Geo1000 – Python Programming – Assignment 2

Due: Monday, October 3, 2022 (18h00)

Introduction

You are expected to create 3 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. keep their names, which & how many arguments the functions take and in which order the functions take arguments as given**.

This assignment is *preferably* made in groups of 2 (enroll with your group or individually in Brightspace), and your mark 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: **Monday, October 3, 2022 (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 Monday, October 3, 2022, 18h15, the maximum amount of points that then can be obtained for the assignment is 90 $(100 - 10 = 90)$.

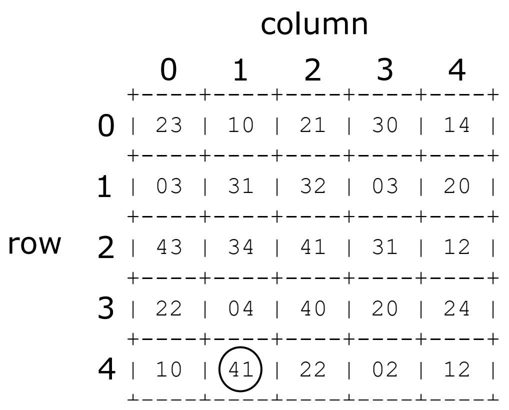
Submit the resulting program files (`hunt.py`, `secret.py`, `tree.py`) via Brightspace (your last submission will be taken into account, also for determining whether you are late). Upload **a zip file** that contains just the program files (with no folders/hierarchy inside)!

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 Treasure hunt – hunt.py (35 points)

In this exercise, you will write a program that finds a hidden treasure in a table. A treasure is hidden in a cell whose own value represents its own coordinates in the table. The table contains clues where the treasure is hidden. Your program must first read in the table data. Then, the program searches for the treasure (note that the search is limited by a maximum number of cells that the function is allowed to visit). Finally, the program should output a message indicating if



		column				
		0	1	2	3	4
	0	23	10	21	30	14
	1	03	31	32	03	20
row	2	43	34	41	31	12
	3	22	04	40	20	24
	4	10	41	22	02	12

Figure 1: A table with a hidden treasure. In this example, you will find the hidden treasure, 41, in 19 steps, where the steps are [(0, 0), (2, 3), (3, 1), (0, 4), (1, 4), (2, 0), (4, 3), (0, 2), (2, 1), (3, 4), (2, 4), (1, 2), (3, 2), (4, 0), (1, 0), (0, 3), (3, 0), (2, 2), (4, 1)].

any treasure was found in the table, and if so, where and how many steps were needed to find it. An example is given in Figure 1, where a treasure is hidden at row 4 column 1.

Each cell contains exactly two integer values. The two integers represent a clue; they are respectively the row number and the column number of the cell containing the next clue. The hunt starts in the upper left corner, at cell (0, 0). The first three clues in Figure 1 are 00, 23, and 31.

You can assume that the cell references are always correct (i.e. they do not point to a location outside the table).

Steps to take:

1. Make a fruitful function `read_grid` that returns a *datastructure* that can be used by the `visit` function. The datastructure has to be a list of lists. Every sublist represents a row in the table and contains for every cell a tuple with (row, col)-indexes, where the indexes are integers.
2. Make an *iterative* fruitful function `visit` that takes `table` (the nested list returned by `read_grid`), `steps_allowed`, and `path` as input. This function appends the cells, i.e. the (row, col)-tuples, it traverses throughout to the `path` list. Note that this works because lists are mutable objects. The parameter `steps_allowed` gives the maximum steps for the search. The function should return a boolean value based on whether or not the treasure was found.
3. Make a function `hunt`. This function makes an empty path list to which a solution is appended. It calls function `visit`; then, it returns a string according to the outcome of function `visit`. The contents of the string should be as follows (with X, Y, and N replaced by the correct values):

   `The treasure was found at row: X, column: Y; it took N steps to find the treasure.`

   Or if no solution was found (within `steps_allowed`):

   `Could not find a treasure (in N steps).`

Start from the following skeleton and do not use any imports.

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

def read_grid(filenm):
    """
    """
    pass
```

```
def visit(table, steps_allowed, path):
    """
    """
    pass


def hunt(filenm, max_steps):
    """
    """
    pass


if __name__ == "__main__":
    print(hunt('finite.txt', 20))
```

2 Encryption – secret.py (25 points)

Make a program that can encrypt/decrypt a string.

This program uses a simple encryption technique. First, the program finds consecutive parts in the input string that consist of letters only (letters are the 26 English letters, and it does not matter whether the letters are uppercase or lowercase). Second, the program reverses these consecutive parts. All other characters (non-letters) should keep their locations the same as in the input string.

For example,

```
Hello, can you find my secret mail@address.com?!
```

Becomes (once encrypted)

```
olleH, nac uoy dnif ym terces liam@sserdda.moc?!
```

Note that if you encrypt an already encrypted string, you should get back the original string.

Steps to take for implementation:

1. Make a function `reverse_part` that takes a list of letters as argument and returns the list with letters reversed. You should not modify the input list (make it a pure function, see also § 16.2 and § 16.3 of the book).
2. Make a function `part_to_str`, which transforms a list of letters to a string and returns this string.
3. Make a function `split_in_parts`. This function should split the string into a list of lists. Every sublist should have either some letters or exactly one non-letter character.
4. Make a function `reverse_relevant_parts` that returns a new list with the relevant sublists reversed (this should be a pure function).
5. Make a function `glue` that transforms the list of sublists back into a new string.
6. Make a function `encrypt` that takes a string as input and returns the encrypted string as output. This function should use only the functions `split_in_parts`, `reverse_relevant_parts`, and `glue`.

Start from the following skeleton:

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

```
def reverse_part(part):
    """Take as input a list
    Returns a new list with elements in input reversed
    Note: Pure function, so should not modify input!

    Example:

        >>> reverse_part(['t', 'h', 'i', 's'])
        ['s', 'i', 'h', 't']
    """
    pass


def part_to_str(part):
    """Take as input a list with letters.
    Returns a new string with letters in the input list

    Example:

        >>> part_to_str(['a', 'b', 'c'])
        "abc"

    """
    pass


def split_in_parts(sentence):
    """Split the string into a list of lists (either containing letters, or just
    one character not part of the alphabet).

    Example:

        >>> split_in_parts("this is.")
        [['t', 'h', 'i', 's'], [' '], ['i', 's'], ['.']]

    """
    pass


def reverse_relevant_parts(parts):
    """Reverse only those sublists consisting of letters

    Input: list of lists, e.g. [['t', 'h', 'i', 's'], [' '], ['i', 's'], ['.']]
    Returns: list with sublists reversed that consist of letters only.
    """
    pass


def glue(parts):
    """Transforms the list of sublists back into a new string

    Returns: string
    """
    pass


def encrypt(sentence):
    """Reverses all consecutive letter parts in a string.

    Input: a string
    Returns: a string
    """
    pass


if __name__ == "__main__":
```

```
paragraph = "toN ylno si ti ysae ot eil htiw spam, ti's laitnesse. oT yartrop
    lufgninaem spihsnoitaler rof a xelpmoc, eerht-lanoisnemid dlrow no a talf
    teehs fo repap ro a oediv neercs, a pam tsum trotsid ytilaer. sA a elacs
    ledom, eht pam tsum esu slobmys taht tsomla syawla era yllanoitroporp hcum
    reggib ro rekciht naht eht serutaef yeht tneserper. oT diova gnidih
    lacitirc noitamrofni ni a gof fo liated, eht pam tsum reffo a evitceles,
    etelpmocni weiv fo ytilaer. erehT's on epacse morf eht cihpargotrac xodarap
    : ot tneserp a lufesu dna lufhturht erutcip, na etarucca pam tsum llet
    etihw seil. --- woH ot eil htiw spam, kraM reinomnoM, 1996."
print(encrypt(paragraph))
assert encrypt(encrypt(paragraph)) == paragraph
```

Do not use any imports.

3 Pythagoras tree – tree.py (40 points)

A Pythagoras tree is a fractal that consists of squares (see https://en.wikipedia.org/wiki/Pythagoras_tree_(fractal)). You may have noticed the wooden Pythagoras tree in the Zuid Serre of our faculty building (see Figure 2).

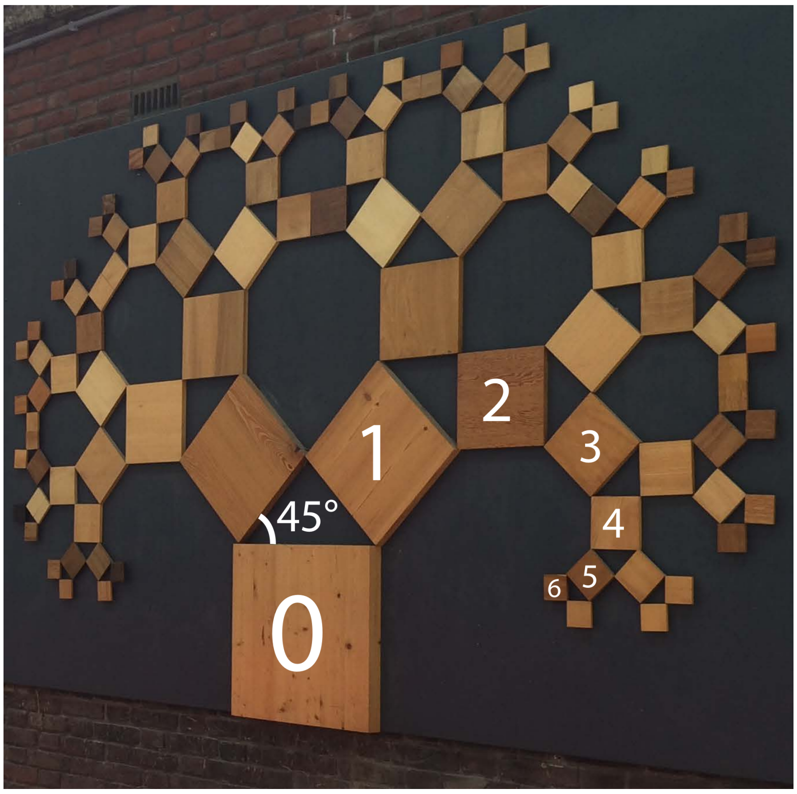


Figure 2: The Pythagoras tree of our faculty.

This particular Pythagoras tree has order 6 and has an angle of 45°. For this assignment, you will write a program that can draw Pythagoras trees of any order and angle. Note that you have to represent a point as a tuple of 2 floats.

Follow these steps:

1. Make a fruitful function `distance` that returns the Cartesian distance between two points in $\mathbb{R}^2$.

2. Make a fruitful function `point_angle_distance`. This function computes point p_2 based on given point p_1 (see Figure 3). The distance from p_1 to p_2 should be d. The angle between the positive x-axis and the line through p_1 and p_2 should be β.

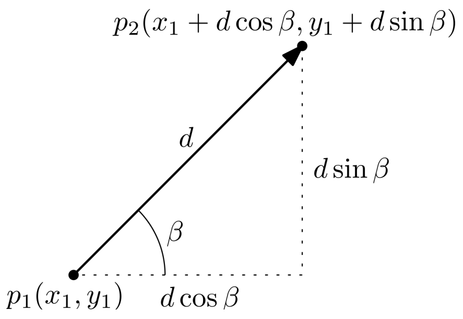


Figure 3: Illustration for function `point_angle_distance`.

3. Make a fruitful function `absolute_angle` that returns the angle (in *radians*) between the positive x-axis and the line through two given points (see angle γ in Figure 4). Hint: Use function `math.atan2()`.

4. Make a fruitful function `opposite_edge`. This function should compute two new points p_3 and p_4 based on two given points p_1 and p_2 such that the four points (i.e., p_1, p_2, p_3, and p_4) form a square (see Figure 4). This function should return a tuple of the two points, i.e., `(`p_3`,` p_4`)`. Hint: Use the `absolute_angle` and `point_angle_distance` functions.

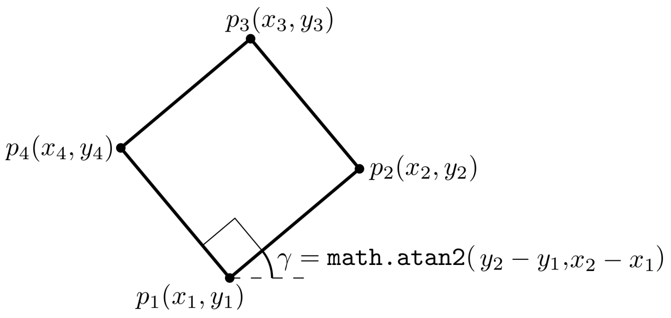


Figure 4: Illustration for function `square`.

5. Make a fruitful function `split_point`. This function should compute *split point* p_3 based on two given points p_1 and p_2 such that the triangle formed by the three points has $\angle p_1 p_3 p_2 = 90°$ and $\angle p_2 p_1 p_3 = \alpha$ (see Figure 5).

6. Make a fruitful function `as_wkt`. This function should return a string of the Well-Known Text (WKT) for four given points (e.g. p_1, p_2, p_3, and p_4). This string should *not* contain any end of line characters.

   Hint: The square p_1, p_2, p_3, p_4 from Figure 4 can be represented in WKT as:

   ```
   POLYGON ((x1 y1, x2 y2, x3 y3, x4 y4, x1 y1))
   ```

   Note the counterclockwise order of points and the repetition of the first point at the end.

7. Now make a *recursive* function `draw_pythagoras_tree` that from given points p_1 and p_2

   - constructs a square p_1, p_2, p_3, p_4 (see Figure 6),
   - writes square p_1, p_2, p_3, p_4 as Well-Known Text (WKT), the current order and its area (in this order) to an output text file with the help of function `as_wkt`,

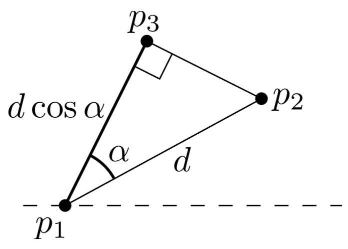


Figure 5: Illustration for function `split_point`.

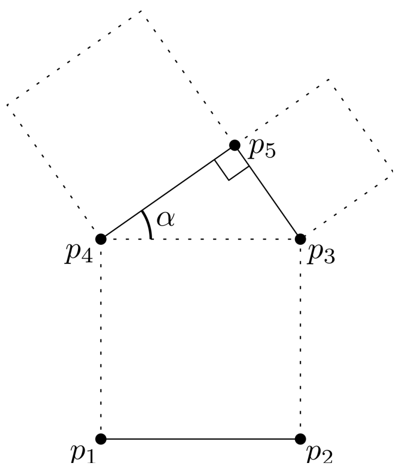


Figure 6: Illustration for function `pythagoras_tree`.

- computes split point p_5 (see Figure 6) based on points p_3, p_4 and angle α,
- applies recursion to the edges incident to the split point.

Note that:

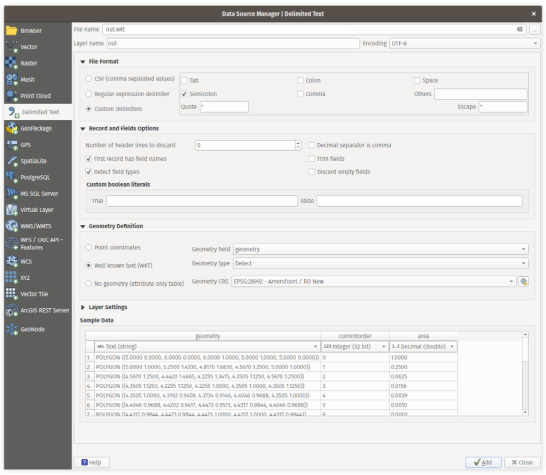
- Use parameter `totalorder` to control the depth of the recursion. When `totalorder=0`, one square should be written to the output file.
- Both the coordinates of the points and the area should appear in the output file as a floating point number with exactly 4 digits behind the comma.
- Use a semi-colon (`;`) as delimiter between the WKT geometry, the current order and the area value.
- Each record (a square together with its order and its area) uses exactly one line in the output file.
- After writing the text file, you can open it with PyCharm (or another text editor, like Notepad++) to see the contents. To get a graphic representation of the squares, you can view the WKT file in QGIS (http://www.qgis.org/) as a Delimited Text Layer (make sure you set the options for reading the WKT file correctly, see Figure 7).

Start from the following skeleton:

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

import math

def distance(p1, p2):
```


Data Source Manager | Delimited Text

File name: out.wkt
Layer name: out
Encoding: UTF-8

File Format

- ( ) CSV (comma separated values)
- ( ) Regular expression delimiter
- (●) Custom delimiters

☐ Tab ☐ Colon ☐ Space
☑ Semicolon ☐ Comma Others:
Quote: " Escape: "

Record and Fields Options

Number of header lines to discard: 0
☐ Decimal separator is comma
☑ First record has field names
☐ Trim fields
☑ Detect field types
☐ Discard empty fields

Custom boolean literals: True: False:

Geometry Definition

- ( ) Point coordinates
- (●) Well known text (WKT)
- ( ) No geometry (attribute only table)

Geometry field: geometry
Geometry type: Detect
Geometry CRS: EPSG:28992 - Amersfoort / RD New

Layer Settings

Sample Data

	geometry	currentorder	area
	Text (string)	Integer (32 bit)	Decimal (double)
1	POLYGON ((5.0000 0.0000, 6.0000 0.0000, 6.0000 1.0000, 5.0000 1.0000, 5.0000 0.0000))	0	1.0000
2	POLYGON ((5.0000 1.0000, 5.2500 1.4330, 4.8170 1.6830, 4.5670 1.2500, 5.0000 1.0000))	1	0.2500
3	POLYGON ((4.5670 1.2500, 4.4420 1.4665, 4.2255 1.3415, 4.3505 1.1250, 4.5670 1.2500))	2	0.0625
4	POLYGON ((4.3505 1.1250, 4.2255 1.1250, 4.2255 1.0000, 4.3505 1.0000, 4.3505 1.1250))	3	0.0156
5	POLYGON ((4.3505 1.0000, 4.3192 0.9459, 4.3734 0.9146, 4.4046 0.9688, 4.3505 1.0000))	4	0.0039
6	POLYGON ((4.4046 0.9688, 4.4202 0.9417, 4.4473 0.9573, 4.4317 0.9844, 4.4046 0.9688))	5	0.0010
7	POLYGON ((4.4317 0.9844, 4.4473 0.9844, 4.4473 1.0000, 4.4317 1.0000, 4.4317 0.9844))	6	0.0002

Figure 7: Delimited Text Layer settings. In QGIS, under the menu ”Layer” → ”Add Layer” → ”Add Delimited Text Layer...”.

```
    """Returns Cartesian distance (as float) between two 2D points"""
    pass


def point_angle_distance(pt, beta, distance):
    """Compute new point that is distance away from pt in direction beta"""
    pass


def absolute_angle(p1, p2):
    """Returns the angle (in radians) between the positive x-axis
    and the line through two given points, p1 and p2"""
    pass


def opposite_edge(p1, p2):
    """Compute and return the edge (as a tuple of two points p3 and p4)
    parallel to the edge defined by the two given points
    p1 and p2 (i.e. the opposite edge in the square).
    """
    pass


def split_point(p1, p2, alpha):
    """Returns the point above this top edge that defines
    the two new boxes (together with points p1 and p2 of the top edge).
    """
    pass


def as_wkt(p1, p2, p3, p4):
    """Returns Well Known Text string (POLYGON) for 4 points
    defining the square
    """
    pass


def draw_pythagoras_tree(p1, p2, alpha, currentorder, totalorder, filename):
    pass


if __name__ == "__main__":
    with open('out.wkt', 'w') as fh:  # 'with' statement closes
```

```
                                           # file automatically
        fh.write("geometry;currentorder;area\n")
    draw_pythagoras_tree(p1=(5,0),
        p2=(6,0),
        alpha=math.radians(45),
        currentorder=0,
        totalorder=6,
        filename='out.wkt')
```

Do not use any other imports than `import math`.