Geo1000 – Python Programming – Assignment 2

Due: Monday, September 29, 2025 (18h00)

Contents

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, in which order the functions take arguments and what they are supposed to return 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, September 29, 2025 (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, September 29, 2025, 18h15, the maximum amount of points that then can be obtained for the assignment is 90 $(100 - 10 = 90)$.

Submit the resulting program files (`robot.py`, `distance.py`, `dms.py`, `query.py`, `patterns.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 Robot – robot.py – (30 points)

A robot moves along a 1D axis with integer numbers (Figure 1). The robot should deliver a good from a start point to an end point. The robot is only allowed to make a fixed number of total moves (per move the robot goes either 1 step left or 1 step right). All moves have to be used, and the robot should arrive exactly at the end point.

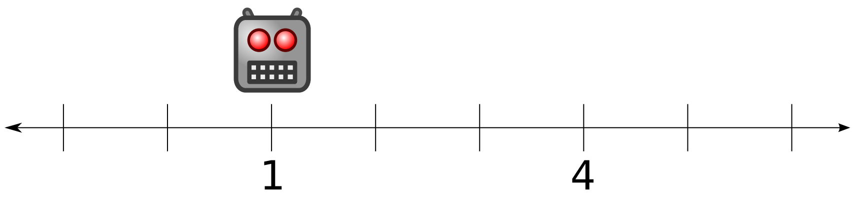

Figure 1: Robot moving along a 1D axis

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 `move` that returns the number of possible paths (as integer) for the robot.

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

Start from the following skeleton:

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


def move(start, end, moves):
    pass


if __name__ == "__main__":
    print("running robot.py directly")
    print(move(1, 4, 5))
    print(move(4, 1, 5))
    print(move(1, 4, 2))
```

Do not use any imports. Also, do *not* use the `global` keyword in your implementation.

2 Distance between 2 places – query.py, nominatim.py, dms.py, distance.py – (35 points)

Write a program that queries the internet for coordinates of two places, formats the location as required and calculates the distance between the two points.

The steps you need to take:

In module `dms.py`: Make the functions that format a geographical coordinate, given in decimal degrees, as degrees, minutes and seconds. The geographical coordinate is given as 2-tuple of floats: (latitude, longitude), i.e. (ϕ, λ). Figure 2 illustrates latitude and longitude.

Given the following `_test` function:

```
def _test():
    coordinates = ((0.0, 0.0),
            (52.0, 4.3287),
```

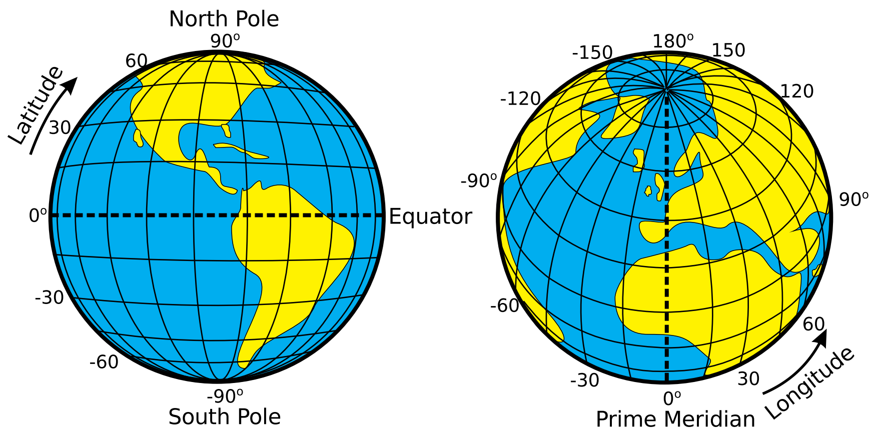

Figure 2: Latitude and Longitude illustrated (image taken from http://www.wikimedia.org)

```
            (-52.0, 4.3287),
            (52.0, -4.3287),
            (-52.0, -4.3287),
            (45.0, 180.0),
            (-45.0, -180.0),
            (-50.4567, 4.3287))
    for coordinate in coordinates:
        print(format_dd_as_dms(coordinate))
```

Make sure that the output will be exactly as follows (including the exact amount of spaces in between, and that there are no additional characters after the last " on a line):

```
N   0°  0'  0.0000", E   0°  0'  0.0000"
N  52°  0'  0.0000", E   4° 19' 43.3200"
S  52°  0'  0.0000", E   4° 19' 43.3200"
N  52°  0'  0.0000", W   4° 19' 43.3200"
S  52°  0'  0.0000", W   4° 19' 43.3200"
N  45°  0'  0.0000", E 180°  0'  0.0000"
S  45°  0'  0.0000", W 180°  0'  0.0000"
S  50° 27' 24.1200", E   4° 19' 43.3200"
```

Implement:

1. a function `dd_dms` that converts a value in decimal degrees (e.g. 4.3287) to a 3-tuple with (degrees, minutes, seconds), as *unrounded* floating point values
2. a function `format_dms`, that returns for the input – an ordinate given as tuple of (degrees, minutes, seconds) – a formatted string
3. a function `format_dd_as_dms`, that converts the coordinate to a formatted string (using the functions `dd_dms` and `format_dms`)

See the docstrings of the functions for the exact specifications.

In module `distance.py`: Write the function for calculating the distance (as float) between 2 points. Implement the haversin function for this.

Given two coordinates: (ϕ_1, λ_1) and (ϕ_2, λ_2):

$$\Delta\phi = \phi_2 - \phi_1; \Delta\lambda = \lambda_2 - \lambda_1$$

The haversin formula that can be used to calculate the distance ($\Delta\sigma$) is as follows:

$$\Delta\sigma = 6371.0 \times 2 \arcsin\left(\sqrt{\sin\left(\frac{\Delta\phi}{2}\right)\sin\left(\frac{\Delta\phi}{2}\right) + \cos\phi_1 \cos\phi_2 \sin\left(\frac{\Delta\lambda}{2}\right)\sin\left(\frac{\Delta\lambda}{2}\right)}\right).$$

The `haversin` function calculates the distance between two points for which spherical coordinates are given as tuple and returns the distance between them in kilometers. Note that you correctly need to handle converting from degrees to radians!

In module `query.py`:

Write the program that ask the user for input of 2 places by implementing the `main` function. When the program starts, it greets the user with the message: '`I will find the distance for you between 2 places.`' Then the program ask the user twice to input a place name (note, `<n>` is replaced by either `1` or `2`):

```
Enter place <n>?
```

The program subsequently uses the Nominatim OpenStreetMap database to query for a WGS'84 location of the given place (see the function `nominatim` in the `nominatim.py` module). In case the placename can not be found in the internet database for a placename, the program notifies the user of this fact: '`I did not understand this place: <placename entered>`'. It subsequently asks again the user to input 2 places. Otherwise the program computes the distance and prints the distance in kilometer, rounded to one decimal: '`The distance between <place1> and <place2> is <distance> km`'. Then the program asks the user again to input two places.

If the user gives as input '`quit`' for one of the 2 places, the program terminates after giving the user a final greeting of '`Bye, bye.`'. Note: The user is expected to give input for *both* the 2 places, before the program will quit (in which either one of the two inputs reads '`quit`').

A sample run of the program is as follows:

```
I will find the distance for you between 2 places.
Enter place 1? Delft
Enter place 2? Bratislava
Coordinates for Delft: N  51° 59' 58.0459", E   4° 21' 45.8083"
Coordinates for Bratislava: N  48°  9'  6.1157", E  17°  6' 33.5027"
The distance between Delft and Bratislava is 1003.4 km
Enter place 1?
Enter place 2? quit
Bye bye.
```

And another run:

```
I will find the distance for you between 2 places.
Enter place 1? where is this place?
Enter place 2?
I did not understand this place: where is this place?
I did not understand this place:
Enter place 1? quit
Enter place 2?
Bye bye.
```

Make sure the program output is *exactly* the same for the same inputs as in the sample runs!

Start from the skeleton given on Brightspace. The docstrings and comments specify the exact behaviour of the functions. Do not use any other imports than already given in the skeleton code.

Query module

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


from nominatim import nominatim
# from nominatim_offline import nominatim # can be used for testing if you are offline
                                          # or if the online Nominatim service does not work
from dms import format_dd_as_dms
from distance import haversin


def query():
    """Query the WGS'84 coordinates of 2 places and compute the distance
    between them.

    A sample run of the program:

I will find the distance for you between 2 places.
Enter place 1? Delft
Enter place 2? Bratislava
Coordinates for Delft: N  51° 59' 58.0459", E   4° 21' 45.8083"
Coordinates for Bratislava: N  48°  9'  6.1157", E  17°  6' 33.5027"
The distance between Delft and Bratislava is 1003.4 km
Enter place 1?
Enter place 2? quit
Bye bye.

    And another run:

I will find the distance for you between 2 places.
Enter place 1? where is this place?
Enter place 2?
I did not understand this place: where is this place?
I did not understand this place:
Enter place 1? quit
Enter place 2?
Bye bye.

    """
    pass


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

Distance module

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

from math import radians, cos, sin, asin, sqrt


def haversin(latlon1, latlon2):
    """
    Calculate the great circle distance between two points
    on the earth (specified in decimal degrees)

    arguments:
        latlon1 - tuple (lat, lon)
        latlon2 - tuple (lat, lon)

    returns:
        distance between the two coordinates (as float, *not* rounded)
    """
    pass
```

```
def _test():
    # You can use this function to test the distance calculation
    pass


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

DMS module

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


def dd_dms(decdegrees):
    """Returns tuple (degrees, minutes, seconds) for a value in decimal degrees

    Arguments:

    decdegrees -- float that represents a latitude or longitude value

    returns:

    3-tuple of *not* rounded floats (degrees, minutes, seconds)
    """
    pass


def format_dms(dms, is_latitude):
    """Returns a formatted string for *one* part of the coordinate.

    Arguments:

    dms -- tuple of floats (degrees, minutes, seconds)
           that represents a latitude or longitude value
    is_latitude -- boolean that specifies whether ordinate is latitude or longitude

    If is_latitude == True dms represents latitude (north/south)
    If is_latitude == False dms represents longitude (east/west)

    returns:

    Formatted string
    """
    pass


def format_dd_as_dms(coordinate):
    """Returns a formatted string for a coordinate

    Arguments:

    coordinate -- 2-tuple: (latitude, longitude)

    returns:

    Formatted string
    """
    pass


def _test():
    """Test whether the format_dd_as_dms function works correctly
    Expected output:

N   0°  0'  0.0000", E   0°  0'  0.0000"
N  52°  0'  0.0000", E   4° 19' 43.3200"
S  52°  0'  0.0000", E   4° 19' 43.3200"
N  52°  0'  0.0000", W   4° 19' 43.3200"
S  52°  0'  0.0000", W   4° 19' 43.3200"
N  45°  0'  0.0000", E 180°  0'  0.0000"
S  45°  0'  0.0000", W 180°  0'  0.0000"
S  50° 27' 24.1200", E   4° 19' 43.3200"

    (note, in PyCharm you can view the whitespace characters in a text file
     by switching on the option View → Active Editor → Show Whitespace)
    """
    coordinates = (
        (0.0, 0.0),
        (52.0, 4.3287),
        (-52.0, 4.3287),
        (52.0, -4.3287),
        (-52.0, -4.3287),
        (45.0, 180.0),
        (-45.0, -180.0),
        (-50.4567, 4.3287),
    )
    for coordinate in coordinates:
        print(format_dd_as_dms(coordinate))


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

Nominatim module

```
# GEO1000 - Assignment 2

from urllib.request import urlopen, URLError, Request
from urllib.parse import quote
import json


def nominatim(place):
    """Geocode a place name, returns tuple with latitude, longitude
    returns empty tuple if no place found, or something went wrong.

    Geocoding happens by means of the Nominatim service.
    Please be aware of the rules of using the Nominatim service:

    https://operations.osmfoundation.org/policies/nominatim/

    arguments:
        place - string

    returns:
        2-tuple of floats: (latitude, longitude) or
        empty tuple in case of failure
    """
    endpoint = "http://nominatim.openstreetmap.org/search?q="
    params = "&format=json"
    url = endpoint + quote(place) + params
    req = Request(
        url=url,
        data=None,
        headers={"User-Agent": "Nominatim Geocode TU Delft Python GEO1000 Exercise"},
    )
    try:
        # try to fetch
        f = urlopen(req)
        lst = json.loads(f.read().decode("utf-8"))
        loc = tuple(map(float, [lst[0]["lat"], lst[0]["lon"]]))
    except:
        # when something goes wrong,
        # e.g. no place found or timeout: return empty tuple
        return ()
    # otherwise, return the found WGS'84 coordinate
    return loc


def _test():
    # Expected behaviour
    # unknown place leads to empty tuple
    assert nominatim("unknown xxxyyy") == ()
    # delft leads to coordinates of delft
    assert nominatim("delft") == (52.0114017, 4.35839)

    # print output for 4 cities
    for name in ['bratislava', 'delft', 'prague', 'new york']:
        print(name, ":", nominatim(name))


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

3 A stack of squares – patterns.py (35 points)

Write a program to produce a text file with each of the following patterns with stacked squares (see Figure 4, 5 and 6). The scale factor to determine the size of the squares (i.e. the size of the square compared to the size of the step before) is configurable and is set to 0.45 as default.

The rectangle with p_1, p_2, p_3, p_4 from Figure 3 can be represented in Well Known Text (WKT) format as:

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

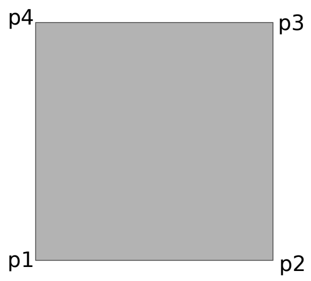

Figure 3: Square with coordinates

The program should write for every pattern its output to a text file that can be read in QGIS[1] using the 'Add Delimited Text Layer'[2] input option. Therefore the very first line of this text file should read `steps_left;geometry`.

The rest of the text file contains the step (for which the square is generated) and the well known text of the squares generated, separated by a semicolon (;), with every square on its own line (note, the step numbers for pattern A are visible in Figure 4). Do not forget to add an end of line character (\n) when you write to the text file.

[1] http://qgis.org/

[2] https://docs.qgis.org/3.40/en/docs/user_manual/managing_data_source/opening_data.html#importing-a-delimited-text-file

The order of the lines corresponds to the drawing order in QGIS: Squares listed later in the file will be rendered on top of those listed earlier. This means that squares may visually obscure other squares (unless styled with transparency).

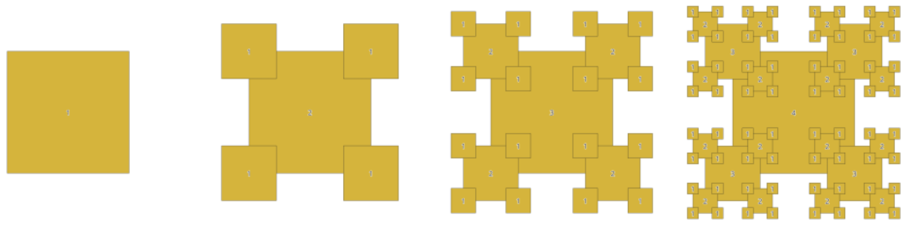
Figure 4: Pattern A

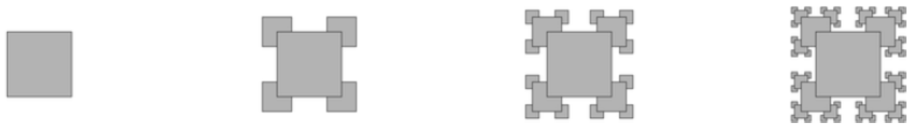
Figure 5: Pattern B

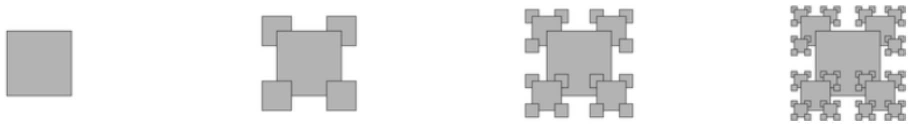
Figure 6: Pattern C

The steps you need to take:

- Make a function that computes the four corners (as four 2-tuples) of a square given its center and half side length.
- Make a function that returns the Well Known Text string of a rectangle. Make sure that the coordinates are ordered counterclockwise, as in the example of Figure 3. Also, note that the coordinates of p_1 are put twice.

  The coordinates should appear in the string with 6 digits behind the comma.
- Make for each of the 3 different patterns a separate *recursive* function (`pattern_a`, `pattern_b` and `pattern_c`) that produces a text file which corresponds to the correct pattern.
- Finish the partly given `main` function that writes the first line for every output file and calls the respective function for producing every pattern.

Note: for `n=0` no squares are produced, `n=1` produces 1 level of squares (with 1 square), `n=2` produces 2 levels of squares (with 5 squares), etc.

Start from the following skeleton. Do not use any imports.

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


def square_corners(center, half_size):
    """
    Computes the four corners of a square given its center and half side length.

    Arguments:
        center: tuple of floats (x, y) - center coordinates of the square
        half_size: float - half the length of one side of the square

    Returns:
        tuple of 4 points (p1, p2, p3, p4) in counterclockwise order:
            p1 = bottom left
            p2 = bottom right
            p3 = top right
            p4 = top left
    """
    pass


def wkt(p1, p2, p3, p4):
    """
    Returns the Well Known Text (WKT) representation of a square defined by four
        corner points.

    Arguments:
        p1--p4: 2-tuples of floats representing the square corners in
            counterclockwise order:
            p1 = bottom left
            p2 = bottom right
            p3 = top right
            p4 = top left

    Returns:
        str: WKT string in the format:
            POLYGON((x1 y1, x2 y2, x3 y3, x4 y4, x1 y1))
            with coordinates formatted to six decimal places.
    """
    pass


def pattern_a(remaining_steps, c, size, scale_factor, file_nm):
    """
    Recursively draws squares in all four corners of the current square.

    Arguments:
        remaining_steps: int - number of recursive steps left
        c: tuple - center coordinates of the current square
        size: float - half side length of the current square
        scale_factor: float - multiplier to determine size of next square (e.g.
            0.75 shrinks)
        file_nm: str - output file name

    Returns:
        None
    """
    pass


def pattern_b(remaining_steps, c, size, scale_factor, file_nm):
    """
    Recursively draws squares in all four corners, then writes the current square.

    Arguments:
        remaining_steps: int - number of recursive steps left
        c: tuple - center coordinates of the current square
        size: float - half side length of the current square
        scale_factor: float - multiplier to determine size of next square
        file_nm: str - output file name

    Returns:
        None
    """
    pass
```

```
def pattern_c(remaining_steps, c, size, scale_factor, file_nm):
    """
    Recursively draws squares in top corners first, writes the current square, then
         bottom corners.

    Arguments:
        remaining_steps: int - number of recursive steps left
        c: tuple - center coordinates of the current square
        size: float - half side length of the current square
        scale_factor: float - multiplier to determine size of next square
        file_nm: str - output file name

    Returns:
        None
    """
    pass


# note, main has optional arguments, see Sec 13.5 of ThinkPython2
def main(n=3, c=(0.0, 0.0), size=10.0, scale_factor=0.45):
    """
    Entry point of the program. Initializes output files and triggers square
        drawing patterns.

    Arguments:
        n: int - number of recursive steps to perform
        c: tuple - center coordinates of the initial square
        size: float - half side length of the initial square
        scale_factor: float - multiplier to determine size of next square (default
            0.45)

    Output:
        Each output file begins with a header line:
            steps_left;geometry
        The order of lines corresponds to drawing order in QGIS:
        squares listed later will be rendered on top of earlier ones.
    """
    funcs = [pattern_a, pattern_b, pattern_c]
    file_nms = ["pattern_a.txt", "pattern_b.txt", "pattern_c.txt"]

    for func, file_nm_out in zip(funcs, file_nms):
        # Finish this function (do *not* change the lines already given,
        # but replace the pass statement below in this function
        # and remove this comment)
        pass


if __name__ == "__main__":
    main(3)
```