiplier': 4},
    {'order_id': 143486, 'origin': 'Russia', 'value': 373, 'multiplier': 1},
    {'order_id': 439326, 'origin': 'Spain', 'value': 208, 'multiplier': 1},
    {'order_id': 983646, 'origin': 'Portugal', 'value': 381, 'multiplier': 1},
    {'order_id': 198686, 'origin': 'USA', 'value': 429, 'multiplier': 17},
    {'order_id': 231326, 'origin': 'Spain', 'value': 243, 'multiplier': 1},
    {'order_id': 755166, 'origin': 'Austria', 'value': 133, 'multiplier': 1},
    {'order_id': 290846, 'origin': 'Spain', 'value': 435, 'multiplier': 1},
    {'order_id': 151006, 'origin': 'Portugal', 'value': 491, 'multiplier': 1},
    {'order_id': 109246, 'origin': 'Italy', 'value': 114, 'multiplier': 1},
    {'order_id': 901726, 'origin': 'Portugal', 'value': 398, 'multiplier': 1},
    {'order_id': 420446, 'origin': 'Germany', 'value': 498, 'multiplier': 5},
]

if __name__ == "__main__":
    for order in orders:
        if order["multiplier"] != 1:
            order["fees"] = calculate_fees(origin=order["origin"], value=order[
                "value"], taxes=order["multiplier"])
            continue
        order["fees"] = calculate_fees(origin=order["origin"], value=order["
            value"])

    pretty_print(orders)
```

The result will be the following (example with first 5 elements):

```
order 594526, fees: 17.05
order 559006, fees: 20.89
order 828446, fees: 53.36
order 855966, fees: 19.27
order 673246, fees: 56.32
...
```

Remember to round results to 2 digits!

Assignment 9.3 - Fibonacci - and the importance of caching (optional)

The Fibonacci numbers is a numeric sequence described as:

$$F_n = F_{n-1} + F_{n-2}$$

Starting with $F_1 = F_2 = 1$, and $F_0 = 0$.

Write a `fibonacci` function that calculates the n^{th} element in the sequence using recursion.

For example, using this main:

```
def fibonacci(n):
    ... # your implementation

if __name__ == "__main__":

    for i in range(5):
        print(fibonacci(i))
```

The result obtained is:

```
0
1
1
2
3
```

Now try to measure the execution time of your function using the following code:

```
import time


def fibonacci(n):
    ... # your implementation

def measure_execution_time(algorithm, number):
    tic = time.perf_counter()
    algorithm(number)
    toc = time.perf_counter()

    print(f"Element calculated in {toc - tic:0.5f} seconds")

if __name__ == "__main__":
    measure_execution_time(fibonacci, 20)
```

Now try to develop a second version of the algorithm. It works just the same but you use a dictionary as a cache.

If a number is not in the dictionary, you calculate it and store the value in the dictionary using the position (n) as key.

Obtaining something similar to:

```python
import time


def measure_execution_time(algorithm, number):
    tic = time.perf_counter()
    algorithm(number)
    toc = time.perf_counter()

    print(f"Element calculated in {toc - tic:0.5f} seconds")


def fibonacci(n):
    ... # your implementation


cache = {}


def fast_fibonacci(n):
    if n in cache.keys():
        return cache[n]

    ... # something

    cache[n] = ... # something else
    return cache[n]

if __name__ == "__main__":
    measure_execution_time(fibonacci, 20)
    measure_execution_time(fast_fibonacci, 200)
```

How do the two compare to each other?