Assessment 2: Introduction to Spatial Programming - Trajectory analysis

1 Introduction

This is an individual assessment. It is worth 20% of your final class mark. No late submissions will be accepted.

2 Aim

The aim of this assessment is to gain experience with the basic elements of Python, namely working with Python data types, writing functions, handling strings, reading a CSV file, writing out to a plain text file, and becoming familiar with conditional execution and iteration. It will also provide an opportunity to gain experience in comprehending program requirements, designing a small program, programming defensively and testing the program.

A function is "a block of **organised**, **reusable** code that is used to perform a **single**, **related** action. Functions provide better modularity for your application and a high degree of code reusing". It is extremely important to organise your code to make it intuitive and readable. A simple way is by using functions, yet how to organise your code into blocks of functions is worth careful planning, typically even before you start to program. Without using any function, it would be hard to see the logic flow of the program, thus making it unreadable and non-reusable.

Design is a core component of programming, which reflects the style of the programmer. In Python, a set of guiding preferences has been articulated as *The Zen of Python*:
https://www.python.org/dev/peps/pep-0020/#the-zen-of-python.

3 Assessment

Write a fully documented Python console program named **assessment2.py** that will perform the tasks below. Take note that the specified function **input arguments** and **output data types** are important in order to follow test rules in the final task.
Input data and necessary files are contained in the Git repository for Assessment 2.

4 Questions

4.1 Task 1: Pre-processing and data preparation

1. Write a function named `import_csv(filename)`, to read GPS trajectory data from `trajectory_data.csv`, returning one Python variable with an appropriate data structure holding all of the CSV's data. For example:

```
def import_csv(filename):
'''this function reads data from a CSV file into a nested list'''
    ...
    return nested_list
```

Instruction: *trajectory_data.csv* stores a set of GPS trajectories, and each trajectory has one or more nodes connected by trajectory segments. Location (in latitude and longitude) and time information is stored for each node. A suggested way to store the information is by using a nested list (similar to the data structure of CSV files), although a dictionary (similar to that of a JSON), or a more complex structure using classes can be used as well if you find this more intuitive.

Be careful! Planning ahead is important, a more complex data structure can make later steps harder to manage, and is not necessarily a better way to perform simple tasks in small programs. Code that is written with less lines, with well thought out, modular abstraction is favoured over complicated implementations and over engineering. Furthermore, refrain from using libraries such as *Pandas*, the function of data storage in this task is **strictly limited** to Python's base types (lists, tuples, dictionaries, classes). **Submissions using Pandas will not be accepted in this Assignment**.

Note: If you do decide to use classes for this function, your first test case when running **assessment2_test.py** will fail, you can ignore this!

2. Write a function named `project_coordinate(from_epsg, to_epsg, in_x, in_y)` to use an existing Python module to project a latitude and longitude coordinate to an X and Y coordinate (in a UTM projection, in meters) through reference system transformation. The output of the function should return a single tuple of (x, y) in the output projection.

   Instruction: Latitude and longitude represent locations from the surface of an ellipsoid, and sometimes it is necessary to adopt another reference system and change the way of representing locations. For example, a map projection is the process to project, or transform, the locations on the earth's surface into locations on a plane in order to create maps. An interesting explanation can be found at http://lyzidiamond.com/posts/4326-vs-3857. A suggested module for this task is *pyproj*, which allows you to specify both input and output reference systems and perform the transformation.

   There are a huge number of reference systems out there; refer to spatialreference.org or epsg.io. The reference system used in *trajectory_data.csv* is WGS84 (you must find the corresponding EPSG code by yourself to use in your transformation). Since the data was collected in Beijing, China, you should use the New Beijing reference system (code: EPSG:4796).

3. The next step is to write one (or more if necessary) function of your own choice in naming and arguments, incorporating the previous functions to:

   (a) Import the original CSV file.

   (b) Process each coordinate into the new chosen projection system.

   (c) Output to a new CSV with two new columns (X_UTM, Y_UTM) containing the projected coordinate for each feature, and also preserving the original latitude and longitude columns. Name the output file **assessment2.csv**.

   (d) Store the data structure containing the output data into **one variable** for use in the next tasks.

4.2 Task 2: Data processing functions

1. Write a function `compute_distance(from_x, from_y, to_x, to_y)` to compute a distance between two UTM coordinate observations. You can assume that the input observations are measurements in metres. The function should output one measurement, also in **metres**.

   Instruction: The task can be done either by using an existing module or a manually implemented function. Even though the part of the code is short, it is still recommended to put it as an individual function.

   Hint: Is there anything we need to be careful with here? There might be an opportunity to include some simple exception handling in this function to prevent errors later in our program.

2. Write a function `compute_time_difference(start_time, end_time)` to compute the time difference between two timestamped observations (use the format of column time in the data CSV file) and return the value in **seconds**.

   Instruction: Be careful with calculating time differences between values in such a timestamp format, as it is easy to overlook special situations. You can safely assume that the time difference between any two adjacent node in a trajectory is within 24 hours. The time difference calculated is the time spent of a segment, which will be used later for calculating segment speed. As above, are there any exception handling cases apparent here?

3. Write a function `compute_speed(total_distance, total_time)` to compute the speed, given the difference in distance and time obtained from the above functions. The function should return **metres per second**.

4.3 Task 3: Analytical processing

You will now need to think of a way to work with trajectories, as now instead of storing individual observations, we are interested in the relationship between two observations (segments), and the total trajectory. You will need to consider ways to loop over your list to obtain these results for both, the segments and trajectory. The computational complexity of this task/algorithm is not considered in marking this assessment, however if your code runs for an excessive duration (more than 10 minutes, for example), this, along with the code will impede your mark.

1. Compute the total length of the trajectory.
2. Compute the length of the trajectory longest segment, and its index in the trajectory (show the first one if there are multiple).
3. Compute the average sampling rate for the trajectory in seconds. This is the sampling time between each pair of adjacent observations in one trajectory.
4. Compute the speed of each segment, and identify the indices of segments that:
   (a) Has the lowest speed.
   (b) Has the highest speed (again, the first one if there are multiple for both cases).
5. And for all trajectories:
   (a) Compute the total length of all trajectories.
   (b) Identify the index of the longest trajectory, and compute its average speed.
   Instruction: It is suggested that each subtask be solved by its own function. You may want to call other functions to use their outputs inside some functions. Take care of error handling in your code, and document reasonable assumptions that you make about the processing of the data, e.g., handling CSV data with empty cells. Again, this requires comprehensive understanding of the problem at hand, and a clear design of work flow and program structure.
   (c) Print the output according to the sample output below and output it to **assessment2_out.txt**.

4.4 Task 4: PEP8 conformity and standardisation

Now that you have all the functions written for the three above tasks, we need to make the code understandable for the rest of our "hypothetical" work environment. Imagine that you are submitting this code to then be used, or further developed on by another individual in your workplace. In this task we will focus on a few ways to do this:

1. Write a `main()` function. This function should only call the most important elements of this assessment. This way, if someone needs to debug an error, they can see the flow of the program from one task to another, or they can easily bypass a task if it's not suitable for their current project. Variables can be passed between functions of major tasks, and constants can be easily changed from within it. One example of how to do this is below, you can adapt this to work with your program. The creation of **task_1()**, **task_2()**, etc is not necessary if you can call your other written functions in a neat and tidy manner.

```
def main():
    # Setup constant
    csv_filename = "trajectory_data.csv"
    filename = os.path.join(os.getcwd(), csv_filename)

    # Run tasks
    reprojected_data = task_1(filename)
    task_2()
    task_3(reprojected_data)

    # Finished
    print("Program complete")
```

In Python, unlike other programming languages, the `main()` is not called by default if you run the program from the command line. In order to do this, the program identifies that it's the "main" script being called, so it is necessary also to include this if statement at the bottom of your program to let it know to run it (for example if you were importing only the functions from another program, the `main()` would not run because it is not the primary program being executed). In assessment 1, we also saw this in the example python program. **Add this to the bottom of your .py file.**:

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

2. Make sure your code conforms with Python PEP8 standards. https://www.python.org/dev/peps/pep-0008/.

   These are naming and formatting standards developed by the Python community. Adhering to these will allow others to understand your code better and give you a better eye at understanding someone else's work. Proper commenting and annotation is part of this, make sure your code is easy to read (variable and function names are readable), or have sufficient commenting.

   There are many tools you can use to check your code automatically, that will tell you on which line of your code you have broken a PEP8 rule that you can fix. One example of this is **flake8**: https://github.com/PyCQA/flake8. After installing (you can do this via Anaconda in your environment!), run this on your python program from the command line by typing:

```
flake8 assessment2.py
```

4.5 Task 5: Unit testing and submission

1. Unit testing is a method to write test cases for each function that compare real world examples with real world answers and checks these with the functions that have been written. This makes sure that your functions are providing a consistent output, even after altering them in a development setting. Say for example, the imported library you relied upon to project coordinates in question 2 updated, and now contains an error that went unchecked, which now affects your output, or someone altered your function and wants to see if it still produces the same results. You want to be able to create test cases that fail if there is any change in your expected output. In the git repository along with this instruction sheet there is a file called **test_assessment2.py**, which will input your functions, and test them.

   Make sure your assessment file is in the same folder as the test script, and you are inside your Anaconda environment for the project. On the command line, type:

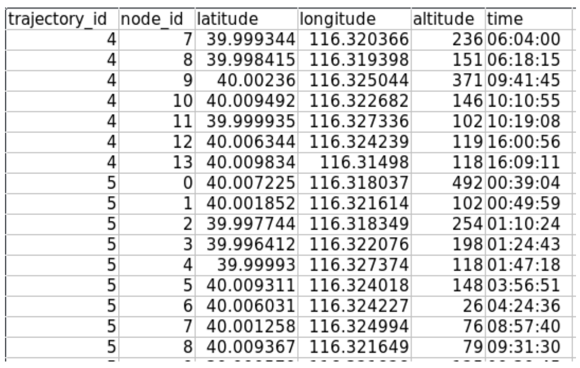
```
python test_assessment2.py -v
```

   For now, testing your functions with this script by running it, and passing them is all that is required to pass this task. We will use unit testing again in later assessments, so it is worthwhile to open the test script and understand what is happening inside of it.

2. Submit according to the submission guidelines.

5 Sample data

Below is an example of the input CSV data. Note that the CSV has a header, and the separator is a "," (csv stands for comma-separated values). You must keep both the latitude, longitude and altitude columns, as well as the added UTM_X and UTM_Y columns in your output csv (**assessment2.csv**).



trajectory_id	node_id	latitude	longitude	altitude	time
4	7	39.999344	116.320366	236	06:04:00
4	8	39.998415	116.319398	151	06:18:15
4	9	40.00236	116.325044	371	09:41:45
4	10	40.009492	116.322682	146	10:10:55
4	11	39.999935	116.327336	102	10:19:08
4	12	40.006344	116.324239	119	16:00:56
4	13	40.009834	116.31498	118	16:09:11
5	0	40.007225	116.318037	492	00:39:04
5	1	40.001852	116.321614	102	00:49:59
5	2	39.997744	116.318349	254	01:10:24
5	3	39.996412	116.322076	198	01:24:43
5	4	39.99993	116.327374	118	01:47:18
5	5	40.009311	116.324018	148	03:56:51
5	6	40.006031	116.324227	26	04:24:36
5	7	40.001258	116.324994	76	08:57:40
5	8	40.009367	116.321649	79	09:31:30

The column trajectory holds a unique ID of the trajectory. The column *node_id* holds an ordinal identifier of the timestamped location within each trajectory. The output of your program must **STRICTLY** follow the format below, including symbols, spaces, and rounding digits, but not font:

```
Trajectory 1's length is 20525.25m.
The length of its longest segment is 7460.19m, and the index is 8.
The average sampling rate for the trajectory is 3317.00s.
For the segment index 1, the minimal travel speed is reached.
For the segment index 7, the maximum travel speed is reached.
----
Trajectory 2's length is 4967.73m.
The length of its longest segment is 924.42m, and the index is 13.
The average sampling rate for the trajectory is 572.14s.
For the segment index 1, the minimal travel speed is reached.
For the segment index 13, the maximum travel speed is reached.
----
...
----
The total length of all trajectories is 278094.47m.
The index of the longest trajectory is 7, and the average speed along the trajectory is 2.45m/s.
Program complete
```

The numbers from the sample output, however, may NOT be the actual result. Note that the program will be tested also against a different dataset, so the values generated have to be automatic. The first trajectory in the printed output gets the number 1 in your output, the second trace the number 2 and so forth (even though in your data, your index will usually start at 0). The precision for the lengths should be **exactly** two digits after the decimal point.

6 Marking scheme

Your assessment will be marked out of 20 according to the following criteria:

- Your program executes correctly and works for the supplied sample data.
- Your program computes the correct values for the supplied sample data as well as for any other trajectory lists stored according to the same specifications.
- Your program is efficient, short, and compact. (e.g., the core functions of the program can be written in around 150 lines, not including comments).
- Your program is well structured and documented, allowing another developer to understand what is happening and why.

Note that only actual code counts as lines. Comments are not counted as lines, and you will lose marks if your comments are too sparse or missing. A very rough guideline: a program with 150 lines of codes has also the same amount of comments. The table below shows detailed marking criteria associated with marks:

Criteria	**Sub-criteria**	**Mark**
Correctness	Task completion, output correctness, unit tests pass, runs from console	8
	Interoperability, special case and exception handling, use of main()	2
Design	Abstraction of problem, appropriate use of functions	3
	Readability, elegance, code structure, PEP-8 conformity	3
Documentation	Understandability of function/variable names (self-documenting)	2
	Commenting and documentation	2

Good self-documenting:

```
def compute_distance_metres(coordinate_one_m, coordinate_two_m):
    pass
```

Acceptable, only with proper docstring:

```
def comp_dist(c1, c2):
    ''' computes distance in metres between two coordinate pairs
    c1, c2: coordinates in (x,y) format in metres '''
```

Bad (“mysterious names”):

```
def cd(a, b):
    pass
```

Code that is difficult to understand or uses poor programming style will lose marks irrespective of length. Code that uses appropriate defensive programming techniques (regardless of lines spent, as long as they are implemented in efficient ways) may be bonified.

7 Tips

Make sure you check carefully that your code works correctly, and produces sensible answers (you may want to test it on a variety of trace lists). You might also like to keep a design notebook, where you keep track of your notes and sketches about the project. This will be good practice for future assessments, which will be completed in groups.

8 Submission

Submit a single **.zip** file (where **studentno** is your student number) containing the following files for submission:

- **assessment2.py** (your code)
- **assessment2.csv** (your projected coordinate CSV)
- **assessment2_out.txt** (the output of your analysis)
- **environment.yaml** (your Anaconda environment file)

studentno_A2.zip

You submit your file by uploading the file to Canvas. Only the last upload before the submission deadline will be marked.