Exercise 01: AdvProg

1) Create a program that asks the user for two numbers and then prints their sum.
   Hint. Use the function input. Recall that the function input returns a str.

2) Create a program that adds a user specified number of numbers. The program should first ask the user for a number *k*, and then *k* times ask the user for a number, and finally print the sum of the *k* numbers.
   Hint. Use a while-loop.

3) Write a program that determines if two intervals [a, b] and [c, d] overlap, where a ≤ b and c ≤ d. The program should ask the user for the four end-points a, b, c and d of the two intervals, and print if the two intervals overlap or not. If the two intervals overlap, the program should print the interval where the two input intervals overlap.
   Hint. Use one or more if-statements.
   Examples:

```
[2, 4] and [6, 9] do not overlap
[5, 7] and [1, 3] do not overlap
[4, 8] and [-6, 2] do not overlap
[2, 5] and [3, 9] overlap in the interval [3, 5]
[3, 6] and [1, 3] overlap in the interval [3, 3]
[1, 8] and [2, 6] overlap in the interval [2, 6]
```

4) Consider the following program for finding the integer square root of a non-negative integer:

```
x = int(input('x: '))

low = 0
high = x + 1

while True:  # low <= sqrt(x) < high
    if low + 1 == high:
        break

    mid = (high + low) // 2

    if mid * mid <= x:
        low = mid
        continue

    high = mid

print(low)  # low = floor(sqrt(x))
```

   a) Walk through and understand the program.
   b) Try to simplify the program, getting rid of the break and continue statements.

5) Write a naive program that tests if a small number $x \geq 2$ is a prime number.
Recall that a number is a prime number if and only if the only numbers dividing the number is 1 and the number itself, like the numbers 2, 3, 5, 7, 11, 13...
The program should ask the user for the number *x* and print whether the number is a prime or not. If *x* is not a prime, it should print a divisor/factor in the range [2, *x* - 1].
Hint. Try all relevant factors. Use the % operator to compute the remainder by integer division, e.g. 13 % 5 == 3.

6) Write a program that, given a positive integer *x*, prints the prime factors of that number. Recall that any positive number has a unique prime factorization, e.g. 126 = 2 * 3 * 3 * 7.

The program should ask the user for the integer *x*, and then print each prime factor. Note that for *x* = 126, the prime factor 3 should be printed two times.

Hint. Repeatedly divide *x* by 2 until 2 is not a prime factor any longer. Repeat the same for 3, 4, 5,... Note that 4 = 2 * 2 is not a prime number. But since its prime factors cannot be divisors of *x* divided by the prime factors found so far, then we will not find 4 (and any other composite number) as a prime divisor of *x*.

7) Use the Newton-Raphson method to compute the square root of a float $n \geq 1.0$, by finding a root of the function $f(x) = x^2 - n$ and letting your initial approximation of the root be $x = n$. Recall $f'(x) = 2 * x$. In your computations you can use +, -, *, and /.

You can compare your result to the result of the builtin sqrt function in the math module using:

```
import math
print(math.sqrt(n))
```

8) Stirling's approximation states that *n*! can be approximated by $S(n) = (2\,\pi\, n)^{0.5} \cdot (n\,/\,e)^n$.

Write a program that asks the user for an *n* and prints the ratio *n*! / *S*(*n*), i.e. how good an approximation *S*(*n*) is to *n*!.

Some example outputs:

```
1! / S(1) = 1.08443755
10! / S(10) = 1.00836536
100! / S(100) = 1.00083368
1000! / S(1000) = 1.00008334
10000! / S(10000) = 1.00000833
100000! / S(100000) = 1.00000083
```

Note: Use import math to get access to the constants math.pi and math.e.

Warning: Computing the ratio for larger *n* can be tricky.

This exercise is inspired by a problem from Kattis.