MAT101 Programming – Homework 5

Deadline: Monday, 6.11.2022, 22:00

Login to `https://w3.math.uzh.ch/my/` with your UZH credentials to submit your solved exercises for grading. You can find more information on how to upload/submit your exercises on `https://wiki.math.uzh.ch/public/studentUpload`.

For submission, please upload at most 1 python file per exercise, except for exercise 5. You could even just upload 2 python files for the whole exercise sheet.
In the case you're not asked to write code, you can either write a comment or `print` those messages in the python file which includes the code for that exercise, or upload them as .pdf or similar file type.

Exercise 1. **15** P.

The *digital root* of a natural number $n \geq 1$ is the (single digit) value obtained by a recursive sum over all the digits in the number n. The process continues until a single-digit number is reached. Let us look at a few examples:

- $16 \to 1+6=7$ hence the digital root of 16 is 7.
- $942 \to 9+4+2=15 \to 1+5=6$ hence the digital root of 942 is 6.
- $132189 \to 1+3+2+1+8+9=24 \to 2+4=6$ hence the digital root of 132189 is 6.
- $493193 \to 4+9+3+1+9+3=29 \to 2+9=11 \to 1+1=2$ hence the digital root of 493193 is 2.

a) Write a function `digital_root(n)` that takes as input an integer $n \geq 1$ and `returns` the digital root of said integer. **10** P.

b) Create a new script in which you `import` "digital_root". Write a few examples/tests in this new script. Based on these examples do you find some kind of pattern between a number and its digital root. **5** P.

Exercise 2. **17** P.

a) Write a function `quadratic_formula(a, b, c)` which computes the solutions to the equation

$$ax^2 + bx + c = 0$$

and `returns` them in a `list`.
`quadratic_formula` should be able to `return` complex solutions, i.e. it should be able to solve the equation $x^2 + 1 = 0$ which has the solutions i and $-i$.

Also, check whether the input is valid: a, b, c should be numbers (int, float, complex) and a should not equal zero so that we have a proper quadratic equation. If that is not the case, the function should return `None`. **15** P.

b) In the case that you want to solve many equations which have a constant coefficient of zero, i.e. $c = 0$, how could you modify your function to account for that? Could you do the same if the linear coefficient is often zero, i.e. $b = 0$, but you needed to change c most of the times? **2** P.

Note: Python can handle complex numbers without needing to import anything, i.e. `(-1)**(0.5)` gives out `(6.123233995736766e-17+1j)`, which looks like a complex number just instead of i python uses `j` as the imaginary unit, and the real part ends with *e-17*, i.e. is of the order 10^{-17} so its practically 0, (it's not 0 due to machine precision).
The exact output from `(-1)**(0.5)` might vary depending on the machine used and the python version, but should be very similar.

Exercise 3. **20** P.

As you have probably seen multiple times while comparing your implementations with your colleagues' or with the provided sample solutions, there are often many ways to implement a function even if the functionality is the exact same.
So how would you go about choosing an implementation if you needed it for a larger project?
One simple way is timing your code and taking the one that is the fastest.

To see a simple example of this, you're going to implement a few ways to sum the elements from 1 to N, where N is a (reasonably large) positive integer and compare the times they need to execute.

a) define the function `while_sum(N)` which takes as input a positive `int` and sums the numbers from 1 to N using a `while` loop and `returns` the result. **4** P.

b) define the function `for_sum(N)` which takes as input a positive `int` and sums the numbers from 1 to N using a `for` loop and `returns` the result. **4** P.

c) define the function `gauss_summation(N)` which takes as input a positive `int` and computes the sum from 1 to N using the explicit formula

$$\sum_{k=0}^{N} k = \frac{N \cdot (N+1)}{2}$$

and `returns` the result. **5** P.

d) now use the module `time` more specifically `time.perf_counter_ns` to measure how long your implementations need to calculate the sum given $N = 10000$. **5** P.

e) comment on your findings in d) and comparing with how long `sum(range(1, N+1))` needs to compute the same sum, argue in which scenario you would choose which function. **2** P.

Note: `time.perf_counter_ns` can be used in the same way as `time.time` which you have seen in the lecture. The main difference is, that `time.perf_counter_ns` returns the time measurement in nanoseconds whereas `time.time` uses seconds and has lower precision.

Exercise 4. **15** P.

a) Define the function `my_primefactors` which takes as input a positive `int` and returns the primefactors of that number as a list. **10** P.

b) Another way of finding the factorization is to use sympy. Install it as you did for instance for numpy and add 'from sympy import primefactors' in the beginning of your script. Compute the prime factors using both your algorithm and the imported one. Do you notice a difference in speed for large numbers? How do you think could your algorithm get more efficient? **5** P.

Exercise 5. **13** P.

a) Write a function `reverse(array)` which takes a `list` as input and `returns` a *new* list containing the same elements as "array" but in reverse order. **5** P.

b) Write a function `reverse_inplace(array)` which takes a `list` as input, reverses the order of the elements of "array" but `returns None`. **6** P.

c) For both a) and b): in the case that "array" is not a `list`, the function should `print` a message stating that the input is invalid and `return None`. **2** P.

Note: if you write in your script:

```
array1 = [1, 2, 3]
rev_array = reverse(array1)
print(rev_array)
```

the output should be `[3, 2, 1]` but crucially

```
print(array1)
```

should still give `[1, 2, 3]`, however if you write

```
array2 = [1, 2, 3, 4]
reverse_inplace(array2)
print(array2)
```

the output should be `[4, 3, 2, 1]`.