Geo1000 – Python Programming – Assignment 2

Due: Friday September 25, 2020 (18h00)

Introduction

You are expected to create 4 programs. All program files have to be handed in. Start with the files that are distributed via Brightspace. It is sufficient to modify the function definitions inside these files (replace `pass` with your own implementation) — **do not change the function signatures, i.e. their names, which & how many and in which order the functions take arguments**.

This assignment is *preferably* made in groups of 2 (enroll with your group or individually in Brightspace), and the mark you will obtain will count for your final grade of the course. Helping each other is fine. However, make sure that your implementation is your *own*.

This assignment in total can give you 100 points. Your assignment will be marked based on whether your implementation does the correct things (as described in the assignment), whether your code is decent (e.g. use of proper variable names, indentation, etc), and whether your files are submitted as required (e.g. on time).

It is due: **Friday September 25, 2020 (18h00)**.

Note that if you submit your assignment after the deadline, some points will be removed. For the first day that a submission is late, 10 points will be removed before marking. For every day after that, another 20 points will be removed. An example: Assume you deliver the assignment at Friday September 25, 2020, 18h15, the maximum amount of points that then can be obtained for the assignment is 90 $(100 - 10 = 90)$.

Submit the resulting program files (`cab.py`, `rds.py`, `sudoku.py`) via Brightspace (your last submission will be taken into account, also for determining whether you are late). Upload **a zip file** that contains just the program files (with no folders/hierarchy inside)! Also, please do **not change** the file names.

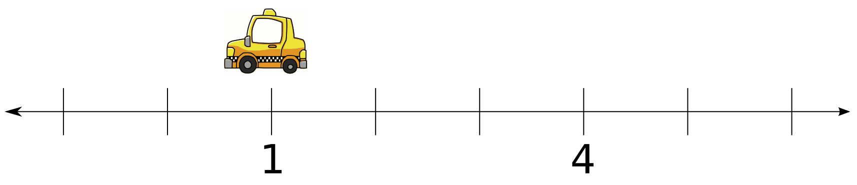
Make sure that each Python file handed in starts with the following comment (augment `Authors` and `Studentnumbers` with your own names and numbers):

```
# GEO1000 - Assignment 2
# Authors:
# Studentnumbers:
```

1 Taxi! – cab.py – (30 points)

A taxi moves along a 1D axis with integer numbers:

The taxi should deliver a person from a start point to an end point. The taxi is only allowed to make a fixed number of moves (per move the taxi goes 1 step left or 1 step right). All moves have to be used, and the taxi should arrive exactly at the end point.



Make a program that computes how many different paths exist to go from the start to the end. For this, write a *recursive* and fruitful function `wiggle` that returns the number of possible paths for the taxi as integer.

An example:
The taxi wishes to take 5 moves to travel from location 1 to location 4; then, there are 5 possible paths. The function `wiggle` in this case returns `5`.

Start from the following skeleton:

```
# GEO1000 - Assignment 2
# Authors:
# Studentnumbers:


def wiggle(start, end, moves):
    pass


if __name__ == "__main__":
    print("running cab.py directly")
    print(wiggle(1, 4, 5))
```

Do not use any imports.

Note, if you are confused about the conditional statement in the skeleton code:

```
if __name__ == "__main__":
```

Please watch https://youtu.be/sugvnHA7ElY.

2 Repeated digital sum – rds.py (30 points)

Write a program that calculates the repeated digital sum for a sentence.

The initial integer is generated by taking the sum of every letter's position in the alphabet (a=1, b=2, etc).

You can assume that the string will contain only letters (no numbers or characters with accents). However, the sentence may contain spaces and punctuation marks like comma, dot, and exclamation mark (`,.!`); these characters have to be skipped during generating the initial integer.

For example, the initial integer of string *`Geomatics is fun!`* is

$161 = (7 + 5 + 15 + 13 + 1 + 20 + 9 + 3 + 19 + 9 + 19 + 6 + 21 + 14)$

The repeated digital sum of a non-negative integer is the (single digit) value obtained by an iterative process of summing digits, on each iteration using the result from the previous iteration to compute a digit sum. The process continues until a single-digit number is reached.

For example, the repeated digital sum of $65,536$ is 7 because $6 + 5 + 5 + 3 + 6 = 25$ and $2 + 5 = 7$.

Make sure you look at the Python documentation, more specifically the constants in § 6.1.1 (https://docs.python.org/3/library/string.html). Note that it might be wise to convert the input first to all lower case letters.

Start from the following skeleton:

```
# GEO1000 - Assignment 2
# Authors:
# Studentnumbers:

import string


def sentence_value(sentence):
    pass


def rds(value):
    pass


if __name__ == "__main__":
    initial_integer = sentence_value("Geomatics is fun!")
    result = rds(initial_integer)
    print(result)
```

Do *not* use any other imports than given. Also, you are *not* allowed to make use of the `ord` function in your implementation.

3 Sudoku checking – sudoku.py (40 points)

Write a program that can check whether a 4x4 sudoku is filled in correctly.

1	2	3	4
3	4	1	2
2	1	4	3
4	3	2	1

Figure 1: Example of a valid 4x4 sudoku

A valid 4x4 sudoku obeys the following rules:

- Every number from 1 upto and including 4 occurs in every row and every column exactly once
- The 4 square blocks containing 4 numbers each also contain every number from 1 upto and including 4 exactly once

Input to the program will be given as a string representing the sudoku, e.g. `1234341221434321` represents the sudoku given in Figure 1.

Start from the following skeleton:

```
# GEO1000 - Assignment 2
# Authors:
# Studentnumbers:


def parse(sudoku):
    pass
```

```
def is_valid(stucture):
    pass


def main():
    pass


if __name__ == "__main__":
    main()
```

Do not use any imports.

Make the program interactive. Write a main function that asks a user for input, by presenting the following line:

```
    >>> Your sudoku for validation, or "quit" to exit.
```

It prints to the user after validating the Sudoku:

```
    This sudoku is *VALID*
```

or

```
    This sudoku is *INVALID*
```

Then the user can choose to enter another Sudoku for checking. To end the program, the user can type `quit`.

If the user enters a string that can not represent a 4x4 sudoku (e.g. the string has fewer than 16 characters, or not all characters are integers), the program should respond with:

```
    Input not understood
```

and ask for input again.

Note that:

- The `parse` function returns `None` if the string given by the user cannot represent a valid 4x4 sudoku. Make sure you look at the documentation: `https://docs.python.org/3/library/stdtypes.html#str.isdigit`.
- The `parse` function makes a data structure that represents the rows, columns and blocks of numbers in the Sudoku. This data structure will be understood by the `is_valid` function. *Important*, the data structure has to be a list with for every element in the sudoku (every row, every column and every block of numbers) a tuple with the 4 correct digits (as integer) for the element, i.e. 12 tuples with 4 int's each will be in the list returned from `parse`.
- The `is_valid` function uses the data structure returned by the `parse` function to perform the validation and returns a boolean (i.e. `True/False`) indicating whether the sudoku is valid or not. Print statements should thus only be placed inside the `main` function.